AED II

The FurfiOS Corporation

Junio 2021

Índice general

1. Especificación **3**
- 1.1. Introducción . . . 3
- 1.2. Comportamiento Automático . . . 7

2. Diseño **9**
- 2.1. Análisis de Complejidad . . . 9
- 2.2. Diseño jerárquico de TADs . . . 13

3. Estructuras de Datos **17**
- 3.1. Conjuntos y Diccionarios . . . 17
- 3.2. Árbol Binario de Búsqueda (ABB) . . . 18
- 3.3. Árboles AVL . . . 21
- 3.4. Radix Searching . . . 23
  - 3.4.1. Árboles de búsqueda digital . . . 23
  - 3.4.2. Tries . . . 24
- 3.5. Hashing . . . 27
  - 3.5.1. Concatenación . . . 29
  - 3.5.2. Direccionamiento abierto . . . 30
- 3.6. Colas de Prioridad . . . 32

4. Sorting **35**
- 4.1. Heap Sort . . . 35
- 4.2. Merge Sort . . . 36
- 4.3. Quick Sort . . . 37
- 4.4. Árboles de decisión . . . 38
- 4.5. Bin Sorting . . . 39
  - 4.5.1. Radix Sorting . . . 39

5. Dividir y Conquistar **41**
- 5.1. Método de sustitución . . . 42
- 5.2. Árbol de Recurrencia . . . 43
- 5.3. Método Maestro . . . 45

6. Memoria Secundaria **46**
- 6.1. Ordenamiento Externo . . . 46
  - 6.1.1. Fusión Múltiple Equilibrada . . . 47
  - 6.1.2. Selección por sustitución . . . 48
  - 6.1.3. Fusión Polifásica . . . 50
- 6.2. Búsqueda Externa . . . 50
  - 6.2.1. Acceso Secuencial Indexado . . . 51
  - 6.2.2. Árboles B . . . 51
  - 6.2.3. Hashing Extensible . . . 55

7. Alternativas a los AVL **57**
- 7.1. Skip Lists . . . 57
- 7.2. Splay Trees . . . 58

Este apunte fue hecho en base a las clases teóricas de los profesores Dr. Esteban Feuerstein y Dr. Emmanuel Iarusi del Primer Cuatrimestre 2020, complementado con bibliografía:

- Capítulos 2, 3, 4, 5.6, 5.7 del Brassard [4].
- Capítulos 4, 11 del Cormen [3].
- Capítulos 4.7, 5.3, 8 del Aho [11].
- Capítulos 9, 10, 13, 15, 17, 18 del Sedgewick [10].
- Capítulo 6.2.4 del Knuth [9].
- Capítulo 4.3 del Tarjan [6]

También se utilizó parte de los apuntes de AED3 (para lo de modelo uniforme y modelo logarítmico) y material extraoficial.

Capítulo 1

Especificación

1.1. Introducción

Típicamente, nos enfrentamos a la siguiente situación. Tenemos un **problema** que se nos es presentado en una manera difusa, vaga, y se pretende resolver a través de la computadora, de forma eficiente. Lo problemático es transitar este recorrido que va desde la definición informal del problema a su resolución computacional. Este camino tiene una serie de pasos que, típicamente, los llamamos: **especificación**, **diseño** e **implementación**. Esta serie de pasos son los que vamos a recorrer a lo largo de esta materia.

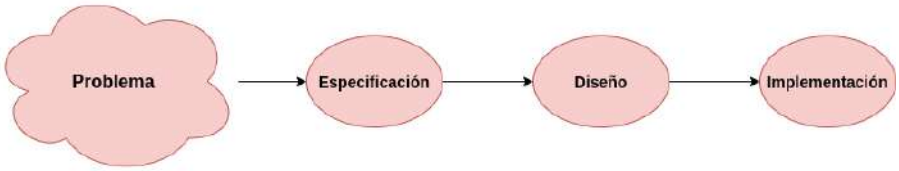

Figura 1.1: Etapas en la resolución de un problema

En esta materia vamos a estudiar **algoritmos** y estructuras de datos. Recordemos qué es un algoritmo:

Definición 1.1.1. *Un algoritmo es un procedimiento para resolver un problema, descripto por una secuencia ordenada y finita de pasos bien determinados que nos llevan de un estado inicial a uno final en un tiempo finito.*

Un algoritmo siempre debe brindar una respuesta, siendo posible que la respuesta sea "no hay respuesta". La descripción debe ser clara y precisa, sin dejar lugar a la utilización de la intuición o la creatividad. Por ejemplo, una receta es una algoritmo si no dice "sal a gusto".

Los lenguajes naturales tienen una *semántica difusa* (polisemia, ambigüedad). Por lo tanto, para poder formalizar la resolución de un problema, primero debemos describirlo en un lenguaje formal de especificación, que sea preciso, es decir, que pueda interpretarse sin lugar a dudas. Es importante que al momento de describir los problemas, no caigamos en la *sobre-especificación*, ya que si bien es sencillo realizar una descripción precisa y libre de ambigüedades si damos todos los detalles acerca de la representación del problema, estaríamos limitando la forma en la que podemos resolver el problema. Este salto entre un lenguaje formal y uno natural se conoce como *brecha semántica*.

A través de los mecanismos de especificación que vamos a estudiar, vamos a ir introduciendo algunos conceptos importantes como la **abstracción**, la **modularización** y el *ocultamiento de la información*. Notemos que cuando estamos especificando un problema nos interesa describir "el qué" y no "el cómo", por lo que no tiene sentido preguntarnos en esta etapa si la especificación es o no *eficiente*.

La herramienta principal que utilizaremos para especificar problemas se conoce como Tipos Abstractos de Datos (**TADs**), que son modelos matemáticos que se construyen con el fin de exponer los aspectos relevantes del problema a resolver. Un **tipo abstracto** es un conjunto de valores cuya "forma" desconocemos, asociados a operaciones que nos permiten obtener información sobre los valores. Este lenguaje axiomático permite estudiar otras formas de demostración muy útiles, como la *inducción estructural*, y cuenta con la ventaja de que no requiere de tipos primitivos que deban definirse por fuera del mismo, además de ser una herramienta tanto flexible como general. Veamos un ejemplo:

Descripción : *El dueño de un restaurant quiere asegurarse de que los pedidos sean atendidos con prolijidad. Los mozos llevan los pedidos hasta la cocina donde los colocan. Cuando el cocinero se libera, saca el primer pedido y prepara el plato indicado. El dueño quiere saber cuál es el próximo plato a preparar, cuántos pedidos atiende el cocinero cada día y cuál fue el día con menos pedidos.*

Podemos notar que hay ciertos elementos en esta descripción del problema que realmente no son necesarios para el modelado del mismo. Por ejemplo, en la descripción del problema se nos habla de un *dueño*, pero este no realiza ninguna operación, por lo que no nos interesa modelarlo. Ahora nos toca realizar un proceso de **abstracción** para identificar aquellos elementos y operaciones que terminan definiendo al problema. En particular, este problema podemos identificar el uso de platos, días y un restaurant. ¿Qué podemos decir sobre estos elementos?

- Los días pasan, por lo que vamos a querer contarlos. Para modelar este comportamiento nos alcanza con renombrar el TAD Día como Nat: **TAD** Día **ES** Nat.
- Solo nos interesa poder diferenciar los platos entre sí. **TAD** Plato **ES** String.

¿Por qué tenemos que renombrar estos TADs y no utilizar String o Nat directamente? Lo que queríamos hacer era utilizar un lenguaje de especificación que nos permita abstraer ciertos conceptos, por lo que usar String directamente no sería correcto. Para el restaurant no tiene sentido un String, pero los Platos sí. De esta manera, nos alejamos de la manera en la que representamos los datos en una computadora y nos acercamos a la definición del problema.

Para especificar un TAD, debemos primero debemos definir la *signatura* del mismo, es decir, las operaciones que ofrece el TAD:

TAD Restaurant

géneros restaurant

operaciones

cant_platos_pendientes : restaurant $\longrightarrow$ nat	
próximo_pedido : restaurant r $\longrightarrow$ plato	{cant_platos_pendientes(r)>0}
preparar_plato : restaurant r $\longrightarrow$ restaurant	{cant_platos_pendientes(r)>0}
tomar_pedido : restaurant $\times$ plato $\longrightarrow$ restaurant	
nuevo_día : restaurant $\longrightarrow$ restaurant	
día_actual : restaurant $\longrightarrow$ día	
platos_por_día : restaurant r $\times$ dia d $\longrightarrow$ nat	$\{d \leq$ día_actual(r)$\}$
inaugurar : restaurant $\longrightarrow$ día	

Fin TAD

Luego, la signatura nos define qué operaciones tiene cada tipo, cuáles son sus parámetros y qué tipos

devuelven. Notemos que las operaciones de los TADs son **funciones totales**, o sea, son funciones que tienen que estar definidas para todos los valores del dominio. Por eso, en casos como preparar_plato(), tuvimos que restringir el dominio. En esta etapa, no sería correcto utilizar una convención de devolver -1 para indicar que no existe resultado (esto sería parte de la etapa de diseño, para el manejo de errores).

Con esto definimos la **sintáctica** del TAD, y lo que nos falta es darle **semántica** o comportamiento a estas operaciones. Para ello, vamos a definir los **axiomas** del TAD. Los axiomas son *fórmulas bien formadas*, es decir, un predicado aplicado a *términos*, donde un término son las variables, las constantes y lo que resulta de aplicar las operaciones. En muchos casos, los axiomas no serán otra cosa que ecuaciones como la siguiente:

$$\text{día\_actual(nuevo\_día(r))} \equiv \text{día\_actual(r)} + 1$$

De esta manera, podemos darle un significado a la operación *día_actual* cuando tenemos una entrada de la forma *nuevo_día(r)* (en este caso significa *día_actual(r) + 1*). Estas ecuaciones permiten aplicar operaciones del TAD a términos.

Comenzamos con los axiomas que corresponden al TAD RESTAURANT:

TAD RESTAURANT

axiomas $\forall r$: restaurant, $\forall p$: plato

día_actual(inaugurar())	≡	0
día_actual(nuevo_día(r))	≡	día_actual(r)+1
día_actual(tomar_pedido(r,p))	≡	día_actual(r)
cant_platos_pendientes(inaugurar())	≡	0
cant_platos_pendientes(tomar_pedido(r,p))	≡	cant_platos_pendientes(r) + 1
cant_platos_pendientes(preparar_plato(r))	≡	cant_platos_pendientes(r) - 1
próximo_pedido(r)	≡	ult(secuencia_de_pedidos(r))
secuencia_de_pedidos(inaugurar())	≡	<>
secuencia_de_pedidos(tomar_pedido(r,p))	≡	p • secuencia_de_pedidos(r)
secuencia_de_pedidos(preparar_plato(r))	≡	com(secuencia_de_pedidos(r))
platos_por_día(d,inaugurar())	≡	0
platos_por_día(d,tomar_pedido(r,p))	≡	platos_por_día(d,r)

platos_por_día(d, preparar_plato(r))	≡	**if** día_actual(r) = d **then**
		platos_por_día(d,r) + 1
		else
		platos_por_día(d,r)
		fi

platos_por_día(d, nuevo_día(r))	≡	**if** día_actual(r)+1 = d **then**
		0
		else
		platos_por_día(d,r)
		fi

($\forall d'$: día) $0 \leq d' \leq dia\_actual(r) \Rightarrow_L$

platos_por_día(r, días_menos_pedidos(r)) $\leq$ platos_por_día(r, d')

Fin TAD

Este último axioma puede parecer un poco raro, al no ser un axioma ecuacional. El motivo es que en la descripción del problema no se nos dice qué hacer en caso de empate, por lo que si en la etapa de especificación definimos cuál debe ser el criterio de desempate, estaríamos sobre-especificando. Otra operación que está axiomatizada de esta forma es *dameUno* del TAD Conjunto(α).

Al momento de axiomatizar, una de las primeras cosas que tenemos que tener en cuenta es que las operaciones que estamos especificando son funciones, así que deberíamos evitar cualquier tipo de inconsistencias. En particular, tenemos que tener en cuenta que no se cuenta con *pattern matching*, dicho de otro modo, todos los axiomas valen a la vez. Tampoco debemos especificar sobre los casos restringidos, ya que están fuera del dominio. Es importante que no caigamos en la *sobre-especifición* ni en la *sub-especificación* (no decir qué valores toma la función para ciertos valores).

Además, se espera mantener el encapsulamiento entre los distintos TADs, por lo que si trabajamos con instancias de un TAD en las operaciones de otro, manipularemos esas instancias a través de sus observadores (y no sus generadores). Por ejemplo:

TAD Restaurant

```
platos_de_cierto_precio(r,c,x) ≡ if ∅?(c) then
                                    ∅
                                 else
                                    if precio(DameUno(c),r) = x then
                                       Ag(DameUno(c), platos_de_cierto_precio(r, SinUno(c), x))
                                    else
                                       platos_de_cierto_precio(r, SinUno(c,x))
                                    fi
                                 fi
```

Es preferible a

```
platos_de_cierto_precio(r, ∅,x) ≡ ∅

platos_de_cierto_precio(r,Ag(p,c), c) ≡ if precio(p, r) = x then
                                           Ag(platos_de_cierto_precio(r,c, x))
                                        else
                                           platos_de_cierto_precio(r,c,x)
                                        fi
```

Fin TAD

Ahora, veamos más en detalle las distintas secciones que tiene un TAD:

- **Parámetros formales**
- **Géneros**: un género es el nombre que recibe el conjunto de valores del tipo. Hay una sutil diferencia entre el nombre del TAD y el género. Si se quiere, pensar en el monoide conmutativo ($\mathbb{N}$, +) (TAD) y en el conjunto de los números naturales (género).
- **Usa**: hace referencia a las operaciones y géneros de otros TADs que utiliza el TAD que queremos definir. Estas operaciones y géneros tienen que estar exportados en el otro TAD. Si se usa todo lo que aparece mencionado, podemos obviar esta cláusula.
- **Exporta**: indica las operaciones y géneros que se deja a disposición de los usuarios del tipo. Esta cláusula tiene un valor por omisión: los géneros, los observadores básicos y los generadores.
- **Igualdad Observacional**: La igualdad observacional es un predicado entre instancias del tipo que

nos dice cuándo son iguales, desde el punto de vista de su comportamiento (semántica) y no desde el punto de vista de la sintáctica. Por ejemplo, la instancia Ag(1, Ag(2, $\emptyset$)) es observacionalmente igual a Ag(2, Ag(1, $\emptyset$)) a pesar de ser sintácticamente distintas. Para definir la igualdad observacional se utilizan a los **observadores básicos**. La $=_{obs}$ es un predicado del metalenguaje, y permite agrupar a las distintas instancias en una misma clase de equivalencia. Vamos a exigir que todas las instancias que sean equivalentes de acuerdo con la igualdad observacional mantengan la *congruencia* al aplicar cualquier función. Recordemos que una función f es congruente con respecto a una relación de equivalencia "$\sim$" si y solo si: $(\forall x, y)(x \sim y \iff f(x) \sim f(y))$.

- **Generadores**: son aquellas operaciones que permiten generar o construir instancias del TAD. Es necesario que el conjunto de generadores esté bien definido, es decir, que entre todos los generadores podemos construir cualquier instancia posible del TAD. Un problema menor, no tan grave, es que una instancia del TAD pueda ser construida de más de una manera. Es importante notar que al aplicar un generador sobre una instancia de un TAD **no se está modificando** la instancia que se recibe como parámetro, sino que se genera una **nueva instancia** basada en la anterior. Recordemos que estamos trabajando con *funciones*, por lo que no existe la noción de *estado*, y que el paradigma funcional trabaja bajo el concepto de *transparencia referencial*, es decir, que los resultados de las funciones solo dependen de sus argumentos.

- **Observadores básicos**: son aquellas operaciones que nos permiten diferenciar instancias del TAD en clases de equivalencia. En general, se axiomatizan en base a todos los generadores.

- **Extiende**

- **Otras operaciones**: son el resto de las operaciones que se necesiten declarar en un TAD. No debería ocurrir que una función que aparezca en esta sección devuelva valores que rompan con la *congruencia* del TAD (si esto ocurre, habría que repensar los observadores). En general, se axiomatizan en base a los observadores, aunque es posible que en algunos casos sea conveniente axiomatizarlos en base a los generadores.

- **Axiomas**: son las reglas que describen el comportamiento desde el punto de vista semántico de los elementos del TAD.

En general, es preferible que el conjunto de generadores y el de los observadores sean *minimales*, permitiendo que los generadores no sean minimales si eso facilita la axiomatización. Si estos no lo fuesen, se corre el riesgo de producir inconsistencias y, además, la redundancia atenta contra la claridad.

1.2. Comportamiento Automático

La idea del comportamiento automático es no modelar operaciones para casos que se dan de forma implícita o automática. Por ejemplo, si cada vez que se da cierta condición A se produce el efecto B a través de una acción C que se da de *forma automática*, seguramente no haga falta hacer alusión a la acción C de ninguna forma para modelar correctamente el objeto de estudio. Veamos un ejemplo.

Descripción del Problema: Se quiere especificar el comportamiento de una fábrica de empanadas que está totalmente automatizada. A medida que se encuentran listas, las empanadas van saliendo de una máquina una a una y son depositadas en una caja para empanadas. En la caja caben 12 empanadas y cuando esta se llena, es automáticamente despachada y reemplazada por una caja vacía. Se quiere saber cuántas cajas de empanadas se despacharon en total y cuántas empanadas hay en la caja que está actualmente abierta.

El enunciado nos dice que cuando la caja actual se llena es *automáticamente* despachada y reemplazada por una caja vacía. Por lo tanto, no debemos definir una operación de *despacharCaja*, ya que estaríamos permitiendo la existencia de instancias en las que esto no ocurra. El mayor impacto que tiene el comportamiento automático sobre la especificación de un problema es en los axiomas, porque este comportamiento debe quedar plenamente descripto en los mismos. En resumen, cuando alguna parte del comportamiento del TAD debe ser automática, no deberíamos:

- especificar una acción *manual* para este comportamiento,

- permitir que existan instancias del TAD en las que el comportamiento debería haberse aplicado y no lo hizo,
- restringir acciones que requieran que suceda el comportamiento, ya que este es automático (no existe ninguna instancia para la cual el comportamiento no se haya dado).

Capítulo 2

Diseño

2.1. Análisis de Complejidad

Hasta ahora, estuvimos enfocándonos en modelar correctamente describiendo, mediante un lenguaje formal, *qué* es lo que necesitamos resolver. En esta sección, vamos a orientarnos hacia el *cómo* resolver el problema, es decir al *diseño* de la solución. Para poder elegir una forma adecuada de resolver el problema, debemos tener alguna medida de *eficiencia*, es decir, cuántos recursos requiere el algoritmo para ejecutar.

En general, nos va a interesar poder medir el **tiempo de ejecución** de un algoritmo, pero también hay otros recursos que habitualmente se usan, como pueden ser el uso de **memoria**, cantidad de procesadores, utilización de la red de comunicaciones, etc. Nos podríamos preguntar si es posible optimizar todos estos criterios al mismo tiempo. En general, esto no va a ser posible, ya que el uso de estos recursos suele entrar en conflicto, es decir, vamos a tener un *trade-off* entre la utilización de los distintos recursos. Por lo tanto, vamos a tener muchas soluciones a un mismo problema, que optimicen distintos recursos, y que nos serán de utilidad bajo distintos *contextos de uso*. En general, el recurso que más no va a interesar es el tiempo de ejecución y, en segundo lugar, el espacio utilizado. Esto se debe a que el tiempo no puede ser recuperarse, mientras que el espacio en la memoria sí.

Necesitamos alguna estrategia para poder medir la eficiencia de los distintos algoritmos. Una primer estrategia es la **empírica** (*a posteriori*), que consiste en programar las distintas soluciones y probarlas con la ayuda de una computadora, para un conjunto arbitrario de instancias. El problema de este enfoque es que para poder comparar cualquier algoritmo con otro, primero debemos implementarlo (lo cual lleva tiempo) y encima los resultados que obtenemos nos son generales, ya que dependen del lenguaje de programación, la máquina sobra la que se ejecuta, las posibles optimizaciones utilizadas y las instancias particulares que se utilizaron. Nos gustaría tener una medida más general, que sea independiente de todas estas variables.

Para ello, vamos a medir la complejidad algorítmica de forma **teórica** (*a priori*), que nos permita *estimar* lo que tardaría la ejecución de un algoritmo, sin tener que ejecutarlo ni implementarlo, el cual vale para instancias de cualquier tamaño, es independiente del lenguaje de programación y de la máquina en la que se ejecuta. Lo primero que necesitamos para independizarnos de una computadora en particular es tener un **modelo de cómputo**. Vamos a inventar una máquina teórica, ideal, cuyas características son consensuadas, y vamos a asociar la complejidad de un algoritmo a esa máquina teórica. Para que el análisis valga para instancias de cualquier tamaño, vamos a definir la medida de complejidad en función del **tamaño** de las instancias y no en función del valor de instancias particulares, enfocándonos en un análisis **asintótico** de la complejidad.

Modelo de cómputo

Estamos buscando una medida que sea general, válida para distintas implementaciones del algoritmo e independiente de la máquina en la que se ejecuta. Con este objetivo en mente, vamos a inventar una máquina teórica que vamos a usar como "banco de pruebas" para la ejecución (teórica) del algoritmo. Esta máquina ideal nos va a permitir definir los conceptos de tiempo de ejecución y espacio de memoria

utilizado. Vamos a entender al tiempo de ejecución como la cantidad de pasos o instrucciones que se ejecutan en la máquina teórica para resolver una instancia del problema. De forma similar, podemos definir medir el consumo de memoria de un algoritmo en función de la cantidad de posiciones de memoria utilizados en una ejecución sobre la máquina teórica.

Para definir una unidad de tiempo en la máquina teórica, vamos a utilizar el concepto de **operaciones elementales**. Las operaciones elementales son aquellas que "tardan" una cantidad constante de unidades de tiempo en ejecutarse en el modelo de máquina que estamos definiendo. En general, vamos a tener un conjunto reducido de operaciones elementales y un conjunto de reglas para calcular cuánto tardan aquellas operaciones que no son elementales.

En principio, no existen operaciones cuyo tiempo de ejecución sea independiente de la longitud de sus operandos. Sin embargo, bajo el **modelo uniforme**, podemos asumir que los operandos envueltos en las instancias son de un tamaño *razonable* y tomar al tiempo de ejecución de una operación elemental sobre cualquier operando como constante. Por otro lado, cuando trabajamos con operandos que pueden crecer de forma arbitraria, nos conviene pensar bajo el **modelo logarítmico**. La idea es que si bien no es razonable asumir constante el tiempo de una operación (incluso si es elemental) para operandos de cualquier longitud, sí podemos asumir que el tiempo de una operación elemental es constante para operandos de 1 bit. Luego, podemos medir el tiempo de ejecución de cada operación en función del tamaño de los operandos, medido en **bits**. En general, vamos a trabajar bajo el modelo uniforme y consideraremos como operaciones elementales a las operaciones aritmético-lógica básicas (suma, división, multiplicación, AND, OR), las comparaciones lógicas, las transferencias de control y las asignaciones a variables de tipos básico.

Vamos a utilizar medida del tiempo de ejecución una función $t(I)$ que mida la cantidad de operaciones elementales que se ejecutan para una instancia particular I, considerando que el tiempo de una OE es de una **unidad**. En general, para el análisis del tiempo de ejecución de un algoritmo vamos a ver cuánto cuestan las operaciones individuales, para luego combinar estos costos según la estructura de control involucrada, llamadas a procedimientos y llamados recursivos. Vamos a definir que si P_1 y P_2 son dos fragmentos sucesivos de un algoritmo, podemos calcular el costo total del algoritmo como $t(A) = t(P_1) + t(P_2)$, sin importar si se tratan de instrucciones simples o complicados sub-algoritmos. Notemos que el tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia ($T(n) = n + T(n-1)$), que veremos cómo resolver posteriormente (ver Divide & Conquer).

Tamaño de la entrada

Habíamos dicho que íbamos a definir la complejidad de un algoritmo en función del tamaño de entrada, pero ¿por qué es importante? Lo que queremos es una complejidad relativa al tamaño de entrada porque los valores que podemos medir para instancias particulares dependen del tipo de máquina, del contexto, lo cual es poco generalizable. Queremos una medida que sea más general y que tenga valor para muchas instancias.

El tamaño de entrada formalmente se corresponde con el número de bits necesarios para representar a esa instancia en una computadora, usando algún esquema de codificación. Sin embargo, normalmente vamos a ser menos formales que esto, y vamos a entender como *tamaño* a cualquier entero que de alguna manera mida el número de componentes de una instancia. A veces, cuando hablamos de problemas que involucran enteros, es más natural dar la eficiencia del algoritmo en términos del *valor* de la instancia, en lugar de su tamaño. Notemos que la cantidad de bits que ocupa un entero de valor n es $\lceil \log n \rceil$, por lo que si conocemos el costo en función del valor, fácilmente podemos traducirlo al costo en función del tamaño.

El problema que tenemos es que no todas las entradas del mismo tamaño consumen el mismo tiempo (o espacio). Esto da a lugar a tres medidas particulares para un mismo algoritmo: el análisis del caso **peor**, del caso **mejor** y del caso **promedio**.

Definición 2.1.1. Sea $t(I)$ el tiempo de ejecución de un algoritmo sobre una instancia I. Definimos el tiempo de ejecución del peor caso, del mejor caso y del caso promedio para instancias de un tamaño n como:

$$T_{peor}(n) = \max_{|I|=n} t(I)$$

$$T_{mejor}(n) = \min_{|I|=n} t(I)$$

$$T_{promedio}(n) = \sum_{|I|=n} P(I) \cdot t(I)$$

donde $|I|$ refiere al tamaño de la instancia I.

Intuitivamente, $T_{peor}(n)$ es el tiempo de ejecución del algoritmo sobre la instancia que implica mayor tiempo de ejecución entre las entradas de tamaño n, mientras que $T_{mejor}(n)$ nos habla del tiempo de ejecución del algoritmo sobre la instancia que implica menor tiempo de ejecución. Por último, el análisis del caso promedio es una medida muy utilizada, y nos habla del tiempo de ejecución esperable sobre instancias "típicas" (ponderado por la probabilidad de ocurrencia de cada instancia).

Comportamiento Asintótico

En general, cuando hacemos un análisis de la complejidad de un algoritmo, nos va a interesar poder determinar su **comportamiento asintótico**. Esta idea se basa en el **principio de invarianza**[1], que nos dice que dado un algoritmo y dos implementaciones M_1 y M_2 que tienen un tiempo de ejecución $T_1(n)$ y $T_2(n)$, siendo n el tamaño de la entrada, existen $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que:

$$T_1(n) \leq c \cdot T_2(n), \qquad \forall n \geq n_0$$

Por lo tanto, no nos va a interesar tanto conocer exactamente la cantidad de operaciones de un algoritmo, sino más bien el **orden de magnitud** del tiempo de ejecución. La idea, entonces, va a ser buscar aquella función (logarítmica, lineal, cuadrática, exponencial, etc.) que exprese el comportamiento del algoritmo, para entradas suficientemente grandes. Para ello, se han propuesto distintas medidas del comportamiento asintótico: $\mathcal{O}$ (cota superior), Ω (cota inferior) y Θ (orden exacto de la función).

Esta notación se dice *asintótica* porque trabaja sobre el comportamiento de funciones en el límite, es decir, para valores suficientemente grandes de sus parámetros. En consecuencia, los argumentos basados en la notación asintótica podrían fallar en cuanto a su valor práctico cuando trabajamos con entradas con valores del "mundo real". En cualquier caso, suele ser de utilidad al momento de comparar algoritmos, y nos facilita algunas cuentas. Notemos que la utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad. Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño, o cuando las constantes involucradas son demasiado grandes, en cuyo caso mantendremos el coeficiente del término de mayor peso (pensar en si conviene usar un algoritmo cúbico en segundos o un algoritmo cuadrático en días).

La notación $\mathcal{O}$ sirve para representar el límite o cota superior del tiempo de ejecución de un algoritmo. Es decir, la notación $f \in \mathcal{O}(g)$ expresa que la función f **no crece** más rápido que alguna función proporcional a g, y decimos que g es cota superior de f. Luego, si sabemos que un algoritmo tiene un tiempo de ejecución $T_{peor} \in \mathcal{O}(g)$, podemos asegurar que para todas las entradas de tamaño suficientemente grande, el tiempo $T_{peor}(n)$ va a ser como mucho proporcional a g. Formalmente, dada una función $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ arbitraria que va desde los números naturales a los reales no negativos, por ejemplo siendo $f(n) = n^2$, siendo n un representante del tamaño de la instancia del algoritmo y $f(n)$ la cantidad de recursos utilizados por el algoritmo para procesar esa instancia. Decimos que $f \in \mathcal{O}(g)$ si y solo si

$$\exists n_0, c > 0 \text{ tal que } n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)$$

Por conveniencia, permitimos el uso incorrecto de notación para decir que $f(n)$ está en el orden de $g(n)$ incluso si $f(n)$ es negativo o indefinido para una cantidad finita de valores de n, y solo vamos a exigir que esté bien definida cuando $n \geq n_0$. La herramienta más poderosa y versátil para probar que cierta función está en el orden de otra se conoce como la **regla del límite**, que nos dice que, dadas dos funciones arbitrarias f, g ambas $\mathbb{N} \to \mathbb{R}^{\geq 0}$:

[1] Este principio no es algo que se pueda probar, sino que se sustenta en base a la observación.

1. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$, entonces $f(n) \in \mathcal{O}(g(n))$ y $g(n) \in \mathcal{O}(f(n))$. La vuelta no vale, ya que es posible que el límite no exista.

2. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, entonces $f(n) \in \mathcal{O}(g(n))$ pero $g(n) \notin \mathcal{O}(f(n))$.

3. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$, entonces $f(n) \notin \mathcal{O}(g(n))$ pero $g(n) \in \mathcal{O}(f(n))$.

Consideremos el problema de ordenamiento. Ya sabíamos que existen algoritmos como Insertion Sort o Selection Sort que pueden ordenar un arreglo de n elementos en $O(n^2)$. Sin embargo, existen otros algoritmos más sofisticados como *Heap Sort* que tienen costo $O(n \cdot \log n)$. Está claro que $n \log n \in O(n^2)$. Por lo tanto, sería correcto decir que *heapsort* está en $O(n^2)$, o incluso $O(n^3)$. Esto se debe a que la notación O está diseñada solamente para dar cotas superiores acerca de la cantidad de recursos requeridos. Claramente, necesitamos una notación dual para dar una cota *inferior*. La notación Ω justamente nos permite representar el límite o cota inferior del tiempo de ejecución del algoritmo.

Consideremos nuevamente dos funciones $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ que vayan de los naturales a los reales no negativos. Luego, la notación $f \in \Omega(g)$ expresa que la función f está acotada inferiormente por alguna función múltiplo positivo de g para valores de n lo suficientemente grandes. Formalmente, decimos que $f \in \Omega(g)$ si y solo si

$$\exists n_0, k > 0 \text{ tal que } n \geq n_0 \Rightarrow f(n) \geq k \cdot g(n).$$

A pesar de la gran similitud entre la notación $\mathcal{O}$ y la notación Ω, hay un aspecto en el que la dualidad falla. Recordemos que normalmente vamos a querer estudiar el tiempo de ejecución en el *peor caso*. Por lo tanto, cuando decimos que $T_{peor}(n) \in \mathcal{O}(g(n))$ para el peor caso, estamos diciendo que **para toda** instancia de tamaño n, existe una constante real positiva k tal que $T_{peor}(n) \leq k \cdot g(n)$. En cambio, cuando decimos que $T_{peor}(n) \in \Omega(g(n))$, estamos diciendo que existe **al menos una** instancia de tamaño n para la cual realmente tiene costo $t(I) \geq k \cdot g(n)$ (para todo $n \geq n_0$). Por lo tanto, podrían existir infinitas instancias de tamaño n que se podrían resolver en un menor tiempo.

En general, vamos a utilizar la notación Ω para dar cotas inferiores en los tiempos de ejecución de algoritmos. Sin embargo, también es posible dar cotas inferiores a la dificultad propia de resolver cierto problema. Por ejemplo, vamos a ver que para **cualquier** algoritmo que sea capaz de ordenar n elementos, basándose en comparaciones de a pares de elementos, se tiene que pertenece a $\Omega(n \cdot \log n)$. Luego, se dice que el *problema* de ordenamiento por comparaciones tiene una complejidad en $\Omega(n \cdot \log n)$. Este resultado es mucho más fuerte que decir que cierto algoritmo particular tiene una cota inferior, e incluso nos dice cuándo un algoritmo de ordenamiento es **óptimo**.

Finalmente, nos queda la notación Θ (orden exacto), que vamos a definir como el conjunto de funciones que crecen asintóticamente de la misma forma, es decir: $\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$. Formalmente:

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

Podemos reformular las reglas de límite de la siguiente forma:

1. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$, entonces $f(n) \in \Theta(g(n))$.

2. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, entonces $f(n) \in \mathcal{O}(g(n))$ pero $f(n) \notin \Theta(g(n))$.

3. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$, entonces $f(n) \in \Omega(g(n))$ pero $f(n) \notin \Theta(g(n))$.

Es posible que cuando analicemos el costo de un algoritmo, este dependa simultáneamente de más de un solo parámetro. En estos casos, la noción de "tamaño de entrada" que estuvimos usando hasta ahora pierde un poco de sentido. Por esta razón, la notación asintótica se generaliza para funciones de varias variables. Sea $g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0}$ una función que va de los pares de números naturales a los reales

no negativos, por ejemplo $g(m,n) = m \cdot \log n$. Sea $f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0}$ otra función. Decimos que $f(m,n)$ está en el orden de $g(m,n)$, denotado por $f(m,n) \in \mathcal{O}(g(m,n)$, si $t(m,n)$ está acotada superiormente por un múltiplo positivo de $g(m,n)$ cuando tanto m como n son suficientemente grandes. Formalmente, $O(g(m,n))$ se define como

$$\{t : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists c \in \mathbb{R}^{+})(\forall^{\infty} m, n \in \mathbb{N})[t(m,n) \leq c \cdot g(m,n)]\},$$

donde $\forall^{\infty}$ quiere decir que la propiedad vale para todo natural, exceptuando por una cantidad finita de excepciones. Para el caso general de n parámetros, la definición es similar.

Fórmula de Stirling : Nos sirve para aproximar factoriales grandes, y reescribir aquellas complejidades que dependan de una suma de logaritmos. La aproximación se expresa como

$$ln(n!) \approx n \cdot ln(n) - n$$

Luego, si un algoritmo $A \in \Theta(\log n!) \Leftrightarrow A \in \Theta(n \cdot \log n)$.

2.2. Diseño jerárquico de TADs

Ahora nos toca ver cómo pasamos de la *especificación* ("el qué") al *diseño* ("el cómo") del algoritmo. Recordemos que para una misma especificación, podemos diseñar distintos algoritmos, y para saber qué algoritmo nos conviene utilizar, deberemos considerar aspectos como la eficiencia en el tiempo de ejecución y el espacio utilizado, de acuerdo con el *contexto de uso*, es decir, en qué orden llegarán los datos, cómo se los consultará, cuáles serán las operaciones más frecuentemente usadas, cuál debe ser su complejidad en el peor caso, etc.

Al diseñar un algoritmo bajo una cierta especificación, estamos pasando de un paradigma *funcional* a un paradigma *imperativo*, y queremos que ese pase resulte lo más ordenado, metódico posible. Notemos que cuanto más abstracto sea el TAD que vamos a implementar, más opciones de diseño tendremos disponibles. Tenemos como objetivo obtener un diseño jerárquico y modular, permitiendo un encapsulamiento y ocultamiento de la información. Cada uno de los niveles en la jerarquía de diseño tendrá asociado un módulo de abstracción. Es decir, habrá distintos tipos abstractos de datos que deberemos diseñar, y a cada uno de ellos le corresponderá un módulo de abstracción. Por ejemplo, consideremos al TAD Conjunto y estos dos posibles diseños:

- Un arreglo redimensionable, con una Inserción (sin repetidos y en orden) en $\mathcal{O}(n)$ y una Búsqueda en $\mathcal{O}(\log n)$.
- Una secuencia, con una Inserción en $\mathcal{O}(1)$ y una Búsqueda en $\mathcal{O}(n)$.

Podríamos preguntarnos cuál de estos dos diseños nos conviene, pero esto va a depender del *contexto de uso* en el cual vamos a utilizar esta estructura de datos (¿vamos a tener muchas más búsquedas que inserciones?).

A grandes rasgos, el método se compone de la elección del TAD a diseñar, el módulo de abstracción para el TAD elegido e ir iterando hasta finalizar ente los distintos niveles. En general, el orden en el cual se diseñan los TADs es arbitrario, pero resulta una buena práctica comenzar por los más importantes o de mayor nivel de abstracción, al ser estos quienes imponen los requerimientos de eficiencia para los TADs menos importantes. Decimos que esta modalidad de trabajo es *top-down*. La idea va a ser, por ejemplo, comenzar diseñando un modelo para representar al TAD conjuntoDeNat, en el cual utilizamos valores del TAD secuenciaDeNat, que se podrían representar con punteros.

Contamos con tipos de datos **primitivos** (`bool, nat, int, real, char, puntero`) para nuestro lenguaje de diseño, los cuales se asume que ya están definidos. También tienen la particularidad de que los parámetros de tipos primitivos son los únicos que siempre se pasan **por valor**, mientras que los tipos no primitivos se pasan por **referencia** (salvo que se diga lo contrario, en cuyo caso se pasa una copia y se paga el costo de realizar dicha copia).

Cada módulo de abstracción está compuesto por dos secciones: la definición de la *interfaz* y la definición de la *representación*. En la interfaz se describe la funcionalidad del módulo y en qué contexto

puede ser usado: complejidad temporal de los algoritmos, *aliasing* y efectos secundarios de las operaciones. Recordemos que estamos trabajando bajo el paradigma imperativo, donde los datos son tratados como entidades independientes de las funciones que los utilizan, por lo que dejamos de tener *transparencia referencial*. Además, suele pasar que tengamos distintas variables que hacen referencia a una misma estructura física, a lo que llamamos *aliasing*.

Para poder formalizar un poco esta idea de aliasing entre dos variables, vamos a definir un "meta-predicado", $alias(\phi)$, que nos va a servir para describir aquellas variables que comparten memoria en la ejecución del programa. *alias* es un metapredicado con un único parámetro ϕ que es un predicado que involucra un conjunto V de dos o más variables del programa. El significado que le damos es que las variables de V satisfacen ϕ durante la ejecución del resto del programa, al menos hasta que le asignemos otro valor a dichas variables. También puede ocurrir que la variable quede indefinida, por ejemplo, cuando el valor al que hace referencia deja de ser válido (normalmente ocurre cuando se elimina un elemento que está siendo iterado). La idea es que al modificar cualquier variable en V, el resto de las variables se deben actualizar manteniendo el invariante ϕ. Por ejemplo, si tenemos alias$(s = t)$, estamos diciendo que s y t comparten la misma memoria de forma tal que cualquier modificación sobre s afecta a t y viceversa, satisfaciendo $s = t$. En general, vamos a usar aliasing para iteradores, punteros y referencias a una misma estructura física.

Cuando estamos pasando del mundo funcional al mundo imperativo, nos van a aparecer algunas cuestiones que tenemos que resolver desde el punto de vista formal. En particular, cuando estamos definiendo la Interfaz, vamos a querer definir funciones de la siguiente forma:

PERTENECE(**in** C : `conjuntoDeNat`, **in** n : `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} n \in C\}$

Sin embargo, en esta expresión estaríamos mezclando variables del mundo imperativo con conceptos del mundo funcional (como la igualdad observacional). Para resolver este problema de manera formal, vamos a introducir un operador funcional que, dado un valor en el mundo imperativo, nos permite vincularlo con su valor correspondiente del mundo funcional. Es decir, introduciremos una notación definida como una función que va del género imperativo G_I al género teórico G_T:

$$\hat{\bullet} : G_I \rightarrow G_T$$

Luego, reescribimos a la operación *pertenece* como:

PERTENECE(**in** C : `conjuntoDeNat`, **in** n : `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{\hat{res} =_{\text{obs}} \hat{n} \in \hat{C}\}$

En la sección de *representación* se elige una forma de representación utilizando otros módulos, y se resuelven las operaciones del módulo en función de su representación. También es necesario definir cuál es la relación entre la estructura de representación y el TAD representado, para finalmente definir, explicar y mostrar cómo son los algoritmos que implementan las operaciones del módulo. Estos algoritmos pueden utilizar elementos de módulos de menor jerarquía, y se deben incluir todos los cálculos que justifican las complejidades de las operaciones.

La *estructura de representación* describe los valores sobre los cuales se representará el género que se está implementando. Esta estructura de representación tiene que ser capaz de representar todos los posibles valores del TAD y, además, debe poder describir cómo es que las operaciones del TAD se traducen en operaciones que se realizan sobre la estructura (satisfaciendo las exigencias del contexto de uso). Para ello, vamos a tener que explicar cómo se relaciona la representación con la abstracción, es decir, qué quiere decir qué. La estructura de representación de las instancias solo será accesible a través de las operaciones que se hayan detallado en la interfaz del módulo de abstracción, permitiendo separar cada nivel en la jerarquía de diseño, ocultando cómo es que realmente se representa cada tipo.

La siguiente parte de un módulo de diseño son los Algoritmos. Consideremos el siguiente problema. Se nos pide implementar un conjunto de naturales, bajo el siguiente contexto: *los números del 1 al 100 deben manejarse en* $\mathcal{O}(1)$*, mientras que el resto en* $\mathcal{O}(n)$*. Además, se debe poder conocer rápidamente*

la cardinalidad. Vamos a representar al conjunto en tres partes. Un arreglo de 100 posiciones booleanas (un 1 en la posición i indica que el elemento i pertenece al conjunto). Esto nos va a permitir manejar a los números del 1 al 100 en $\mathcal{O}(1)$. Además, vamos a incluir en nuestra representación una secuencia, donde se almacenarán todos los elementos que pertenezcan al conjunto y que sean mayores que 100. Y por último, vamos a incluir un *nat* para la cardinalidad. Esto lo vamos a notar de la siguiente manera:

`conjunto_semi_rápido` **se representa con** `estr`

donde `estr` es `tupla`(*rápido*: `arreglo [1 ... 100] de bool` , *resto*: `secu(nat)` , *cardinal*: `nat`)

Ahora bien, si nos convencimos de que ésta es una buena representación, tenemos que preguntarnos si es posible representar a cualquier valor del TAD Conjunto y, por otro lado, si cualquier valor que tome esta estructura tiene una contraparte en el TAD.

Está claro que todo conjunto puede ser representado por esta estructura, sin embargo, no todos los valores corresponden a instancias válidas. Por ejemplo, la tupla $([0, \ldots, 0], \langle 37, 107, 28\rangle, 3)$ no representa ninguna instancia del TAD. Por lo tanto, es necesario tener algún predicado que nos permita distinguir entre representaciones válidas de las representaciones inválidas, y así poder establecer una relación entre la representación y el valor representado del TAD. Este predicado se conoce como **invariante de representación**.

Este invariante de representación lo vamos a agregar a las precondiciones, y lo vamos a tener que cumplir en las postcondiciones de las operaciones del módulo (no así en las operaciones auxiliares). Esto nos va a obligar, pero también permitir, mantener ciertas garantías sobre las estructuras de representación, lo que nos puede permitir hacer las cosas más eficientemente (podemos usar algoritmos que aprovechen las garantías del invariante).

Volviendo al ejemplo, las condiciones del invariante de representación, descriptas de manera informal, deberían ser las siguientes:

1. Que *resto* solo tenga números mayores que 100.
2. Que *resto* no tenga números repetidos.
3. Que *cardinal* tenga la longitud de *resto* más la cantidad de celdas de *rápido* que estén en 1 (*true*).

Veamos ahora la descripción formal del invariante. Formalmente, el invariante de representación es una función booleana que tiene como dominio la versión abstracta del género de representación (el $\hat{\bullet}$), e indica si la instancia es o no válida. Luego, el invariante de representación, descripto formalmente, nos queda:

$$\begin{aligned} &Rep : \hat{estr} \to bool \\ &(\forall e : \hat{estr})Rep(e) \equiv true \Leftrightarrow \\ &\qquad (1)(\forall n : nat)(\text{esta?}(n, e.resto) \ \Rightarrow \ n > 100) \wedge \\ &\qquad (2)(\forall n : nat)(\text{cant\_apariciones}(n, e.resto) \leq 1) \wedge \\ &\qquad (3)e.cant = long(e.resto) + \text{contar\_trues}(e.rapido) \end{aligned}$$

donde *esta?*, *cant_de_apariciones* y *long* son funciones de secuencias, y *contar_trues* es una función sobre arreglos. En los problemas más complicados se vuelve muy importante analizar metódicamente la estructura de representación a la hora de elegir los predicados del invariante. Una forma de organizarse es empezar mirando cada campo de la estructura de forma individual, para ver bien qué condiciones tienen que cumplir. Una vez analizado esto, podemos continuar analizando la relación que deben tener los distintos campos entre sí, mirando de a pares, para que su información sea consistente.

Con esto podemos restringir el conjunto de valores posibles que podría tomar la estructura de representación. Sin embargo, todavía nos falta poder vincular las distintas instancias válidas de la estructura con el valor representado del TAD. Para ello, utilizamos la **función de abstracción**. Esta función de abstracción toma una instancia abstracta ($\hat{\bullet}$) de la estructura de representación y nos devuelve una

instancia abstracta del género representado. La función de abstracción debe ser un homomorfismo con respecto a la signatura del TAD, es decir, para toda operación $\bullet$ del módulo, se tiene que cumplir que:

$$Abs(\bullet(p_1, \ldots, p_n)) =_{\text{obs}} \hat{\bullet}(Abs(p_1), \ldots, Abs(p_n)$$

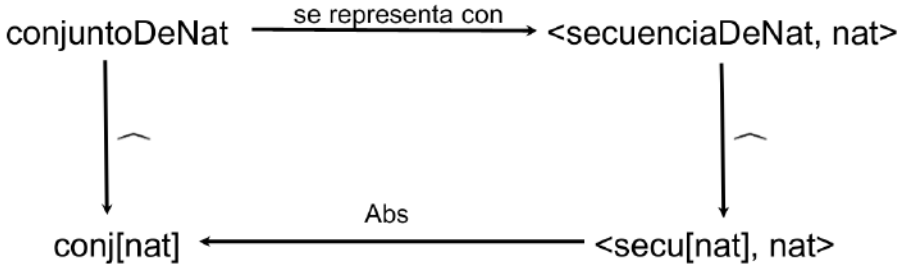

Figura 2.1: Función de Abstracción

Normalmente, tenemos dos formas de describir la función de abstracción. La primera de ellas es en función de sus observadores básicos, dado que estos permiten identificar de manera unívoca a cualquier instancia. Otra forma de describirla es en función de los generadores del TAD. Veamos cómo podemos escribir la función de abstracción del módulo de conjunto_semi_rápido:

Abs: $\hat{\text{estr}}$ e $\rightarrow$ conjunto_semi_rápido {Rep(e)}

$$\begin{aligned} Abs(e) =_{\text{obs}} C \,/\, & \\ & (\forall n : nat)(n \in C \Leftrightarrow ((n \geq 100 \wedge r.rapido[n]) \vee \\ & (n > 100 \wedge \text{está?}(n, e.resto))) \end{aligned}$$

La función de abstracción debe ser total, restringiéndose a las instancias que cumplan con el invariante de representación. Además, no tiene por qué ser inyectiva, es decir, dos estructuras diferentes pueden representar al mismo término de un TAD, pero tiene que ser sobreyectiva sobre las clase de equivalencia del TAD (dadas por la igualdad observacional). Por último, nos falta diseñar los algoritmos que implementan las distintas operaciones del módulo. Junto con los algoritmos, incluiremos la precondición y la postcondición. Veamos cómo notamos el algoritmo para la operación AGREGAR(...):

AGREGAR(**in/out** C : conjunto_semi_rápido, **in** e : nat)
Pre $\equiv \{\hat{C} =_{\text{obs}} C_0 \quad \wedge \quad \hat{e} \notin C\}$
Post $\equiv \{\hat{C} =_{\text{obs}} Ag(C_0, \hat{e})\}$

Luego, una posible implementación del algoritmo podría ser la siguiente:

```
1 iAgregar(in/out C : estr, in e : nat)
2 C.cant++
3 if e < 100 then
4 |  C.rápido[e] = true
5 else
6 |  AgregarAtrás(C.resto, e)
```

Cuando estamos en la etapa de implementación, tenemos la flexibilidad de ponerle un nombre especial para el algoritmo. En este caso, lo llamamos I, de implementación, pero podríamos ponerle algún otro nombre. Otra cosa a tener en cuenta es que estamos trabajando directamente sobre la estructura de representación (y no la abstracción).

Capítulo 3

Estructuras de Datos

En los capítulos anteriores estuvimos hablando del camino que hay que recorrer entre la descripción informal hasta llegar a una solución concreta, y vimos algunas herramientas formales para describir y realizar ese proceso. En esta parte nos vamos a dedicar a estudiar soluciones concretas a problemas concretos, que nos van a permitir avanzar en la implementación de problemas fundamentales de la informática.

3.1. Conjuntos y Diccionarios

Vamos a empezar con los TADs Conjuntos y Diccionarios, que constituyen uno de los núcleos de la algoritmia: el problema de organizar y buscar la información en un sistema informático. En los conjuntos, lo único que nos interesa es saber si un elemento particular pertenece o no al conjunto, mientras que en los diccionarios, además de saber si una *clave* está definida para ese diccionario, nos interesa poder encontrar su *significado*.

Es fácil ver que si tenemos una implementación de un diccionario, podemos implementar un conjunto simplemente ignorando el significado de las claves. Por otro lado, también podemos implementar un diccionario a partir de un conjunto, por ejemplo, guardando a las claves junto con sus significados en una tupla ⟨clave, significado⟩. En caso de tener como significado a una estructura compleja, podemos guardarnos un puntero a la misma, en lugar de la estructura en sí. Otra opción para el manejo de claves con significados podría ser mantener dos estructuras paralelas. La idea es que en la posición i de una tengamos la clave, y en la misma posición i de la otra tengamos su significado. Como podemos representar diccionarios a partir de conjuntos, y conjuntos a partir de diccionarios, podemos decir que ambos problemas son equivalentes. A continuación, vamos a trabajar sobre la representación de este tipo de estructuras, buscando representaciones e implementaciones eficientes de las mismas.

La primer idea que se nos ocurre para representar a los conjuntos es a través de estructuras secuenciales: en cada posición del arreglo, almacenamos la tupla ⟨ clave, significado ⟩, y un puntero para saber dónde termina el arreglo.

Ahora, pensemos cuáles van a ser los distintos órdenes de complejidad de las operaciones básicas:

- Vacío: para crear un diccionario vacío, nos basta con crear un arreglo vacío y definir el puntero al último como NIL, por lo que la operación nos cuesta $O(1)$.
- Definir: para definir un nuevo elemento, tenemos que agregar este nuevo elemento al final del arreglo y actualizar el puntero al último elemento, por lo que la operación nos cuesta $O(1)$.
- Definido?: para saber si una clave está o no definida, tenemos que realizar una búsqueda secuencial por todo el arreglo, por lo que la operación nos lleva $O(n)$.
- Significado: para obtener el significado de una clave, nuevamente tenemos que realizar una

búsqueda secuencial, con una costo $O(n)$.

Otra posibilidad es mantener el arreglo ordenado, permitiendo hacer uso de la búsqueda binaria, mejorando la complejidad de DEFINIDO? y OBTENER hasta $O(\log n)$. El problema de mantener el arreglo ordenado es que al momento de DEFINIR una nueva clave, debemos buscar la posición que le toca en el arreglo, y correr todos los elementos siguientes, con un costo de $O(n)$. Por lo tanto, este segundo modelo sería mejor en contextos de uso estáticos, en el que se realicen muchas búsquedas y pocas inserciones.

Cualquiera de estas dos opciones tienen operaciones con complejidad en $O(n)$, por lo que nos gustaría tener alguna estructura más eficiente, de manera tal que todas las operaciones se puedan realizar con un costo sub-lineal. Para ello, vamos a representar conjuntos a través de árboles binarios. Un grafo $G = (V, E)$ es un conjunto de nodos (V) unidos por un conjunto de lineas, llamadas *ejes* (E). Decimos que un grafo es conexo cuando podemos ir de cualquier nodo a cualquier otro nodo siguiendo una secuencia de ejes. Las secuencias de ejes pueden formar caminos (si no visitamos dos veces un mismo vértice) y ciclos (si visitamos un vétice más de una vez). Un *árbol* es un grafo conexo y acíclico. Si cuenta con un nodo distinguido, llamado *raíz*, decimos que el árbol es *enraizado*.

Normalmente, cuando dibujamos un árbol enraizado, ponemos la raíz en la cima, similar a un árbol familiar, con los otros ejes saliendo hacia abajo de la raíz. Podemos describir las relaciones entre los nodos haciendo una analogía del árbol familiar, usando términos como *padre*, *hijos*, hermanos, ancestros, etc. Una *hoja* en un árbol es un nodo sin hijos; los otros nodos se denominan nodos *internos*; también existen los nodos *externos*, que vendrían ser nodos imaginarios que no pertenecen al árbol (la idea es que si un nodo no tiene algún hijo, su eje apunta a un nodo externo). Un árbol k-ario es aquel en el que cada nodo puede tener a lo sumo k hijos. En esta sección, vamos a trabajar específicamente sobre árboles binarios, donde tenemos un nodo inicial, el nodo raíz, del cual pueden salir a lo sumo dos nodos (hijo izquierdo e hijo derecho), y por cada nodo que no sea el raíz, tenemos un nodo padre y podemos tener a lo sumo dos hijos.

Pensando un poco sobre el costo de las operaciones de un diccionario, podemos ver que las complejidades serían las siguientes:

- VACÍO: para crear un diccionario vacío, nos basta con crear un árbol vacío, el cual tiene como raíz a un NIL, por lo que la operación nos cuesta $O(1)$.
- DEFINIR: para definir un nuevo elemento, tenemos que definir una regla para agregarlo al árbol. Podemos tomar como regla que se agrega en el primer lugar posible en el que entra el nodo, sin tener que inaugurar un nuevo nivel. Esto lo podemos hacer manteniendo un puntero al último, por lo que la operación nos cuesta $O(1)$.
- DEFINIDO?: para saber si una clave está o no definida, tenemos que realizar una búsqueda secuencial por todo el árbol, por lo que la operación nos lleva $O(n)$.
- SIGNIFICADO: para obtener el significado de una clave, nuevamente tenemos que realizar una búsqueda secuencial, con una costo $O(n)$.

Podemos ver que no hemos mejorado en cuanto a la complejidad de las operaciones. Sin embargo, lo que podemos hacer es restringir un poco la forma en la que está organizado el árbol binario, manteniendo un invariante sobre la estructura que nos permita mejorar el costo de las operaciones.

3.2. Árbol Binario de Búsqueda (ABB)

Un **Árbol Binario de Búsqueda** (ABB) es un árbol binario que satisface que para todo nodo, los valores de los elementos en su subárbol izquierdo son menores que el valor de ese nodo, y los valores de los elementos de su subárbol derecho son mayores. Luego, podemos definir el invariante de un árbol ABB de la siguiente forma:

1. el valor de todos los elementos del subárbol izquierdo es menor que el valor de la raíz,
2. el valor de todos los elementos del subárbol derecho es mayor que el valor de la raíz,
3. el subárbol izquierdo es un ABB,

4. el subárbol derecho es un ABB.

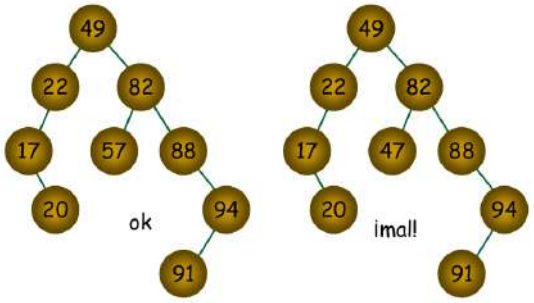

Figura 3.1: Ejemplos de árboles binarios. (47 no puede pertenecer al subárbol derecho de 49.)

Formalmente, tomando como estructura de representación para los árboles binario *ab(nodo)* la tupla $\langle I, r, D\rangle$, donde I es el subárbol izquierdo, r la raíz y D el subárbol derecho, podemos plantear el siguiente invariante de representación para los árboles ABB:

$$
\begin{aligned}
&ABB : ab(nodo) \to bool \\
&(\forall e : ab(nodo))ABB(e) \equiv true \Leftrightarrow \\
&\qquad (0)ABB(e) \equiv \text{Nil?}(e) \vee \\
&\qquad (1)(\forall c : clave)(\text{esta?}(c, Izq(e)) \;\Rightarrow\; c < clave(raiz(e))) \wedge \\
&\qquad (2)(\forall c : clave)(\text{esta?}(c, Der(e)) \;\Rightarrow\; c > clave(raiz(e))) \wedge \\
&\qquad (3)ABB(Izq(e)) \wedge \\
&\qquad (4)ABB(Der(e))
\end{aligned}
$$

Notemos que para poder construir un ABB, es necesario poder dar un orden total a las claves, es decir, que podamos determinar, dadas dos claves cualesquiera c_1, c_2, si $c_1 \leq c_2$. Ahora, veamos algunos algoritmos básicos para los ABB:

iVacio(**in** $\langle\,\rangle$, **out** A : ab(nodo)

```
1 return NIL
```

iDefinir(**in** c : clave, s : significado, **in/out** A : ab(nodo)

```
1 if A = NIL then
2     A ← ab(NIL, ⟨c, s⟩, NIL)
3 else
      // A = ab(I, ⟨r_c, r_s⟩, D)
4     if c < r_c then
5         A ← ab(definir(c, s, I), ⟨r_c, r_s⟩, D)
6     else
7         A ← ab(I, ⟨r_c, r_s⟩, definir(c, s, D))
8 return A
```

Podemos ver que el costo de inserción va a depender de la cantidad de recursiones que tenemos que hacer para llegar desde el nodo raíz hasta la hoja en donde vamos a colocar el nuevo nodo, es decir, depende de la longitud de la rama. Sin embargo, si vamos insertando de forma descuidada los distintos elementos, el ABB resultante podría quedar *desbalanceado*. Con esto nos referimos a que muchos nodos en el árbol tengan un solo hijo, de manera tal que las ramas se vuelvan largas y finas. Cuando esto ocurre, las operaciones en el árbol dejan de ser eficientes, ya que en el peor caso, todos los nodos en el árbol podrían tener exactamente un hijo (exceptuando la única hoja). Por lo tanto, para la búsqueda de un elemento en el árbol nos quedaría en $O(n)$ comparaciones, al tener que comparar con todos los nodos

del árbol. Por otro lado, si las claves que nos van llegando siguen una distribución uniforme, es esperable que nos vaya quedando un árbol casi balanceado (casi completo), por lo que nos quedaría un árbol de altura $\approx \log n$, y la búsqueda nos quedaría $O(\log n)$ en el caso promedio.

Para el algoritmo de borrado, Borrar(u : nodo, A : ab(nodo)) tenemos tres casos:

1. u es una hoja. En este caso, lo único que tenemos que hacer es buscar al padre de u en A, y luego eliminar la hoja u (reemplazándolo por NIL). Si u no tiene padre, significa que es la raíz, y como es hoja, estamos en el caso de un árbol de un único nodo, por lo que al quitarlo, nos quedaríamos con el árbol vacío.

2. u tiene un solo hijo. En este caso, primero debemos buscar al padre w de u. Su u no tiene padre, significa que u es la raíz, por lo que debemos reemplazar a la raíz de A por la raíz del único subárbol de r_A. Su u tiene padre, tenemos dos casos: o bien u es hijo izquierdo o bien es hijo derecho. Si es hijo izquierdo, significa que u es menor a w, al igual que todos los hijos de u. Entonces, podemos mover a todo el subárbol de u (no importa si es subárbol derecho o izquierdo), y colocarlo donde antes estaba u, ya que sabemos que todos ellos son menores a w. El caso de que u sea hijo derecho es análogo al anterior.

3. u tiene dos hijos. En este caso, lo que vamos a hacer es buscar un reemplazo de u dentro de alguno de los subárboles de u, el cual vamos a mover desde su posición original a la posición de u. Para ello, debemos encontrar algún nodo que cumpla con el invariante que debe cumplir u, es decir, debe ser mayor a todos los nodos del subárbol izquierdo de u y debe ser menor a todos los nodos del subárbol derecho de u. Para cumplir esta condición, tenemos dos posibles candidatos: el nodo de más a la derecha dentro del subárbol izquierdo y el nodo de más a la izquierda dentro del subárbol derecho. Vamos a considerar el primer caso (el otro es similar).

   El nodo que está más a la derecha dentro del subárbol izquierdo es mayor a todos los nodos del subárbol izquierdo, además, es menor que u, por lo que también es menor a todos los nodos del subárbol derecho. Por lo tanto, queda claro que lo podríamos mover hacia la posición de u. Para moverlo, lo que vamos a hacer es copiarlo a la posición de u, y luego borrarlo dentro del subárbol izquierdo al que pertenecía. Para ello, tenemos que darnos cuenta que al ser el nodo más a la derecha, no puede tener un hijo derecho, porque sino no sería el de más a la derecha. Luego, estamos en el caso 2, en el que tenemos que borrar un nodo con un solo hijo, que sabíamos resolver.

Una última propiedad de los árboles ABB es que podemos recorrerlo en tiempo lineal, de forma tal de que podamos imprimir a las claves ordenadas de menor a mayor, a través de un algoritmo recursivo llamado **inorder tree walk**. Se llama así porque primero imprime todas las claves del subárbol izquierdo, luego la clave de la raíz y finalmente todas las claves del árbol derecho (recursivamente). Similarmente, tenemos el **preorder tree walk** que imprime la raíz antes que cualquier subárbol, y el **postorder tree walk** que imprime la raíz después de ambos subárboles (en los tres casos se imprime primero el subárbol izquierdo y antes del subárbol derecho).

Inorder-tree-walk(x)

```
1 if x ≠ NIL then
2 |   Inorder-tree-walk(izq)
3 |   Imprimir x
4 |   Inorder-tree-walk(der)
```

Para darnos cuenta de que el algoritmo es lineal en función de la cantidad de elementos, pensar que solo visitamos a cada nodo una única vez, y que el costo de cada visita es constante. Esta forma de recorrer el árbol nos está dando implícitamente una forma de ordenar n elementos en tiempo $\Theta(n \log n)$, lo cual nos estaría diciendo de que esta estructura es óptima (en el caso promedio), en cuanto al costo asintótico de sus operaciones, ya que sabemos que el problema de ordenamiento basado en comparaciones de claves completas es $\Omega(n \log n)$.

3.3. Árboles AVL

Queríamos obtener una estructura que nos permita mejorar la complejidad de las operaciones de diccionario para el peor caso, y con los árboles ABB seguimos teniendo el mismo problema de antes, ya que seguimos teniendo algunas operaciones que tienen costo lineal. Uno de los conceptos claves que deberíamos tener en cuenta, a partir del desarrollo de los ABBs, es que el costo de las operaciones depende de la profundidad del árbol, por lo que si pudiéramos controlar esta profundidad, podríamos mejorar la complejidad de las operaciones. Luego, nuestro objetivo es encontrar una estructura similar a los ABBs que nos asegure que el árbol se mantenga **balanceado**. La idea es que si logramos que las ramas del árbol sean lo más parecidas entre sí o, dicho de otro modo, que el árbol sea lo más completo posible, tendríamos que la rama más larga, que define el caso peor, sería lo más corta posible.

Existen varios métodos para mantener un árbol balanceado, garantizando la eficiencia en las operaciones de búsqueda, inserción y borrado. Entre las técnicas más viejas tenemos el uso de árboles AVL y árboles 2-3; otras estructuras más modernas incluyen a los Red-Black Trees y los Splay Trees. En esta sección vamos a estudiar las propiedades de los árboles AVL para ver cómo consiguen mantener balanceado el árbol.

Cuando hablamos de un árbol **perfectamente balanceado**, a lo que nos referimos es a que todas las ramas tengan casi la misma longitud (dos ramas difieren en a lo sumo una unidad), y que todos los nodos internos tengan ambos hijos. Se puede demostrar que un árbol perfectamente balanceado de n nodos tiene altura $\lfloor log_2(n) \rfloor + 1$. El problema de mantener un árbol perfectamente balanceado es que no se conocen algoritmos eficientes que permitan mantener el invariante de los ABB a través de sucesiones de inserciones y borrados, por lo que tendríamos una degradación en la performance (más adelante vamos a ver otra estructura conocida como *heap*, cuyo invariante es lo suficientemente débil como para que sea sencillo mantenerla perfectamente balanceada).

Lo siguiente que falta hacer es buscar una propiedad que sea más débil que el balanceo perfecto, pero que nos permita tener operaciones eficientes. Se dice que un árbol es *balanceado en altura* si las alturas de los subárboles izquierdo y derecho *de cada nodo* difieren en a lo sumo una unidad. Antes pedíamos que la altura de *todas* las ramas difieran en a lo sumo una unidad, mientras que ahora lo que estamos pidiendo es que la *altura* del subárbol derecho y el izquierdo difieran en a lo sumo una unidad, es decir, que la rama más larga de uno y la rama más larga del otro difieran en longitud a lo sumo una unidad. De esta manera, permitimos tener ramas que varíen en más de una unidad en su longitud, por lo que queda claro que la propiedad de balanceo en altura es más débil que la propiedad de balanceo perfecto. Con esta idea en mente, definimos el *factor de balanceo* (**FDB**) de un nodo u como

$$FDB = h(D_u) - h(I_u),$$

siendo D_u el subárbol derecho de u e I_u el subárbol izquierdo, y decimos que un árbol es balanceado en altura si para todo nodo u vale que $-1 \leq FDM(u) \leq 1$. Vamos a asumir que como parte de la estructura de los árboles AVL tenemos precalculado el factor de balanceo para cada nodo.

Lo que queremos ver es que un árbol AVL tiene una altura logarítmica en función de la cantidad de nodos. Para demostrar esto, lo que se hace es ver que para toda altura h, cualquier árbol AVL tiene una cantidad exponencial de nodos. Para ello, nos alcanza con ver que aquel árbol AVL de altura h de menor cantidad de nodos, tiene una cantidad exponencial de nodos en función de la altura. Este tipo de árboles AVL, que tienen el mínimo número de nodos para una altura h, se conocen como **árboles de Fibonacci**, porque se puede probar que la cantidad de nodos de un árbol de Fibonacci $Fibo_h = Fibo_{h-1} + Fibo_{h-2} + 1$, y que $Fibo_h = F_{h+2} - 1$, siendo F_i el i-ésimo término de la sucesión de Fibonacci. Para ver esto, lo que tenemos que saber es que se puede construir un árbol de Fibonacci de altura $i + 2$ a partir de unir dos árboles de Fibonacci: uno de altura i con otro de altura $i + 1$.

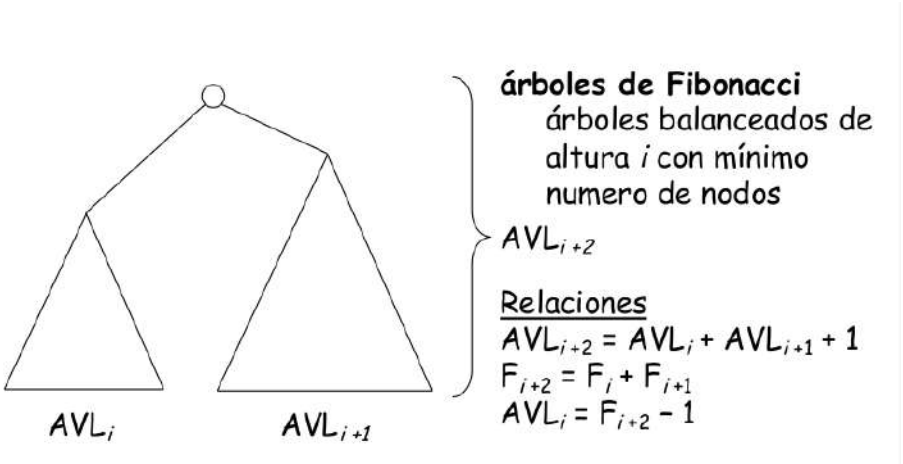

Figura 3.2: Árboles de Fibonacci

A partir de esto, se puede probar que un árbol de Fibonacci con n nodos tiene una altura menor a $1{,}44\log(n+2) - 0{,}328$, por lo que quedaría demostrado que un AVL de n nodos tiene altura $\Theta(\log n)$, al ser un árbol de Fibonacci lo "más desbalanceado" que puede llegar a ser un árbol AVL. Con todo esto en mente, nos falta ver si es posible realizar las operaciones de diccionarios de forma eficiente, manteniendo esta estructura de los AVLs.

Inserción : Básicamente, la inserción en un árbol AVL consta de tres pasos: primero tenemos que insertar al elemento como lo haríamos en un ABB ($\Theta(\log n)$), recalculamos los FDBs partiendo desde abajo hacia arriba, y luego tenemos que realizar una serie de **rotaciones** que nos permitan recuperar el invariante del AVL, es decir, que $-1 \leq FDB(u) \leq 1$ para todo nodo u.

Para recalcular los FDB, tenemos que darnos cuenta que solo pueden verse alterados aquellos FDB asociados a los nodos que pertenecen a la rama que va desde la raíz hasta el nodo agregado, por lo que a lo sumo vamos a tener que recorrer $\Theta(\log n)$ nodos, ya que sabemos que la altura de los árboles AVL es logarítmica con respecto a la cantidad de nodos.

Por último, nos queda ver cuántas rotaciones tenemos que hacer para que el árbol vuelva a ser un AVL, y cuánto nos cuesta hacer cada una. Tenemos dos tipos de rotaciones sobre un nodo: rotación a izquierda y rotación a derecha.

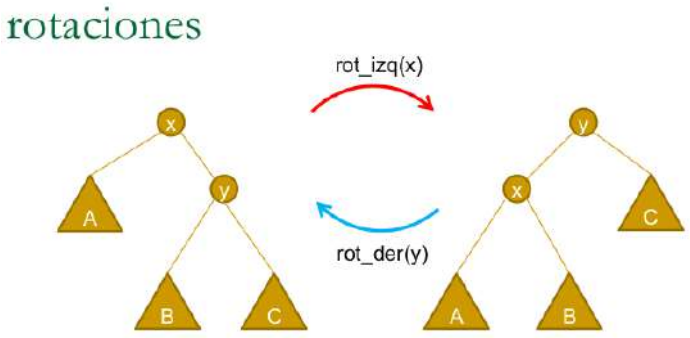

Figura 3.3: Tipos de Rotaciones

Notemos que al momento de aplicar cualquier tipo de rotación, ya sea a izquierda o a derecha, se mantiene el invariante de los árboles ABB, es decir, se cumple que para todo nodo en el árbol, todos los nodos de su subárbol derecho son mayores y todos los nodos de su subárbol izquierdo son menores a este

nodo. Además, podemos ver que una rotación nos cuesta $O(1)$, porque lo único que hacemos es cambiar una cantidad constante de punteros de lugar.

Ahora, lo que nos falta ver es cómo rebalanceamos el árbol que había sido AVL, pero que le agregamos un nuevo elemento. En el peor caso, a lo sumo vamos a tener que hay nodos que tengan $FDB(u) = \pm 2$, con u perteneciente a la rama que va desde la raíz hasta el nodo que queríamos agregar. Para analizar todos los casos posibles, vamos a separar en cuatro casos:

- RR: queremos insertar en el subárbol derecho de un hijo derecho Q. En este caso, nos basta con hacer una rotación a izquierda sobre el padre de Q.
- LR: queremos insertar en el subárbol izquierdo de raíz R de un hijo derecho Q, con padre P. En este caso, tenemos que hacer una rotación doble: primero hacemos una rotación a derecha sobre Q, y por último hacemos una rotación a izquierda sobre P.
- RL: queremos insertar en el subárbol derecho de un hijo izquierdo. Similar al caso LR.
- LL: queremos insertar en el subárbol izquierdo de un hijo izquierdo. Similar al caso RR.

Con todo esto en cuenta, nos queda que el costo del primer y segundo paso es proporcional a la altura del árbol ($\Theta(\log n)$), y el costo del último paso, que consiste en hacer a lo sumo dos rotaciones, nos queda en $O(1)$. Por lo tanto, concluimos que el costo total del algoritmo de inserción en los árboles AVL es $\Theta(\log n)$.

Borrado : El algoritmo de borrado consiste en tres pasos: borrar el nodo como lo haríamos en un ABB, recalculamos los factores de balanceo, y por último tenemos que hacer una rotación simple o doble por cada nodo con FDB = ± 2 ($O(\log n)$ en el peor caso). Por lo tanto, la operación de Borrado también cuesta $\Theta(\log n)$.

3.4. Radix Searching

Existen varios métodos de búsqueda que proceden examinando a las claves de a partes, en lugar de hacer comparaciones entre claves completas en cada paso, conocidos como *radix searching methods*. La motivación de estas estructuras es encontrar una implementación de diccionarios que termine dependiendo del *tamaño* de las claves, y no tanto de la *cantidad* de claves. Para ello, nos vamos a enfocar en realizar comparaciones de *partes* de las claves, en lugar de comparar claves completas. Por ejemplo, si las claves son enteros, sus partes podrían ser los bits que lo codifican; si las claves son strings, las partes podrían ser sus caracteres.

Algunas de las principales ventajas de los métodos de radix searching son: proveer una eficiencia razonable en el peor caso sin la complicación de los árboles balanceados; proveer una forma fácil de manejar claves de longitud variable; ahorrar espacio guardando parte de la clave dentro de la estructura de búsqueda; y proveer un acceso muy rápido a los datos. Uno de los problemas de hacer radix searching es que no podemos lidiar fácilmente con claves iguales (se podría usar una lista de punteros a los distintos registros). En general, asumimos que todas las claves con las que vamos a trabajar son distintas.

A continuación, vamos a examinar varios métodos de radix searching, donde cada uno corrige un problema inherente del anterior, llegando a un método importante que resulta bastante útil en aplicaciones de búsqueda donde tenemos claves muy largas.

3.4.1. Árboles de búsqueda digital

El método de radix searching más simple son los árboles de búsqueda digital. Los Árboles de Búsqueda Digital son parecidos a los ABB en cuanto a que para guardar una clave, vamos recorriendo el camino en el árbol que nos lleva a su posición. La diferencia está en que la posición no está determinada por comparaciones de los valores completo de la clave, sino que por comparaciones sobre los bits (dígitos o caracteres) de la clave. En la Fig. 3.4 podemos ver un ejemplo de un Árbol de Búsqueda Digital. Supongamos que queremos insertar el elemento X en el árbol. Para ello, en cada paso nos movemos a

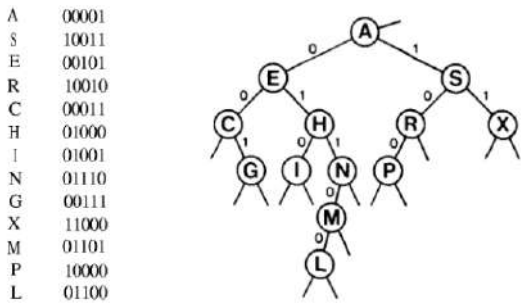

Figura 3.4: Árboles de búsqueda digital.

derecha o a izquierda según si la i-ésima posición de la clave tenemos un 1 o un 0. Una vez alcanzamos una hoja, simplemente se coloca a X como hijo izquierdo o derecho de dicha hoja, según corresponda.

Podemos ver que la altura del árbol está acotada por la longitud máxima de las claves. Mirando un poco más en detalle, podemos notar que la altura del árbol de búsqueda digital es exactamente el mayor número de bits sucesivos iguales entre dos claves cualesquiera, es decir, la longitud del mayor prefijo común entre dos claves cualesquiera. Si tenemos n claves de longitud b, en el caso promedio, podemos ver que a lo sumo se van a necesitar $O(\log n)$ comparaciones de claves completas, mientras que en el peor caso vamos a tener $O(b)$ comparaciones de claves. Necesitamos hacer comparaciones de claves completas porque estamos almacenando el valor de la clave dentro de cada nodo, y a partir de su posición solo podemos deducir que la clave tiene cierto prefijo y no podemos deducir cuál es su valor completo. Notemos que si tenemos muchas claves de poca longitud, entonces $b \ll n$, por lo que resultan una alternativa atractiva frente a los ABB tradicionales.

3.4.2. Tries

Las claves de búsqueda suelen ser muy largas, posiblemente de más de 20 caracteres. En estas situaciones, el costo de comparar una clave por igualdad puede terminar siendo dominante en el costo de la búsqueda, por lo que no puede ser ignorado. En los árboles digitales de búsqueda todavía teníamos que comparar claves en cada nodo. En esta sección, vamos a ver una estructura de datos que nos permita implementar diccionarios comparando un solo dígito en cada nodo, en lugar de comparar claves completas.

Para evitar tener que guardar en cada nodo una clave, lo que vamos a hacer es poner todas las claves en las hojas. Por lo tanto, vamos a tener dos tipos de nodos: los nodos internos, que simplemente contienen los enlaces a otros nodos, y lo nodos hoja, que contienen las claves y no tienen enlaces. Esta estructura se conoce como *trie*, derivado de la palabra *retrieval*. Para buscar una clave en el árbol, vamos a movernos según los bits de la clave, pero no vamos a compararla con ninguna clave hasta llegar a una hoja. Cada clave en el árbol se almacena en una hoja que pertenece al camino descripto por el patrón de dígitos de la clave, y completamos la búsqueda comparando la clave deseada con la clave almacenada en la hoja. Es una estructura apropiada cuando tenemos muchas palabras que empiezan con la misma secuencia de caracteres, es decir, cuando la cantidad de prefijos distintos entre todas las palabras en el conjunto es mucho menor que la longitud total de todas las palabras.

En general, un trie es un árbol k-ario para alfabetos de $k-1$ elementos posibles. En principio, un trie admite claves de cualquier longitud, por lo que es necesario tener un dígito adicional que nos permita saber que la clave terminó ahí ("$"). Si las claves fueran de una longitud prefijada, no sería necesario tener este dígito adicional, ya que la terminación de una clave vendría dada por el hecho de encontrarnos en el nivel k. Luego, el trie pasaría a ser un árbol k-ario para alfabetos de k elementos. También es posible tener distintos tipos de nodos con distinta aridad, por ejemplo, podríamos tener nodos con una alta aridad en los primeros niveles y nodos con baja aridad en los últimos (este método se dice *híbrido*).

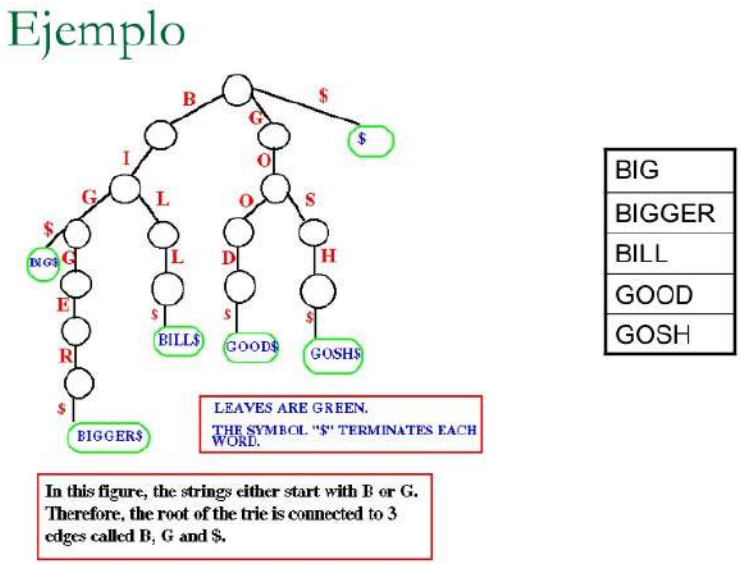

Figura 3.5: Ejemplo de Trie.

Para un trie construido a partir de n claves de b bits, se puede probar que se necesitan $\Theta(\log n)$ comparaciones **de bits** en el caso promedio, asumiendo una distribución uniforme de la codificación de los elementos, es decir, dada una posición i, es igualmente probable encontrar un 1 que un 0. En el peor caso, nos queda en $O(b)$ comparaciones de bits. También vale la pena mencionar que la estructura del trie es independiente del orden en el que se insertan las claves, es decir, hay un **único** trie para cada conjunto de claves (cosa que no pasaba ni en los ABB ni en los ABD).

El problema que tenemos en los Tries clásicos es la ramificación que surge al momento de insertar dos claves que tienen un prefijo muy largo en común. Por ejemplo, las claves que difieren en tan solo el último bit requieren de un camino cuya longitud sea igual a la longitud de la clave, sin importar la cantidad de claves en el árbol. Por lo tanto, la cantidad de nodos internos puede ser mayor que la cantidad de claves. Vamos a ver más adelante cómo podemos corregir este problema.

Implementación de Tries: Básicamente vamos a tener los dos formatos para representar a los nodos: arreglos (memoria estática) y listas (memoria dinámica). La primera posibilidad consiste en tener en cada nodo interno un arreglo de punteros de longitud k.

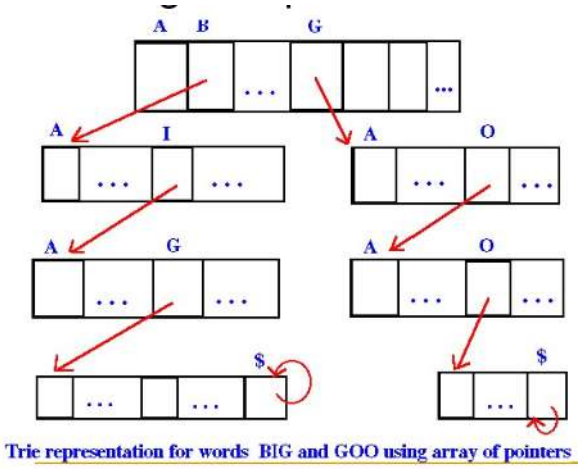

Figura 3.6: Implementación de Trie con Arreglos

El problema que tiene este esquema es posible que gran parte de las entradas no se utilicen, por lo

que se estaría desperdiciando mucha memoria, especialmente cuando tenemos un alfabeto grande y un diccionario chico. La ventaja que tiene es que la eficiencia en términos de tiempo es extrema, al tener un acceso rápido a cada posición del arreglo.

La segunda posibilidad es tener una representación basada en listas, donde en cada nodo se almacenan todas las posibles continuaciones a través de una lista enlazada. La idea es que cada nodo tenga un puntero a la lista de sus hijos, y un segundo puntero a su siguiente hermano (formando una lista de hermanos).

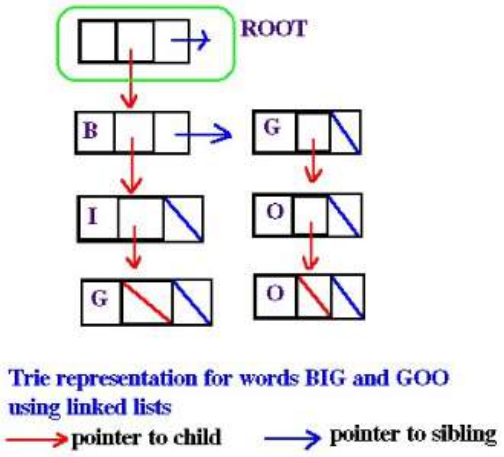

Figura 3.7: Implementación de Trie con Listas

Esta solución es eficiente en términos de tiempo solo si hay pocas claves (la lista de hermanos es corta), pero es mucho más eficiente en términos de memoria.

Tries Compactos: Para terminar, vamos a ver algunas variantes de los Tries que son utilizadas en la práctica. La primera optimización son los **Tries Compactos**, en los que colapsamos las cadenas que llevan a una única hoja. Es decir, si en algún momento llegamos a tener una serie de nodos en los que cada nodo tiene un único hijo, es innecesario representarlo con la cadena completa, ya que podemos directamente conectar al primer nodo de la cadena con el nodo hoja.

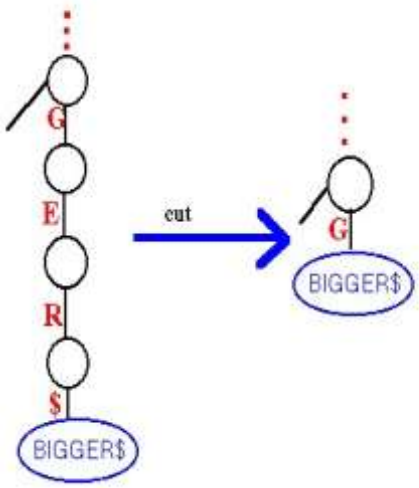

Figura 3.8: Tries Compactos

Patricia: Aún seguimos teniendo dos problemas a resolver: cuando tenemos dos claves con un prefijo en común muy largo, se nos forma una rama muy larga en la que todos los nodos tienen un único hijo, lo que lleva a la creación de nodos adicionales en el árbol, y el segundo problema que tenemos es que hay dos tipos distintos de nodos en el árbol, lo cual complica la implementación de los algoritmos, agregando un cierto overhead. D. R. Morrison descubrió una forma de evitar ambos problemas, método al que llamó **Patricia** ("Practical Algorithm To Retrieve Information Coded In Alphanumeric"). Patricia nos permite la búsqueda de n claves de longitud arbitraria en un árbol de tan solo n nodo, requiriendo una única comparación de clave completa por búsqueda. (Vamos a explicar el caso de un trie Patricia binario).

La ramificación "unidireccional" se evita de la siguiente forma. Cada nodo contiene el índice del bit que debe ser testeado para decidir qué camino tomar como salida. En los tries tradicionales, en el i-ésimo nivel mirábamos el i-ésimo dígito menos significativo. En cambio, en Patricia miramos el índice indicado por el nodo, respetando el hecho de que a medida de que bajamos en el árbol, miramos dígitos cada vez más significativos. El efecto es el mismo que si compactáramos los caminos intermedios que no tienen bifurcaciones en un único eje.

Los nodos externos[1] son evitados reemplazándolos por enlaces que apuntan hacia arriba en el árbol. Además, volvemos al esquema de los árboles digitales de búsqueda, donde teníamos las claves en los nodos. Sin embargo, en Patricia, las claves en los nodos no se utilizan en el camino de bajada durante la búsqueda; simplemente se almacenan allí para ser referenciados al momento de llegar a un nodo externo. Podemos reconocer cuándo un enlace lleva a un nodo más arriba en el árbol comparando el índice del nodo actual con el índice del nodo referenciado, ya que sabemos que los índices de bit va decrementando a medida que bajamos por el árbol (se considera que un nodo que se apunta a sí mismo, está subiendo en el árbol).

Para realizar una búsqueda en este árbol, comenzamos en la raíz y procedemos a bajar por el árbol, usando el índice que contiene cada nodo, que nos dice cuál bit examinar en la clave de búsqueda, yendo para la derecha si el bit es 1 e izquierda si el bit es 0. Las claves almacenadas en los nodos no se revisan en el camino de bajada, simplemente se examina el bit indicado en cada nodo. Eventualmente, un enlace que nos lleva hacia arriba en el árbol es encontrado: cada uno de estos enlaces apuntan a la única clave en el árbol cuyos bits se correspondan con el camino tomado. Luego, si la clave apuntada es igual a la clave buscada, la búsqueda fue exitosa; sino, no está el elemento en el árbol.

En conclusión, Patricia es un método de radix searching eficiente que consigue identificar los bits que distinguen las claves de búsqueda y los utiliza para formar una estructura de datos (sin nodos sobrantes) que permite encontrar rápidamente cualquier clave en la estructura. Para conjuntos de n claves razonables, el árbol debería tener una altura proporcional a $\log n$.

Por último, vamos a comparar a los *tries* con los árboles AVL. Sabemos que en cualquier método de búsqueda, la longitud de las claves está incorporada de alguna manera en el algoritmo, por lo que el tiempo de ejecución es dependiente de la longitud de las claves. Sin embargo, al momento de analizar el costo de las operaciones de los AVL, no habíamos considerado la posibilidad de tener claves lo suficientemente grandes como para que las comparaciones entre claves no se puedan asumir constantes. Bajo el modelo logarítmico, si tenemos claves de b bits (variable), las operaciones del AVL dejan de ser $O(\log n)$, y pasan a ser $O(b \log n)$, ya que en cada paso realiza una comparación de claves de b bits. En cambio, las operaciones del trie siguen siendo $O(b)$, ya que todas las operaciones involucradas son comparaciones sobre bits [2].

3.5. Hashing

Muchas aplicaciones requieren de un conjunto dinámico que dé soporte a las operaciones de diccionario de inserción, búsqueda y borrado. Por ejemplo, un compilador que traduce un lenguaje de programación mantiene una tabla de símbolos, en la cual las claves son cadenas de caracteres arbitrarias correspondientes a identificadores en el lenguaje. Una tabla hash es una estructura de datos efectiva para implementar diccionarios. A pesar de que la búsqueda de un elemento en una tabla hash puede tomar tiempo $\Theta(n)$ en el peor caso, en la práctica resultan extremadamente eficientes. Bajo asunciones

[1] Recordar que un nodo externo es un nodo imaginario que no pertenece al árbol, pero que usamos para no tener que distinguir entre nodos con distitna cantidad de hijos.

razonables, el tiempo promedio de búsqueda de un elemento en una tabla hash es $O(1)$. Otro uso de las tablas hash es cuando trabajamos con el procesamiento de datos almacenados en memoria secundaria (ver *Hashing Extensible*).

Los arreglos trabajan con **direccionamiento directo**. Supongamos que tenemos que almacenar 10000 perfiles distintos, identificables a partir de una clave; en este caso, el número de DNI. El DNI es un número que va entre 0 y $10^8 - 1$, por lo que si quisiéramos utilizar un diccionario implementado a partir de un arreglo, tendríamos que tener 10^8 entradas del mismo. La ventaja de utilizar un arreglo es que podemos acceder a los datos del perfil en $O(1)$, pero estamos desperdiciando mucha memoria, ya que solo tenemos que almacenar 10000 perfiles distintos. En general, utilizar arreglos para representar conjuntos resulta eficiente cuando la cantidad de claves almacenadas es cercana a la cantidad total de claves posibles. Cuando esto no se cumple, las tablas hash resultan una alternativa eficiente, ya que éstas utilizan un espacio proporcional a la cantidad de claves utilizadas. La idea es que en lugar de utilizar la clave como índice, el índice sea *computado* a partir de la clave. En esta sección vamos a ver cómo se calculan los índices, las distintas formas de direccionamiento y cómo se manejan las *colisiones* en las tablas hash.

Para indexar con otro tipos de datos, escapando de la necesidad de utilizar enteros, lo que vamos a hacer es encontrar una función de correspondencia que mapee a cada dato posible con un entero. La resolución de este problema se conoce como **Pre-hashing**. Una primera alternativa (teórica) consiste en aprovechar el hecho de que los datos en una computadora se representan como una tira de bits, la cual podemos interpretar como un entero. En la práctica, se cuenta con una función de pre-hasheo, que depende de la implementación de cada lenguaje de programación, la cual nos devuelve un entero a partir de la clave. Este tipo de funciones van a respetar (idealmente) que pre-hash$(x) =$ pre-hash$(y) \iff x = y$. Una posible función de pre-hashing para string puede ser asociar a cada caracter su código ASCII, y al string asignarle el número obtenido en base 2.

Cuando trabajamos con una tabla hash, vamos a representar un diccionario a partir de una tupla $\langle T, h \rangle$, donde T es un arreglo con m celdas, y $h : Claves \to \{0, \ldots, n-1\}$ es la función de hashing, que nos da una correspondencia entre el conjunto de claves posibles al conjunto de las posiciones de la tabla (estos índices son conocidos como *pseudoclaves*). De esta manera, la posición de un elemento en el arreglo se calcula a través de la función h.

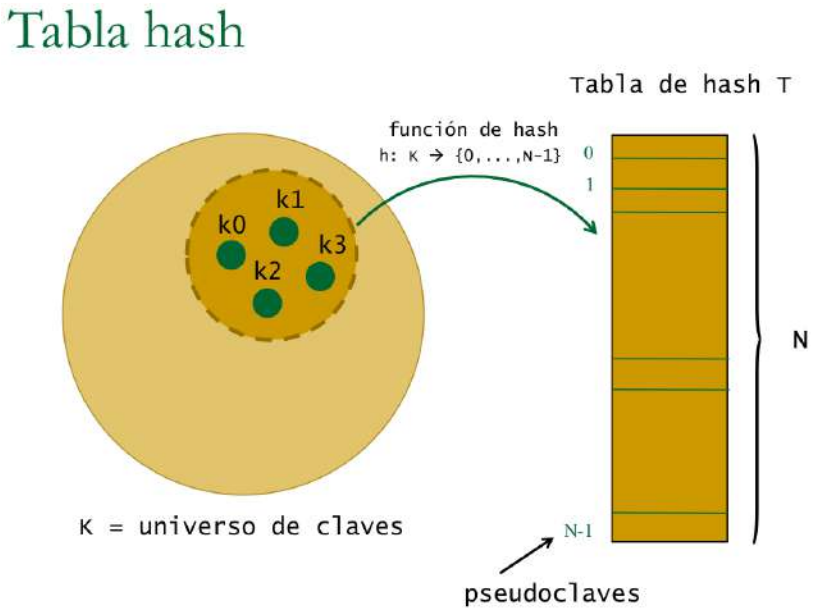

Figura 3.9: Tabla Hash

Existen mucho métodos para definir funciones hash:

- División: $h(k) = k \mod m$. En general, se eligen un tamaño m para la tabla hash tal que sea un

número primo no demasiado cercano a una potencia de 2.

- Partición: Se particiona la clave k en muchas claves $k_1, k_2, \ldots, k_n$, y se toma como función de hash $h(k) = f(k_1, k_2, \ldots, k_n)$.
- Extracción: Se usa solo una parte de la clave para calcular la posición, por ejemplo, si tenemos una clave de 12 cifras, se consideran solo las 6 centrales.
- Mid-square, etc.

Siempre se tiene como objetivo romper de alguna manera cualquier estructura o patrón previo que traigan consigo las claves, distribuyendo de forma uniforme las distintas claves.

Una función hash se dice **perfecta** si para todo par de claves distintas, las pseudoclaves asociadas son distintas. El problema que tenemos es que para poder asegurar que una función hash sea perfecta para cualquier conjunto de claves, se requiere que el tamaño de la tabla hash sea mayor o igual al cardinal del universo de claves, que es lo que queríamos evitar. Por lo tanto, en la práctica lo que se hace es tener una función hash que no sea perfecta, donde el tamaño de la tabla sea mucho menor a la cantidad de claves posibles. Como no podemos tener una función de hashing perfecta, vamos a tener que definir otra propiedad que nos diga cuándo una función de hashing es buena.

Decimos que una buena función de hashing es aquella que satisface (aproximadamente) la propiedad de *uniformidad simple*: distribuye las claves de forma uniforme e independiente entre valores de 0 y $m-1$. Esto quiere decir que para cualquier clave k elegida al azar de un conjunto de claves K,

$$P(h(k) = i) \approx 1/m \qquad \forall 0 \leq i \leq m-1.$$

El problema que tenemos es que típicamente no conocemos la distribución probabilística de las claves, por lo que no tenemos manera de comprobar si esta propiedad se cumple. Tampoco podemos randomizar $h(k)$ para que h sea una función no determinística, ya que de hacerlo, no podríamos buscar de forma eficiente a k en la tabla (podría estar en cualquier entrada). En la práctica, se utilizan heurísticas para crear funciones de hash que funcionan bien. Para el análisis de complejidad de las operaciones de las tablas hash, se asume la existencia de funciones que cumplan la uniformidad simple, a pesar de que obtener este tipo de funciones en la práctica sea una tarea elusiva. Notemos que el costo de calcular $h(k)$ debe ser bajo para cualquier clave $k \in K$ para que nos sea de utilidad.

En resumen, vamos a tener que lidiar con escenarios **frecuentes** en los que dos claves distintas van a ser mapeadas al mismo lugar, originando una **colisión** (pensar en la paradoja del cumpleaños). Por lo tanto, se necesitan de métodos para la resolución de colisiones, los cuales se diferencian por la forma en la que ubican a los elementos involucrados en una colisión. Vamos a estudiar dos familias de métodos para la resolución de colisiones: los métodos por concatenación (también llamados de direccionamiento cerrado o de Hashing Abierto) y los métodos de direccionamiento abierto (también llamados de Hashing Cerrado).

3.5.1. Concatenación

En el método de resolución de colisiones por concatenación, colocamos a todos los elementos que hashean a la misma posición de la tabla en una misma lista enlazada. Es decir, en la i-ésima posición de la tabla se coloca la cabeza de la lista que almacena a todos los elementos tales que $h(k) = i$; en caso de que no hayan elementos, se coloca un *NIL*. Es tentador intentar mejorar este esquema reemplazando estas listas de colisión por árboles balanceados, pero esto no vale la pena si el *factor de carga* se mantiene pequeño, a menos que sea esencial mejorar el costo asintótico en el peor caso.

la inserción de un elemento k consiste en colocar al principio (o al final) de la lista asociada a la posición $h(k)$, por lo que la operación de inserción nos queda en $O(1)$. La operación de búsqueda consiste en recorrer linealmente la lista asociada a la posición $h(k)$, por lo que el costo de esta operación depende de su longitud. Para la operación de borrado, nuevamente, recorremos secuencialmente la lista asociada a $h(k)$, y borramos el elemento deseado, por lo que el costo nos queda lineal en función del tamaño de la lista.

Para analizar la longitud de las listas, definimos el **factor de carga** $\alpha = n/m$, para una tabla T con

m posiciones que almacenan n elementos, siendo α la longitud promedio de las listas. En el peor caso, todas las claves terminan hasheadas en la misma posición, creando una lista de tamaño n, por lo que en el peor caso ambas operaciones nos quedan en $\Theta(n)$. Suponiendo uniformidad simple en la función y que se puede computar la función de hash en tiempo constante, se puede probar que en el caso promedio tenemos un costo $\Theta(1+\alpha)$. Si tenemos que la cantidad de posiciones en T es al menos proporcional a la cantidad de elementos, nos queda que $\alpha = n/m \approx 1$, por lo que las operaciones nos quedan en $\Theta(1)$ en el caso promedio.

3.5.2. Direccionamiento abierto

En el método de resolución de colisiones por direccionamiento abierto, todos los elementos se guardan en la tabla. Es decir, cada entrada en la tabla contiene un elemento o *NIL*. Cuando hacemos una búsqueda, empezamos desde una posición inicial en la tabla, determinada por la clave, y vamos barriendo sobre la tabla *computando* la secuencia de entradas a visitar, hasta que encontremos al elemento o hasta que tengamos la certeza de que el elemento no está presente en la tabla. Como todos los elementos se almacenan en la tabla, es posible que la tabla se llene, por lo que debemos tener algún mecanismo de redimensionamiento. La ventaja que presentan los métodos de direccionamiento abierto es que no se requieren de punteros adicionales, por lo que obtenemos un mejor aprovechamiento del espacio, permitiendo una mayor cantidad de entradas por la misma cantidad de memoria, potencialmente reduciendo la cantidad de colisiones (mejorando la eficiencia de las búsquedas).

El algoritmo de búsqueda consiste en ir visitando las distintas posiciones dadas por la función de hash, hasta encontrar alguna en la que esté el elemento. Para ello, vamos barriendo sobre la tabla, probando distintas posiciones. Como en cada intento debemos cambiar de posición a examinar, la función de hash va a depender de la cantidad de posiciones examinadas que se hicieron hasta el momento. Luego, se utilizan funciones de hash $h(k,i)$ que nos definan la posición para la clave k en el i-ésimo intento de búsqueda. Cuando la casilla está ocupada por un elemento distinto al buscado, lo que hacemos es aumentar i, y probar devuelta con $h(k,i+1)$ para que, eventualmente, podamos encontrar al elemento o a un espacio vacío (indicando que el elemento no está en la tabla). Entonces, buscamos funciones de hash tales que para cada clave k, la secuencia de barrido

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$$

sea una permutación de $\langle 0,1,\ldots,m-1\rangle$. La operación es similar, con la única diferencia de que seguimos barriendo hasta encontrar una entrada vacía, y decimos que cuando intentamos insertar la clave en una posición, pero la posición está ocupada, tenemos una colisión.

Notemos que estamos asumiendo que las claves nunca son borradas de la tabla. Cuando permitimos el borrado de una clave de la posición i, no podemos simplemente marcarla como *NIL* en su lugar. Si lo hiciéramos, es posible que seamos incapaces de recuperar cualquier clave que había pasado por esa posición y la había encontrado ocupada. Podemos resolver este problema marcando esa entrada como BORRADA en lugar de *NIL*. Sin embargo, cuando usamos el valor de BORRADA, el costo de búsqueda (y de inserción) deja de depender del factor de carga, y es por este motivo que el método de concatenación suele ser más utilizado cuando se necesita la operación de borrado.

El concepto de *uniformidad* generaliza la idea de uniformidad simple para este tipo de funciones, y nos dice que una función de hash cumple con la propiedad de uniformidad si para cualquier clave k elegida al azar, cualquiera de las $m!$ permutaciones de $\langle 0,1,\ldots,m-1\rangle$ tiene la misma probabilidad de ser su secuencia de barrido. Implementar funciones de hash que realmente satisfagan la propiedad de uniformidad es demasiado complicado, sin embargo, en la práctica se utilizan aproximaciones como el *hashing doble*.

Vamos a diferenciar los métodos de direccionamiento abierto a partir de la **técnica de barrido** que utilicen, es decir, según cómo la función $h(k,i)$ va recorriendo todas las posiciones de la tabla ante una colisión. Las tres más típicas son el Barrido Lineal, el Barrido Cuadrático y el Hashing doble. Hacemos esta diferenciación porque cambia la complejidad de cálculo y su comportamiento respecto a los fenómenos de **aglomeración** (cómo se distribuyen las clave cuando ocurren colisiones).

Barrido Lineal: En este caso, la función de hash $h(k,i)$ va a ser igual a una función de hashing $(h'(k)+i) \mod m$, para $i = 0,1,\ldots,m-1$. Dada una clave k, primero probamos con $h'(k)$, la casilla

dada por la función de hash auxiliar. Luego probamos con $h'(k)+1$, y así siguiendo hasta $m-1$. Después, damos la vuelta a la tabla y seguimos con $0, 1, \ldots, h'(k)-1$. Como $h'(k)$ determina completamente la secuencia de barrido, solo tenemos m secuencias distintas.

La técnica de barrido lineal es fácil de implementar, pero tiene el problema de **aglomeración primaria**. Se empiezan a armar segmentos largos de casillas todas ocupadas, incrementando el tiempo promedio de búsqueda. Estas aglomeraciones aparecen porque una casilla vacía precedida por i entradas llenas tiene probabilidad $(i+1)/m$ de llenarse, porque la secuencia de barrido de cualquier clave que caiga en cualquiera de las i entradas va a terminar en esta casilla. Esto lleva a que cada vez sea más probable caer en estos segmentos, extendiéndolos cada vez más.

Por ejemplo, en la Fig. 3.10 podemos ver que tenemos la secuencia de inserciones: 2, 103, 104, 105, ... Primero se inserta el 2, como es el primer intento $i = 0$, por lo que $h(2,0) = (h'(2)+0) \mod 101$, siendo $h'(k) = k \mod 101$, el 2 va a la posición 2. Luego, si inserta el 103, como es el primer intento $i = 0$, por lo que $h(103,0) = 103 \mod 101 = 2$, pero la posición 2 está ocupada. Entonces, se vuelve a intentar insertar con $i = 1$, y esta vez se coloca en la posición 3, que estaba libre.

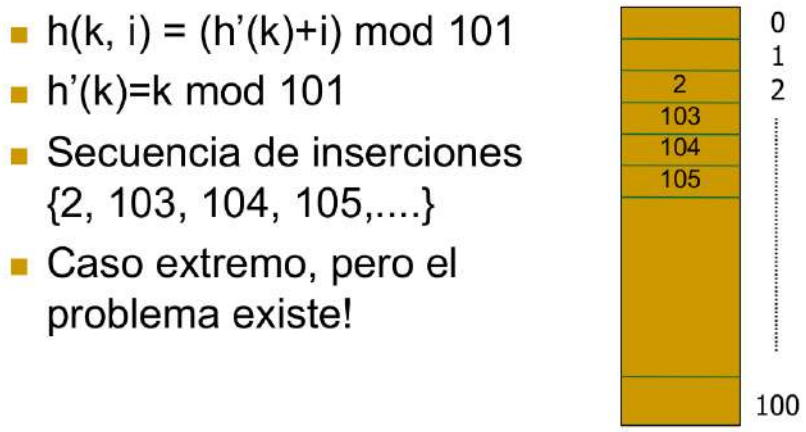

Figura 3.10: Ejemplo de Aglomeración Primaria.

Barrido Cuadrático: En este caso, la función de hash es de la forma

$$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m,$$

donde c_1 y c_2 son constantes. La posición inicial es $h'(k)$, y las posiciones siguientes son desplazamientos que dependen de forma cuadrática de la cantidad de intentos. De esta manera, se evitan los grandes segmentos de la aglomeración primaria. El problema que seguimos teniendo es que para cualquier par de claves k_1, k_2 tales que $h'(k_1) = h'(k_2)$, toda su secuencia de barrido es exactamente la misma. Esta propiedad tiene un efecto más suave de aglomeración, conocido como **aglomeración secundaria**. Además, al igual que en la técnica de barrido lineal, $h'(k)$ determina la secuencia de barrido completa, por lo que solo hay m secuencias distintas.

Hashing Doble: Ofrece uno de los mejores métodos disponibles para direccionamiento abierto. La idea es que el barrido también depende de la clave, utilizando una segunda función de hashing:

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m.$$

Por lo tanto, a diferencia de los casos de barrido lineal y de barrido cuadrático, la secuencia de barrido depende de dos formas independientes de la clave, ya que tanto la posición inicial como el desplazamiento varían en base a ésta. Para que la tabla pueda ser recorrida completamente, el valor de $h_2(k)$ debe ser coprimo con el tamaño de la tabla m . Una manera conveniente de asegurar esto es que m sea una potencia de 2 y que h_2 sea diseñada para que siempre devuelva un número impar. Cuando esto se cumple, la técnica de hashing doble termina generando $\Theta(m^2)$ secuencias de prueba, en lugar de las $\Theta(m)$ de las técnicas

anteriores, ya que cada par $\langle h_1(k), h_2(k) \rangle$ define una secuencia distinta, acercándonos al ideal de $m!$ secuencias de barrido. En conclusión, la técnica de hashing doble no tiene aglomeración primaria, pero sí puede tener aglomeración secundaria, siendo menos probable que en los casos anteriores.

3.6. Colas de Prioridad

La Cola de Prioridad es un TAD parecido al TAD Cola, pero con la diferencia de que el orden en el que salen los elementos no está dado por el orden de llegada, sino que está dado por un atributo de los elementos, denominado **prioridad** (típicamente, la clave). Se permite el agregado de elementos en cualquier orden, y tenemos una operación de Desencolar elemento, que nos devuelve el elemento encolado de mayor prioridad. Si tenemos empates, de alguna manera tenemos que resolverlo (por ejemplo, agregando un segundo orden de prioridad del tipo FIFO). Las colas de prioridad dinámicas pueden modelar una lista de tareas a ser ejecutadas por un sistema operativo.

Para implementar este TAD, vamos a utilizar una estructura adecuada cuyas operaciones tienen la misma complejidad asintótica que los árboles AVL, estructura conocida como **Heap**. Algunas de las ventajas de los heaps con respecto a los AVL es que es una estructura mucho más simple de implementar. Además, es posible que al ser una estructura más sencilla, el factor constante que acompaña a las operaciones de los heaps va a ser mucho más chico que el que acompaña a las operaciones de los AVL (pensar que en los AVL tenemos que recalcular los factores de balanceo y aplicar rotaciones, en lugar de intercambios). Esto toma particular importancia cuando trabajamos con operaciones que toman tiempo logarítmico, porque si tenemos que dos algoritmos A, B que tienen costo $O(\log n)$, pero el factor constante de una es el doble de la otra $T_A(n) = 2T_B(n)$, entonces el tiempo que tarda A en procesar una entrada de tamaño n es similar al tiempo que tarda B en procesar una entrada de tamaño n^2, dado que $2 \log n = \log n^2$.

Un *heap* es un árbol enraizado *perfectamente balanceado* (e izquierdista) que puede ser implementado eficientemente en un arreglo sin punteros explícitos. Cada nodo del árbol se corresponde con un elemento en el arreglo. Esta estructura tiene numerosas aplicaciones, incluyendo la técnica de ordenamiento de *heapsort*. Es una estructura de datos ideal para encontrar el máximo (o el mínimo) elemento de un conjunto, removerlo, agregar un nuevo nodo o modificarlo. La idea es tener a los elementos parcialmente ordenados, permitiendo extraer el elemento más chico (o el más grande), de manera tal de que sea fácil mantener el invariante de representación de la estructura. Este invariante se conoce como **condición de heap**, y consiste en las siguientes propiedades:

1. Un heap es un Árbol Binario perfectamente balanceado.
2. La prioridad de cada nodo es mayor que la de sus hijos.
3. Todo subárbol de un heap es a su vez un heap.
4. Es izquierdista, es decir, el último nivel está lleno desde la izquierda o, dicho de otro modo, todos los espacios libres en el último nivel tienen que estar a la derecha de los elementos de ese último nivel.

Para implementar los heaps, nos sirven todas las representaciones usadas para árboles binarios: representación con punteros, donde contamos con nodos que tienen punteros explícitos a sus hijos, y la representación con arreglos, que es una representación implícita porque la posición en la que se encuentra cada elemento establece la relación que tiene con el resto de los nodos. Nos vamos a centrar en esta última, al ser la representación más práctica para implementar heaps.

Para la representación con arreglos, lo que vamos a hacer es almacenar cada nodo v en la posición $pos(v)$, donde $pos(v)$ es una función recursiva que definimos de la siguiente manera:

- Si v es la raíz, entonces $pos(v) = 0$.
- Si v es hijo izquierdo de u, entonces $pos(v) = 2pos(u) + 1$.
- Si v es hijo derecho de u, entonces $pos(v) = 2pos(u) + 2$.

Esto nos permite navegar fácilmente a través del heap, sin la utilización de punteros explícitos, ya que no solo podemos acceder desde un nodo padre a cualquiera de sus hijos, sino que además podemos determinar la posición del padre de un nodo, a partir de la siguiente cuenta: $pos(padre(u)) = \lfloor (i-1)/2 \rfloor$, siendo i la posición del hijo u en el arreglo. Además, al evitar la utilización de punteros explícitos, obtenemos un mejor uso de la memoria (y un uso más eficiente de la cache al tener una mejor vecindad espacial entre los elementos[1]).

La desventaja que tenemos es propia del uso de memoria estática, en la que nos vemos obligados muchas veces a redimensionar el arreglo a medida que se van agregando/eliminando elementos (lo cual podría ser un problema en aplicaciones real-time). Notemos que el motivo por el que podemos utilizar arreglos para representar heaps de forma eficiente sin desperdiciar memoria es que estamos trabajando con **árboles completos**, a diferencia de los árboles ABB y los árboles AVL en donde no podíamos garantizar esta propiedad.

Podemos diferenciar a los heaps en dos tipos: los *max-heaps* y los *min-heaps*. Cuando tenemos que a mayor clave, mayor propiedad, tenemos un *max-heap*; mientras que cuando a menor clave, mayor prioridad, tenemos un *min-heap*. Notemos que en la raíz de un heap, vamos a tener al elemento de mayor prioridad: en el caso de los min-heaps, será el de clave mínima, mientras que en el caso de los max-heaps, será el de clave máxima. Veamos ahora cómo nos queda el costo para las operaciones básicas definidas en el TAD Cola De Prioridad cuando utilizamos heaps:

- Vacía: Crea un heap vacío. En cualquiera de las implementaciones, tenemos costo $O(1)$.
- Próximo: Devuelve el elemento de máxima prioridad, sin modificar el heap. Consiste en mirar la raíz del heap, por lo que tenemos costo $O(1)$.
- Encolar: Agrega un nuevo elemento, teniendo que restablecer el invariante.
- Desencolar: Elimina el elemento de máxima prioridad, teniendo que restablecer el invariante.

Encolar: La operación de *encolar* consiste en dos pasos. Primero, insertamos el elemento a agregar en la primera posición disponible. Como sabemos que un heap es izquierdista, podemos saber fácilmente la posición en la que el nuevo elemento debe ser insertado (simplemente recordamos cuál fue el último lugar en el arreglo ocupado). El siguiente paso consiste en recuperar el invariante del heap: la prioridad de cada nodo debe ser mayor a la de sus hijos. En este caso, insertamos a un elemento en el último nivel sin tener en cuenta su prioridad, por lo que debemos *subirlo* hasta el nivel que corresponda. Para hacer esto, comparamos la prioridad del nuevo nodo con la de su padre, intercambiándolos en caso de que el nuevo nodo tenga mayor prioridad, y repetimos este proceso hasta encontrar algún nodo padre de mayor prioridad. En el peor caso, el nodo terminaría subiendo todos los niveles hasta volverse la nueva raíz del heap, y como la altura de un árbol perfectamente balanceado es $\log n$, a lo sumo vamos a necesitar de $O(\log n)$ intercambios y comparaciones.

Desencolar: La operación de *desencolar* consiste en extraer el elemento de mayor prioridad del heap, es decir, eliminar la raíz. Nuevamente, consta de dos pasos. Primero, pisamos la clave almacenada en la raíz con la clave almacenada en la última hoja (la de más a la derecha), y luego eliminamos la última hoja. Esto lo hacemos para mantener el invariante del heap, que nos dice que debe ser un árbol perfectamente balanceado e izquierdista. Aún nos falta recuperar parte del invariante, ya que en la raíz nos quedó un elemento con baja prioridad. Para corregir esto, lo que vamos a hacer es *bajar* este nodo que nos quedó en la raíz hasta un lugar en el que se cumpla el invariante. Es decir, mientras este nodo p tenga al menos un hijo con mayor prioridad, lo que vamos a hacer es intercambiarlo con su hijo de mayor prioridad, siendo el peor caso que baje hasta volverse una hoja. Por lo tanto, a lo sumo vamos a necesitar de $O(\log n)$ intercambios y comparaciones.

Por último, vamos a mencionar un algoritmo que nos permite construir un heap en base a un arreglo cualquiera, en tiempo lineal, conocido como *heapify* (también llamado algoritmo de Floyd). Es utilizado para la implementación de heapsort, mejorando el factor constante del algoritmo.

Algoritmo de Floyd : Lo que queremos hacer es, dado un **arreglo** cualquiera, transformarlo en un heap (implementado como arreglo). La idea del algoritmo de Floyd es transformando localmente, de abajo

hacia arriba (estrategia *bottom-up*), cada uno de los subárboles de la estructura hasta llegar a la raíz. La idea es empezar con los subárboles más chicos que contengan hojas[2], e ir aplicando la operación de *bajar* para convertirlos en heaps. Luego, una vez tenemos todos estos subárboles de altura k transformados en heaps, lo que hacemos es considerar los subárboles de altura $k+1$, y se aprovecha el hecho de que todos los subárboles de altura k son heaps, por lo que a lo sumo se necesita de *bajar* un nodo por cada subárbol de altura $k+1$ (el que había quedado como raíz en los heaps de altura $k+1$). Se puede probar que el costo total nos queda en $O(n)$.

[2]Notemos que no hace falta hacer nada en el caso de considerar el subárbol que contenga solo una hoja, porque este ya cumple el invariante de heap.

Capítulo 4

Sorting

El problema de ordenamiento es uno de los problemas más clásicos y útiles de la informática. Vamos a dividir este tema en dos partes: ordenamiento interno y ordenamiento externo. El ordenamiento interno se realiza en la memoria principal de una computadora. El ordenamiento externo es necesario cuando el número de objetos a ser ordenados es demasiado grande como para entrar en la memoria principal, y será abarcado en un capítulo posterior.

Los algoritmos más simples de ordenamiento suelen tomar costo $O(n^2)$ para ordenar n objetos y son útiles solo para listas cortas (insertion sort, selection sort, bubblesort, shellsort, etc.). Uno de los algoritmos más populares de ordenamiento es *quicksort*, que toma tiempo $O(n \log n)$ en el caso promedio. Quicksort funciona bien en las aplicaciones comunes, sin embargo, en el peor caso tiene costo $O(n^2)$. Existen otros métodos, como *heapsort* y *mergesort*, que toman tiempo $O(n \log n)$ en el peor caso. Por otro lado, mergesort es un algoritmo apropiado para ordenamiento externo (ver Sort-Merge). También vamos a considerar otros algoritmos llamados "bin" o "bucket" sort. Estos algoritmos solo funcionan en tipos especiales de datos, como los enteros que están limitados en cierto rango, pero cuando son aplicables, son extremadamente eficientes, teniendo costo $O(n)$ en el peor caso.

Vamos a asumir que los objetos a ser ordenados son registros de uno o más campos. Uno de los campos, llamado la *clave*, es el tipo que tiene una relación de orden lineal $\leq$ definida. Enteros, reales y arreglos de caracteres son ejemplos típicos de claves. El problema de ordenamiento consiste en colocar una secuencia de registros de manera tal que los valores de sus claves se encuentren en forma creciente (es posible que haya claves repetidas). Existen varios criterios para evaluar el tiempo de ejecución de un algoritmo de ordenamiento interno. Los más comunes son medir la cantidad de pasos que se requieren para ordenar n registros, la cantidad de comparaciones entre pares de claves (importante si las claves son largas) y, si los registros son muy grandes, es de interés conocer la cantidad de veces en las que se debe mover a un registro.

Decimos que un algoritmo de ordenamiento es **estable** si dos registros i, j con claves iguales mantienen su orden relativo una vez ordenado el arreglo. Por ejemplo, en la Fig. 4.1 tenemos un arreglo con cierto orden, en el que aparece primero i y luego j. Después de ordenar con un algoritmo *estable*, lo que se asegura es que i va a terminar antes que j en el arreglo. En cambio, después de ordenar con un algoritmo *inestable*, no tenemos esta garantía, sino que podría pasar que j termine antes que i, o también podría pasar que i termine antes que j en el arreglo. Es importante saber cuándo un algoritmo de ordenamiento es estable, ya que esto permite implementar otros algoritmos que requieren de la utilización de algoritmos de ordenamiento estables como procedimientos auxiliares (como bin sort o radix sort).

4.1. Heap Sort

La idea es básicamente aplicar Selection Sort, es decir, seleccionar el mínimo elemento del sub-arreglo que ocupa las posiciones $i = 0, \ldots, n-2$, siendo n la cantidad de elementos del arreglo. La diferencia va a estar en que vamos a transformar el arreglo en un max-heap, utilizando el algoritmo de Floyd en $O(n)$, y luego vamos a desencolar n veces el máximo, colocando a los elementos en orden creciente. Para hacer el

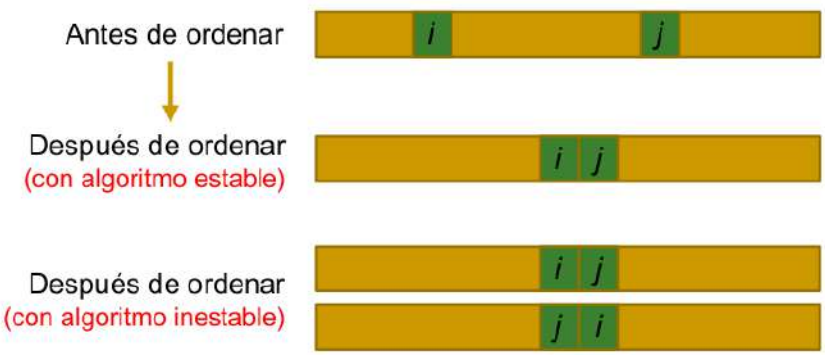

Figura 4.1: Estabilidad

algoritmo *in-place*, es decir, reutilizar el mismo arreglo en el que vienen los datos, vamos a aprovechar el hecho de que al momento de desencolar el i-ésimo elemento del heap, nos van a quedar libres los últimos $n - i$ espacios, en donde podemos ir colocando a los elementos a medida que los vamos desencolando. El costo de este algoritmo es $O(n \cdot \log n)$.

Veamos ahora si este algoritmo es o no estable. Supongamos que tenemos el siguiente arreglo:

$$A = [(10, a), (5, a), (5, b), (5, c), (4, a), (3, a), (2, b)],$$

y veamos cómo el algoritmo termina cambiando el orden relativo entre $(5, a), (5, b)$ y $(5, c)$. Al aplicar el algoritmo de Floyd, como este arreglo ya es un heap, no se hace ningún intercambio, por lo que se mantiene el orden inicial del arreglo. El siguiente paso es aplicar $n - 1$ veces la operación de desencolar en el max-heap. Primero se desencola el $(10, a)$, y como ambos hijos tienen la misma clave, vamos a suponer que en caso de empate se baja por la rama izquierda, quedándonos como raíz a $(5, a)$. Ahora, $(5, a)$ tiene como hijo izquierdo a $(5, c)$ y tiene como hijo derecho a $(5, b)$, por lo que si desencolamos, nos queda como raíz a $(5, c)$. Si seguimos desencolando, lo que nos termina quedando es el arreglo ordenado $\langle(10, a), (5, a), (5, c), (5, b), (4, a), (3, a), (2, a)\rangle$, que no respeta el orden relativo original de las claves $(5, a), (5, b)$ y $(5, c)$. El caso en el que se baja por la rama derecha en caso de empate es similar. Por lo tanto, podemos concluir que el algoritmo no es estable.

4.2. Merge Sort

Es un algoritmo basado en la técnica de *Divide* & *Conquer*, técnica que se basa en *dividir* un problema en problemas similares, pero de tamaño más chico, llamados *sub-problemas*, para luego *combinar* las soluciones de los sub-problemas y así obtener la solución al problema original.

El algoritmo consiste en dividir al arreglo a la mitad, y a cada mitad del arreglo la ordenamos de forma recursiva. En caso de llegar a un arreglo conformado por un solo elemento, este arreglo ya está ordenado, por lo que podemos terminar la recursión. Una vez tenemos ambas partes ordenadas, las unimos haciendo un *merge* en $O(n)$ (es por este motivo que el algoritmo se llama *mergesort*).

Merge Sort

Entrada: Arreglo desordenado de n elementos
if $n < 2$ **then**
    El arreglo está ordenado.
else
    Dividir el arreglo en dos partes iguales de $n/2$ elementos.
    Ordenar recursivamente ambas mitades.
    Combinar ambas mitades ya ordenadas en un único arreglo

Podemos ver que si intentamos calcular el costo en el peor caso de este algoritmo, nos queda la

siguiente recurrencia:

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ 2T(n/2) + O(n) & \text{caso contrario} \end{cases}$$

y considerando que a lo sumo vamos a tener $O(\log n)$ recursiones, por el hecho de que en cada paso dividimos a la mitad el tamaño del arreglo, podemos intuir que el costo total nos queda en $\mathcal{O}(n \cdot \log n)$. Para esta demostración, se asume que n es potencia de 2. Esto se puede hacer ya que la función es **suave**, es decir, vale que $b \cdot f \in O(f)$ para todo $b \geq 2$. Si este no fuera el caso, no sería una cota exacta. Luego, se utiliza alguno de los métodos de resolución de recurrencias presentados en el capítulo de *Divide and Conquer*.

4.3. Quick Sort

Vamos a ver un algoritmo de ordenamiento que probablemente sea el algoritmo de ordenamiento más usado que cualquier otro, llamado *quicksort*. Es popular al no ser demasiado complicado de implementar, funciona bien en una gran variedad de situaciones, y consume menos recursos que cualquier otro algoritmo de ordenamiento en muchas situaciones. Entre sus ventajas principales podemos encontrar que puede realizarse *in-place*, que requiere $n \log n$ operaciones en el caso promedio, y que tiene un ciclo interno extremadamente sencillo (se incrementa un puntero y se compara un elemento del arreglo contra un valor fijo). Entre sus principales desventajas, podemos mencionar que es un algoritmo que suele implementarse de forma recursiva (la versión iterativa es complicada), tiene un costo del peor caso en $O(n^2)$, y es sencillo meter la pata en la implementación. Una versión optimizada de Quicksort suele ser significativamente más eficiente que cualquier otro método de ordenamiento en la mayoría de las computadoras.

La esencia de quicksort es ordenar un arreglo $A[1], \ldots, A[n]$ a través de elegir algún valor v en el arreglo como elemento *pivote*, alrededor del cual se ordena el arreglo. Idealmente, el pivote estaría lo más cercano a la clave mediana del arreglo, es decir, que sea precedida por la aproximadamente la mitad de las claves y que le siga la otra mitad. Luego, permutamos los elementos del arreglo de manera tal que para algún j, todos los registros con claves menores que v aparezcan en $A[1], \ldots, A[j]$, y que todos aquellos cuyas claves sean mayores o iguales a v aparezcan en $A[j+1], \ldots, A[n]$. Se continua aplicando recursivamente quicksort sobre ambos sub-arreglos. Como todas las claves del primer grupo son menores que todas las claves del segundo grupo, el arreglo termina ordenado. Notemos que la recursión termina cuando tenemos un arreglo un solo elemento, y que el algoritmo es *in-place*.

El problema que surge es cómo hacemos para encontrar al elemento mediano. Si bien existen algoritmos lineales que permiten encontrar el elemento mediano en un arreglo, estos vienen acompañados por una constante lo suficientemente alta como para que no nos resulte conveniente utilizarlos en la práctica. Luego, la idea va a ser tomar un elemento al que llamaremos *pivote*, y asumir que es lo suficientemente cercano al mediano.

Quick Sort

```
   Entrada: arreglo de n elementos
1  if array.length > 1 then
2      Elegir pivote
3      while Haya elementos en el array do
4          if elemento generico < pivote then
5              insertar elemento en subarray1
6          else
7              insertar elemento en subarray2
8      quicksort(subarray1)
9      quicksort(subarray2)
```

El costo de este algoritmo depende de la cantidad de comparaciones que se realicen a lo largo de las recursiones. En el peor caso, el pivote elegido siempre va a dividir el arreglo en un sub-arreglo de un solo elemento y en otro con los $n-1$ elementos restantes, por lo que nos quedaría un costo de $O(n^2)$. En el mejor caso y en el caso promedio, se puede demostrar que nos queda un costo $O(n \cdot \log n)$.

En la práctica, la elección del pivot es fundamental, y se suelen tomar medidas para evitar caer en el peor caso. Algunos ejemplos pueden ser:

- Usar la implementación iterativa (importante cuando trabajamos con arreglos muy grandes, ya que el stack de la recursión podría exceder la capacidad de memoria disponible).
- Permutar el input.
- Elegir el pivote al azar, obteniendo una probabilidad baja para el peor caso. Este es un ejemplo simple de algoritmo probabilístico, que utiliza la aleatoriedad para mejorar la eficiencia, sin importar el orden original del input. Sin embargo, para quicksort es probablemente exagerado utilizar un generador de números completamente aleatorios solo para este propósito: un número arbitrario normalmente funcionaría igual de bien.
- Una mejora más útil consiste en tomar 3 elementos del arreglo (normalmente se elige uno de la izquierda, uno del medio y uno de la derecha), y luego quedarnos con la mediana de los tres como pivote (se obtiene una mejora de un 5 %). Esta idea se puede generalizar tomando k elementos, obteniendo la mediana con algún algoritmo eficiente para dicho propósito (la mejora sería marginal con respecto a tomar $k = 3$).
- Llamar a los algoritmos simples de ordenamiento cuando los arreglos son lo suficientemente chicos como para que resulten más eficientes que quicksort (normalmente se toma como umbral algún número entre 5 y 25), pudiendo obtener cerca de un 20 % de mejora en la mayoría de las aplicaciones.
- Si las claves son grandes y costosas de mover, podemos usar punteros para no tener que mover las claves de lugar, a costa de un mayor uso de memoria y que acceder a las claves para realizar las comparaciones es más lento.

4.4. Árboles de decisión

Vamos a centrarnos en los algoritmos de ordenamiento que solo usan a los elementos para comparar dos claves. Podemos dibujar un árbol binario en el que cada nodo representa el *estado* del programa luego de realizar una cierta cantidad de comparaciones de claves. También podemos ver a un nodo como representante del orden inicial de los datos que traen al programa a este estado. Luego, un estado de un programa es esencialmente el conocimiento acerca del orden inicial del arreglo obtenido hasta el momento por el programa. (La idea es que la ejecución de cualquier algoritmo de ordenamiento basado en comparaciones debe comportarse distinto para cualquiera de las $n!$ permutaciones del arreglo, y mostrar cuántos estados debe recorrer la máquina para que el algoritmo pueda saber cuál es el ordenamiento inicial del arreglo.)

Si cualquier nodo representa dos o más posibles ordenamientos iniciales, entonces el programa no puede saber el ordenamiento correcto, por lo que debe realizar otra comparación de claves, como ¿es $k_1 < k_2$? Luego, podemos crear dos hijos para el nodo, donde el hijo izquierdo represente aquellos ordenamientos iniciales consistentes con que $k_1 < k_2$; mientras que el hijo derecho representaría aquellos ordenamientos que sean consistentes con el hecho de que $k_1 > k_2$. Entonces, cada hijo representa el estado consistente en la información acerca del padre más el hecho de que $k_1 < k_2$ o que $k_1 > k_2$, dependiendo si es el hijo izquierdo o el derecho.

En general, si tenemos que ordenar una lista de n elementos, existen $n!$ posibles salidas, que se corresponden con los posibles ordenamientos iniciales de la lista. Es decir, cualquiera de los n elementos podría ser el primero, cualquiera de los $n-1$ restantes podría ser el segundo, etc. Por lo tanto, cualquier árbol de decisión que describa un correcto algoritmo de ordenamiento debe tener al menos $n!$ hojas. La longitud de un camino a una hoja nos da la cantidad de comparaciones que se necesitan para ordenar una lista. Luego, la longitud del camino más largo desde la raíz hasta una hoja es una cota inferior a la cantidad de pasos realizados por cualquier algoritmo de ordenamiento en el peor caso. Si un árbol tiene $n!$ hojas, sabemos que al menos tiene altura $\log n!$, por lo que cualquier algoritmo de ordenamiento debe ser $\Omega(\log(n!)) = \Omega(n \log n)$ (por la aproximación de Stirling) en el peor caso. Esto nos dice que que tanto mergesort como heapsort tienen complejidad asintótica óptima. Esta cota también vale para las operaciones de los árboles AVL, al poder recorrerlos *in-order* para ordenar un arreglo. Esta cota sobre el

costo asintótico de los algoritmos de ordenamiento basados en comparaciones también vale para el caso promedio.

4.5. Bin Sorting

Se suele definir los métodos de ordenamiento en términos de operaciones básicas de comparación entre dos claves y el intercambio de lugar entre la mismas. Todos los métodos de ordenamiento que estuvimos viendo pueden describirse en términos de estas dos operaciones. Vimos que para este tipo de algoritmos se necesitan de $\Omega(n \log n)$ pasos para ordenar n elementos, si no asumimos nada acerca de las claves.

Sin embargo, para muchas aplicaciones, es posible tomar ventaja del hecho de que las claves pueden ser pensadas como números en algún rango restringido, siendo posible que podamos ordenarlas en menor tiempo que $O(n \log n)$. Supongamos que queremos ordenar un arreglo de enteros A que sabemos que están en un rango $1, \ldots, n$, sin duplicados, donde n es la cantidad de elementos. Entonces, podemos ordenar los elementos colocándolos en un arreglo auxiliar B según las claves:

Bin Sorting

```
1 for i = 1, ..., n do
2     B[A[i]] = A[i]
```

Este código calcula a dónde pertenece el registro $A[i]$ y lo coloca en su lugar, en tiempo $O(n)$. Funciona correctamente cuando tenemos exactamente un registro con clave v para cada valor de v en $1, \ldots, n$. Este es un ejemplo de "bin-sort", un proceso de ordenamiento en el que creamos un "bin" para que almacene todos los registros con cierto valor. En el caso general, vamos a tener que prepararnos para almacenar más de un registro en un bin y poder concatenarlos en el orden correcto. Para ello, vamos a tener un arreglo B de listas, que vendrían siendo los bins para cada tipo de clave. En general, si tenemos n elementos a ordenar y tenemos m claves distintas, el costo total de bin sorting nos queda en $O(n+m)$. Si tenemos que el número de claves $m \leq n$, entonces $O(n + m) = O(n)$.

4.5.1. Radix Sorting

Si $m = n^2$, el algoritmo anterior nos quedaría en $O(n^2)$. La pregunta que nos surge es, ¿se puede mejorar? La respuesta es que incluso si los posibles valores de las claves son $1, 2, \ldots, n^k$, para cualquier k fijo, entonces existe una técnica que generaliza bin sorting que toma tiempo $O(n)$.

La idea es que en lugar de comparar las claves completas, vamos a procesar y comparar partes de las mismas. Consideremos el caso en el que tenemos que ordenar n enteros en el rango de valores $0, \ldots, n^2 - 1$. Vamos a ordenar a los n elementos en dos pasos. Primero usamos n bins, uno por cada entero $0, \ldots, n-1$. Colocamos cada entero i en el bin número $imodn$, al final de la lista. Por ejemplo, si tenemos $n = 10$ y el arreglo

$$A = [36, 9, 0, 25, 1, 49, 64, 16, 81, 4],$$

al finalizar el primer paso nos quedan 10 bins ordenados internamente en el mismo orden de aparición de la lista original. Luego, concatenamos los bins y nos queda

$$A = [0, 1, 81, 64, 4, 25, 36, 16, 8, 49].$$

Como segundo paso, lo que hacemos es volver a distribuir los números de la nueva lista, pero usando una estrategia de selección distinta. Ahora, en lugar de considerar el dígito menos significativo, vamos a mirar únicamente el dígito más significativo. Por lo tanto, vamos a colocar el i-ésimo elemento en el bin $\lfloor A[i]/n \rfloor$, respetando el orden de los elementos en la lista (iteramos sobre $i = 1, \ldots, n$). Cuando concatenamos esta segunda lista, terminamos con una lista ordenada. Para ver por qué este algoritmo funciona, nos tenemos que dar cuenta que cuando colocamos varios enteros en un bin, estos deben estar en orden creciente, ya que la lista resultante del primer paso los había ordenado por el dígito de más a la derecha (el menos significativo). Por lo tanto, en cualquier bin tenemos que los dígitos de más a la derecha forman una secuencia ordenada. De forma más general, podemos pensar a los enteros entre 0 y $n^2 - 1$ como números de dos dígitos en base n y usar el mismo argumento para ver que la estrategia

funciona. Este algoritmo se conoce como *Radix Sorting*. La idea central detrás de radix sorting es hacer bin sort sobre todos los registros, primero en f_k, el dígito menos significativo, luego concatenar los bins, manteniendo los valores más chicos primero, volver a aplicar bin sort en f_{k-1}, y así hasta terminar con la lista ordenada. El costo total de radix sorting, para claves en el rango $0, \ldots, n^k - 1$ es de $O(n+kn) = O(n)$. Si k aumenta a medida que aumenta n, por ejemplo si las claves son strings de longitud $\log n$, entonces nos quedaría $O(n \log n)$.

Un algoritmo de ordenamiento lineal evidentemente es deseable para muchas aplicaciones, pero hay razones por las cuales radix sort no es tan bueno como parece ser. Primero, su eficiencia realmente depende en que los dígitos de las claves estén uniformemente distribuidos. Si esta condición no se satisface, es probable tener una severa degradación en la eficiencia. En segundo lugar, se requiere de espacio adicional requerido para los punteros de las listas comparado con otros algoritmos de ordenamiento in-place. En tercer lugar, el ciclo interno del programa contiene varias instrucciones, por lo que si bien es lineal, no sería más rápido que quicksort, salvo para arreglos muy grandes (en donde el espacio adicional requerido por radix sort empieza a ser un factor importante a tener en cuenta). La elección entre quicksort y radix sort es difícil, y es probable de que no solo dependa de las características de la aplicación como el tamaño del arreglo, sino que también de las características del lenguaje y de la máquina en la que se ejecuta, ya que va a depender de la eficiencia en la que se puede acceder a los dígitos de las claves.

Capítulo 5

Dividir y Conquistar

Recordemos que la técnica algorítmica de DC consiste en resolver un problema de forma recursiva, aplicando tres pasos en cada nivel de la recursión. Primero, **dividir** el problema en un número de subproblemas que son instancias más chicas del mismo problema. Luego, **conquistar** los subproblemas resolviéndolos de forma recursiva. Si los subproblemas son lo suficientemente chicos, resolverlos de forma *adhoc*. Para finalmente **combinar** las soluciones de los subproblemas en la solución del problema original (si existe algún costo asociado a la división del problema en subproblemas, lo contamos como parte del costo de la etapa de combinación). Notemos que este método tiene sentido siempre y cuando la división y la combinación no sean operaciones excesivamente costosas.

Algoritmo DC

```
   Entrada: Instancia genérica X
1  if X es lo suficientemente simple como para resolverlo ad hoc then
2      adhoc(X)
3  else
4      Descomponer X en sub-instancias X_1, X_2, ..., X_k
5      for i = 1...k do
6          Y_i ← DC(X_i)
7      Combinar las soluciones de Y_i para obtener una solución Y para la instancia original X.
```

Las recurrencias van de la mano con el paradigma de DC, porque nos dan una forma natural de caracterizar los tiempos de ejecución de este tipo de algoritmos. Una *recurrencia* es una ecuación o inecuación que describe la función en términos de su valor en entradas más chicas. Por ejemplo, vimos que podíamos describir el costo de *merge sort* por la recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

cuya solución dijimos (sin demostrar) que intuitivamente era $T(n) \in \Theta(n \cdot \log n)$.

Las recurrencias pueden tomar muchas formas. Por ejemplo, un algoritmo recursivo podría dividir un problema en subproblemas de distintos tamaños (uno de tamaño 2/3 y otro de tamaño 1/3). El tamaño de los subproblemas no necesariamente está restringido a ser una fracción constante del tamaño del problema original. Por ejemplo, una versión recursiva de la búsqueda secuencial crearía un subproblema conteniendo solo un elemento menos que el problema original. Vamos a ver tres métodos para resolver recurrencias:

- El *método de sustitución*, en el que probamos una cota y usamos inducción para probar que es correcta.
- El *método del árbol de recurrencia*, que convierte la recurrencia en un árbol cuyos nodos representan

los costos asociados a los niveles de la recursión. Usamos técnicas para acotar sumas para resolver la recurrencia.

- El *método maestro* que provee cotas para recurrencias de la forma

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{si } n > 1 \\ 1 & \text{si } n = 1 \end{cases}$$

con $a \geq 1, b > 1$ y $f(n)$ una función dada. Este último tipo de recurrencias aparecen frecuentemente, ya que caracterizan a los algoritmos DC que crean a subproblemas, cada uno de tamaño n/b, y que se pueden dividir y combinar en tiempo $f(n)$. Para ver otros métodos de resolución de ecuaciones de recurrencia, ver 4.7 de [4] y el método Akra-Bazzi.

5.1. Método de sustitución

El método de sustitución para resolver recurrencias se compone de dos pasos:

1. Suponer que la función tiene cierta forma.
2. Usar inducción para encontrar las constantes y mostrar que la solución es correcta.

Como ejemplo, determinemos una cota superior para la recurrencia

$$T(n) = 2T(n/2) + n.$$

Suponemos que la solución es $T(n) = O(n \log n)$. Debemos probar que $T(n) \leq c \cdot n \log n$ para una cierta constante $c > 0$. Empezamos asumiendo que vale para todos los positivos $n' < n$, por lo que $T(n/2) \leq c \cdot \frac{n}{2} \log(n/2)$. Si sustituimos en la recurrencia, nos queda que

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2\left(\frac{cn}{2} \log\left(\frac{n}{2}\right)\right) + n \\ &= cn \log(n/2) + n \\ &= cn \log(n) - cn \log(2) + n \\ &\leq cn \log(n) \qquad \text{si } c \geq 1. \end{aligned}$$

Para completar la demostración, nos falta ver los casos bases. Por la definición de costo asintótico debemos probar que $T(n) \leq cn \log n$ para $n \leq n_0$, donde n_0 es una constante que podemos elegir, por lo que no hace falta que la desigualdad valga par $n_0 = 1$. Luego, tomamos el caso $n_0 = 2$ y como toda recursión de $T(n)$ con $n \geq 2$ debe derivar en $T(2)$ o $T(3)$, basta con probar que existe un c tal que $T(2) \leq 2c \log(2)$ y $T(3) \leq 3c \log(3)$. Como $c = 2$ cumple, queda demostrado que $T(n) \in O(n \log n)$.

A veces podemos acertar en la cota asintótica de la solución de recurrencia, pero fallamos en demostrarla por inducción. El problema suele ser que estamos tomando una hipótesis inductiva demasiado débil como para probar la cota. Consideremos la recursión

$$T(n) = T(n/2) + T(n/2) + 1$$

Claramente, $T(n) \in O(n)$, por lo que intentamos probar que $T(n) \leq cn$. Si sustituimos en la recurrencia nos queda:

$$T(n) = cn/2 + cn/2 + 1 = cn + 1$$

que no implica que $T(n) \leq cn$ para cualquier elección de c. Lo que podemos hacer es *sustraer* un término de menor grado de nuestra solución. Es decir, vamos a probar que $T(n) \leq cn - d$, con $d \geq 0$ constante. Ahora nos queda que

$$T(n) \leq cn - 2d + 1$$

y tomando $d \geq 1$ quedaría demostrado que $T(n) \leq cn - d$. Notemos que no podíamos decir que

$$cn + 1 \in O(n) \;\Rightarrow\; T(n) \in O(n),$$

porque estaríamos usando como argumento lo que queríamos demostrar: habíamos supuesto que $T(n) \leq cn$ y no lo habíamos demostrado.

A veces, nos puede servir hacer un cambio de variables para facilitarnos la resolución de la recurrencia. Por ejemplo, si tenemos

$$T(n) = 2T(\sqrt{n}) + \log n$$

no se nos ocurre qué forma tiene a simple vista. Sin embargo, podemos simplificar esta recurrencia haciendo un cambio de variables, renombrando $m = \log n$. Entonces

$$T(2^m) = 2T(2^{m/2}) + m,$$

y si llamamos $S(m) = T(2^m)$ nos queda la siguiente recurrencia

$$S(m) = 2S(m/2) + m,$$

que ya sabemos resolver (es la misma recurrencia que la de *merge sort*). Volviendo de $S(m)$ a $T(n)$, nos queda

$$T(n) = S(m) = O(m \log m) = O(\log n \cdot \log(\log n)).$$

5.2. Árbol de Recurrencia

Uno de los problemas que tiene el método de sustitución es que no conocemos ningún método general para conocer la forma que tiene la función. Afortunadamente, podemos usar los árboles de recurrencia para generar buenas aproximaciones. En un *árbol de recurrencia*, cada nodo representa el costo de algún subproblema en el conjunto de llamados recursivos (o sea, el costo de la etapa de *combine* si es un nodo interno o el costo de la etapa de *adhoc* si es un nodo hoja en el árbol de recursión). Podemos sumar los costos de cada nivel para obtener un conjunto de costos por nivel, y luego sumar a estos costos por nivel para obtener el costo total de la recursión. Si tenemos el suficiente cuidado con estas cuentas, podemos usarlo como solución de una función de recurrencia; sino, debemos formalizar mediante el método de sustitución. Veamos un ejemplo.

Queremos conocer una cota superior para $T(n) = 3T(n/4) + cn^2$. En la Fig. 5.1 podemos ver cómo derivar el árbol de recurrencia. La parte (a) de la figura muestra $T(n)$, que se expande en la parte (b) en un árbol equivalente representando la recurrencia. El término cn^2 en la raíz representa los costos asociados a la cima de la recursión, y los tres subárboles de la raíz representan el costo asociado a resolver los subproblemas de tamaño $n/4$. En la parte (c) vemos el siguiente paso de este proceso, expandiendo cada nodo con costo $T(n/4)$ de la parte (b). El costo para cada uno de los tres hijos de la raíz es $c(n/4)^2$. Continuamos expandiendo cada nodo en el árbol siguiendo la recurrencia.

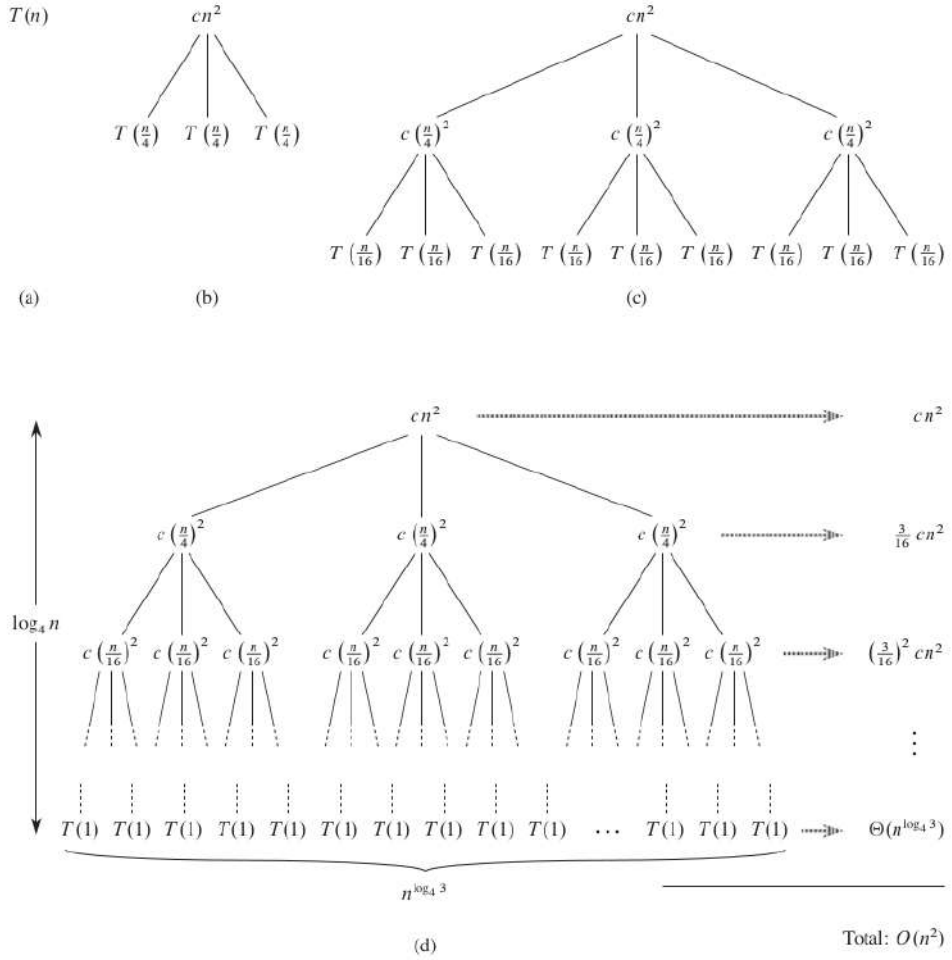

Figura 5.1: Construyendo un árbol de recurrencia para $T(n) = 3T(n/4) + cn^2$. Parte **(a)** muestra $T(n)$, que se va expandiendo progresivamente a **(b)-(d)** para formar el árbol de recurrencia. El árbol completamente expandido en la parte (d) tiene altura $\log_4 n$.

Como el tamaño de cada subproblema decrece por un factor de 4 en cada nivel, eventualmente vamos a llegar a un caso base que sabemos resolver *adhoc*. Para simplificar, suponemos que el caso base es cuando $n = 1$, por lo que llegamos a este caso luego de $\log_4 n$ niveles. Por lo tanto, nos queda un árbol de altura $\log_4 n + 1$.

Ahora debemos determinar el costo de cada nivel del árbol. Sabemos que cada nivel tiene 3 veces más nodos que el nivel de arriba, por lo que la cantidad de nodos a profundidad i es 3^i. Por otro lado, como el tamaño de los subproblemas se reduce por un factor de 4 en cada nivel, tenemos que cada nodo a profundidad i tiene un costo asociado de $c(n/4^i)^2$. Multiplicando ambos resultados, podemos ver que el costo total de todos los nodos a profundidad i es $(3/16)^i cn^2$, exceptuando a las hojas. El último nivel, como está a profundidad $\log_4 n$, tiene $3^{\log_4 n} = n^{\log_4 3}$ nodos, y como cada nodo cuesta $T(1)$, nos queda que el último nivel cuesta $\Theta(n^{\log_4 3})$. Si sumamos todos los costos por nivel, nos queda

$$\begin{aligned} T(n) &= n^{\log_4 3} + \sum_{i=0}^{log_4 n} \left(\frac{3}{16}\right)^i \cdot cn^2 \\ &= n^{0,8} + cn^2 \underbrace{\sum_{i=0}^{log_4 n} \left(\frac{3}{16}\right)^i}_{\text{constante}} \end{aligned}$$

por lo que $T(n) \in O(n^2)$.

5.3. Método Maestro

El método maestro nos da una receta para resolver recurrencias del tipo

$$T(n) = aT(n/b) + f(n),$$

donde $a \geq 1$ y $b > 1$ son constantes positivas y $f(n)$ es una función asintóticamente positiva. Este tipo de recurrencias describen algoritmos que dividen a un problema de tamaño n en a subproblemas de tamaño n/b que se resuelven en tiempo $T(n/b)$. La función $f(n)$ engloba todos los costos asociados a la división y combinación de los resultados de los subproblemas.

El método maestro depende del siguiente teorema.

Teorema Maestro: Sea $a \geq 1$ y $b > 1$ constantes, $f(n)$ una función, y sea $T(n)$ definida en los enteros no negativos por la recurrencia

$$T(n) = aT(n/b) + f(n).$$

Entonces, $T(n)$ tiene las siguientes cotas asintóticas:

1. Si $f(n) \in O(n^{\log_b(a)-\epsilon})$ para alguna constante $\epsilon > 0$, entonces $T(n) \in \Theta(n^{\log_b(a)})$.
2. Si $f(n) \in \Theta(n^{\log_b a})$, entonces $T(n) \in \Theta(n^{\log_b a} \cdot \log n)$.
3. Si $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ para alguna constante $\epsilon > 0$, y si para todo n suficientemente grande vale que $af(n/b) \leq cf(n)$ para alguna constante $c < 1$, entonces $T(n) \in \Theta(f(n))$.

En los tres casos, estamos comparando la función $f(n)$ con $n^{\log_b a}$. Intuitivamente, el mayor de las dos funciones determina la solución a la recurrencia. Por ejemplo, en el caso 1 la función $n^{\log_b a}$ es la mayor, y esto determina que $T(n) \in \Theta(n^{\log_b a})$.

Notemos que en el primer caso, $f(n)$ no solo debe ser menor a $n^{\log_b a}$, sino que debe ser *polinomialmente* menor. Esto quiere decir que $f(n)$ debe ser asintóticamente menor que $n^{\log_b a}$ por un factor n^ϵ para alguna constante $\epsilon > 0$. Entonces, existe una brecha entre los casos 1 y 2 cuando $f(n)$ es menor que $n^{\log_b a}$, pero no polinomialmente menor. En el tercer caso, además, no solo $f(n)$ debe ser polinomialmente mayor a $n^{\log_b a}$, sino que debe satisfacerse la condición de que $af(n/b) \leq cf(n)$. Esta condición suele satisfacerse en las funciones polinomialmente acotadas. Si alguna de estas condiciones no se cumplen, no podemos usar el método maestro para resolver la recurrencia.

Capítulo 6

Memoria Secundaria

En este capítulo vamos a ver cómo se resuelven los problemas de *Sorting* y de *Searching* en **memoria secundaria**. Hasta ahora, solo nos interesaba hablar de complejidad asintótica y de TADs. Hay situaciones en las que la abstracción que hacemos para definir los problemas necesita un enfoque práctico que tenga en cuenta las propiedades físicas de una computadora. Muchas aplicaciones de ordenamiento involucran el procesamiento de archivos muy grandes, demasiado grandes como para poder entrar en la memoria principal de cualquier computadora. Los métodos apropiados para este tipo de aplicaciones son llamados métodos *externos*, ya que involucran una gran cantidad de procesamiento externo a la CPU. Principalmente hay dos factores que hacen de los algoritmos externos diferentes de los algoritmos que veníamos viendo. Primero, el costo de acceder a un objeto es mucho mayor que cualquier costo asociado a los cálculos en la CPU. Para tener algún marco de referencia, podemos pensar que el acceso sobre una memoria secundaria es 10^6 veces más lento que el acceso sobre la memoria principal. En segundo lugar, incluso con su mayor costo, hay varias restricciones en el acceso, dependiendo del tipo de memoria externa usada: por ejemplo, los registros en una cinta magnética solo pueden ser accedidos de forma secuencial.

Es por este motivo que se buscan algoritmos y estructuras de datos que minimicen la cantidad de operaciones sobre las memorias secundarias, a pesar de que eso resulte en más operaciones en la memoria principal. La idea es que el análisis asintótico sigue valiendo, pero cuando las constantes son tan grandes, vale la pena pensar soluciones alternativas. En particular, nos vamos a centrar en el problema de ordenamiento y búsqueda, pero con la diferencia que el volumen de los datos hace que los mismos se encuentren, necesariamente, en la memoria secundaria. En este contexto, lo que buscamos es minimizar la cantidad de veces que un elemento es movido de la memoria secundaria a la memoria principal, y que estas transferencias se realicen de la forma más eficiente que permita el hardware.

Vamos a centrarnos en los métodos básicos de ordenamiento sobre cintas magnéticas y discos, porque es probable que estos dispositivos sigan siendo de uso masivo y, además, ilustran dos modos fundamentalmente distintos de acceso que caracterizan muchos sistemas de memoria externa. Para muchos de los dispositivos de memoria secundaria, el tiempo de acceso está dominado por el tiempo que se tarda en posicionar para leer el dato, más aún considerando que los datos se pueden acceder de a bloques en forma simultánea. Por ejemplo, en un disco rígido se leen de a bloques (de tamaño fijo) de bytes, y no de a bytes sueltos. Por este motivo, conviene pensar en algoritmos que accedan a datos contiguos en la memoria secundaria.

6.1. Ordenamiento Externo

La mayoría de los métodos de ordenamiento externo usan la siguiente estrategia general, conocida como *Ordenación-Fusión* (Sort-Merge): primero hacer una pasada sobre el archivo a ser ordenado, dividirlo en bloques de tamaño similar al de la memoria interna, y ordenar estos bloques. Luego, combinar los bloques ordenados, haciendo varias pasadas por el archivo, obteniendo bloques ordenados cada vez más largos, hasta que eventualmente tengamos todo el archivo ordenado. Los datos suelen ser accedidos de forma secuencial, lo que hace a este método apropiado para la mayoría de los dispositivos externos.

Los algoritmos de ordenamiento se esfuerzan por reducir el número de pasadas sobre el archivo y reducir el costo de una sola pasada para que sea lo más cercano al costo de copia posible.

Como la mayoría de los costos en un método de ordenamiento externos se debe a la entrada / salida, podemos dar una medida aproximada del costo de un algoritmo de sort-merge a partir de contar el número de veces en las que cada palabra en el archivo es leída o escrita. Dentro de esta familia de algoritmos de ordenamiento externo, vamos a centrarnos en las siguientes tres implementaciones:

- Fusión Múltiple Equilibrada (Balanced Multiway Merging).
- Selección por sustitución (Replacement Selection).
- Fusión Polifásica (Polyphase Merging).

6.1.1. Fusión Múltiple Equilibrada

La FME es el procedimiento más sencillo basando en la estrategia sort-merge. Para entender su funcionamiento, vamos a seguir los distintos pasos para un archivo de ejemplo. Supongamos que tenemos que ordenar el siguiente archivo:

"EJEMPLODEORDENACIONFUSION"

Los registros están en una cinta y solo pueden ser accedidos secuencialmente (el segundo registro no puede ser leído antes que el primero, etc). Además, en memoria solo hay espacio para un número fijo de registros (vamos a suponer 3 registros), pero disponemos de todas las cintas auxiliares que necesitemos.

El primer paso para el algoritmo de FME consiste en recorrer secuencialmente el archivo de entrada, trayendo hasta 3 registros a la memoria principal de a uno, y ordenándolos en utilizando algún algoritmo de ordenamiento interno, para finalmente escribirlos ordenados en una cinta auxiliar. Ahora, para poder combinar estos bloques, estos deben estar en cintas diferentes (porque el acceso a una cinta es secuencial). Si queremos hacer una fusión de 3 vías, entonces usaríamos tres cintas auxiliares distintas (escribiendo el resultado del i-ésimo bloque leído en la cinta i módulo 3). Si el archivo no es múltiplo de 3 registros, simplemente el último bloque va a quedar con uno o dos elementos vacíos.

FME - Ordenación (Primera Pasada)

```
   Entrada: Archivo a ordenar
1  while Haya elementos en la cinta de entrada do
2      for i = 1, ..., 3 do
3          Leer de a bloque 3 registros, ordenarlos en RAM, y escribirlos en la cinta i.
```

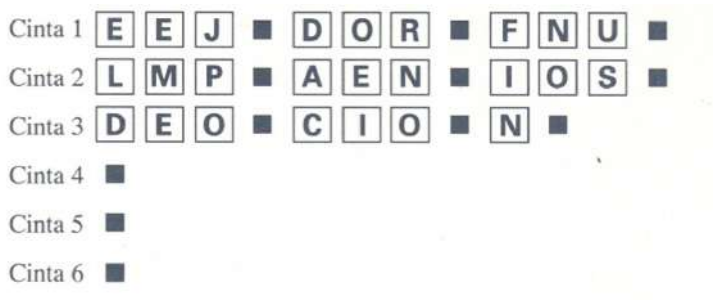

Figura 6.1: Fusión equilibrada de tres vías: resultado de la primera pasada.

Ahora estamos en condiciones de empezar a combinar los bloques ordenados de tamaño 3. Primero, leemos el primer registro de cada cinta auxiliar y escribimos como salida aquella con la clave más chica entre los tres registros. Si hubiésemos usado más de 3 cintas para almacenar los registros, no podríamos hacer este paso, al no alcanzarnos la capacidad de la memoria principal para hacer la fusión.

Luego, se lee el siguiente registro de la misma cinta del registro que acaba de salir, para volver a escribir la clave más chica entre los tres registros. Cuando llegamos al final de un bloque de tres palabras, entonces esa cinta es ignorada hasta que los bloques de las otras dos cintas se terminen de procesar, es decir, hasta que los nueve registros se hayan escrito en la salida. El proceso se repite para combinar los segundos bloques de tres palabras de cada cinta en un segundo bloque de nueve palabras (que se escribe sobre una cinta diferente, y así estar lista para la siguiente combinación). Continuando, tenemos tres bloques largos de 9 elementos ordenados que pueden fusionarse para completar el ordenamiento.

Si tuviésemos un archivo mucho más largo con muchos bloques de tamaño 9 en cada cinta, entonces terminaríamos la segunda pasada con bloques ordenados de 27 elementos en las cintas $1, 2$ y 3. Luego, una tercera pasada produciría bloques de tamaño 81 en las cintas $4, 5$ y 6, y así siguiendo. En conclusión, necesitamos seis cintas para ordenar un archivo de tamaño arbitrario: tres para usarlas como entrada y tres para usarlas como salida de cada una de las combinaciones de 3 vías. Si no tenemos tantas cintas auxiliares, es posible utilizar tan solo 4 cintas: la salida se guardaría en una sola cinta, y los bloques de esta cinta se distribuiría en las tres cintas de entrada en cada paso de fusión.

Este método se conoce como *Fusión Múltiple Equilibrada*: es un algoritmo fácil de implementar y un buen punto inicial para la implementación de algoritmos de ordenamiento externo. Los algoritmos más sofisticados que veremos más adelante pueden ordenar el doble de rápido, pero no mucho más. Sin bien no parece una gran mejora, cuando tenemos tiempos de ejecución muy altos, incluso una pequeña mejora puede ser significativa.

FME - Fusión (Segunda Pasada)

```
   Entrada: Archivo a ordenar
1  while haya segmentos de 3 palabras ordenados en las cintas 1, 2, 3 do
2      for i = 4, ..., 6 do
3          Fusionar los 3 bloques de longitud 3, armando uno de 9 y almacenarlo en la cinta i.
```

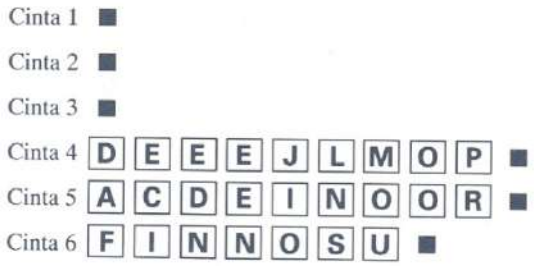

Figura 6.2: Fusión equilibrada de tres vías: resultado de la segunda pasada.

Supongamos que tenemos N palabras a ordenar y una memoria interna de tamaño M. Entonces, el primero paso de ordenamiento produce N/M bloques ordenados. Si hacemos una fusión de P vías en cada paso siguiente, entonces el número de pasadas sobre el archivo es alrededor de $\log_P(N/M)$, ya que cada pasada reduce el número de bloques ordenados en un factor P.

6.1.2. Selección por sustitución

Resulta que los detalles de implementación del método anterior pueden ser desarrollados de forma elegante y eficiente usando colas de prioridad. Primero, vamos a ver que las colas de prioridad proveen una forma natural de implementar una FME. La idea va a ser utilizar una **cola de prioridad** de tamaño M para implementar ambas partes de la FME. La ventaja que nos da la utilización de una cola de prioridad es que vamos a poder generar secuencias ordenadas de mayor longitud en la primera pasada, incluso superando el límite impuesto por la capacidad de la memoria principal (recordemos que en el caso anterior, solo podíamos ordenar 3 registros a la vez, al contar con una memoria de tan solo 3 registros), reduciendo la cantidad de fusiones necesarias para ordenar el archivo.

Lo único que se necesita para realizar la operación de fusión a P vías es poder darle salida al elemento más chico perteneciente a los bloques a fusionar. Una vez extraído ese elemento, se reemplaza con el siguiente elemento del bloque al que pertenecía. Entonces, la idea va a ser utilizar una cola de prioridad para armar esos bloques, y así vamos *encolando* y *desencolando* a los elementos del archivo a través de la cola de prioridad, sustituyendo al elemento más chico por el siguiente del bloque. Esta manera de usar colas de prioridad a veces se llama *selección por sustitución* (replacement selection). Para hacer una fusión de P vías, podemos usar el método de selección por sustitución sobre una cola de prioridad de tamaño P para encontrar en la memoria principal el siguiente elemento a escribir.

Este método no nos resulta de interés por el hecho de que podemos obtener el mínimo en tiempo $O(\log n)$, a diferencia de una solución de fuerza bruta en $O(n)$, ya que estos cálculos los realizamos en la memoria principal, por lo que son despreciables. La importancia real de este método es la manera en la que puede ser usado en la primera etapa del proceso de Ordenación-Fusión, formando bloques ordenados iniciales de mayor longitud en la primera pasada, reduciendo la cantidad de pasadas de fusión necesarias para ordenar el archivo.

La idea es pasar el input (desordenado) a través de una larga cola de prioridad, siempre sacando el elemento más chico en la cola de prioridad, y reemplazarlo siempre con el siguiente elemento de la cinta a la que pertenecía, con una salvedad adicional: si el nuevo elemento es más chico que el que acaba de salir, entonces, no es posible que agregarlo al bloque ordenado actual (de forma eficiente). Sin embargo, lo que podemos hacer es marcarlo como miembro del siguiente bloque y tratarlo como si fuera mayor que todos los otros elementos en el bloque actual. Un elemento marcado de la misma manera que el elemento raíz se considera que está en el bloque actual, y que el resto pertenece al siguiente bloque. De esta manera, evitamos el uso de centinelas, y permitimos generar salidas más largas. Cuando un elemento marcado llega a la cima de la cola de prioridad, el viejo bloque ya terminó y un nuevo bloque acaba de empezar.

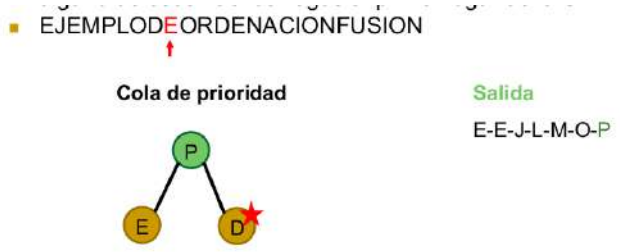

Figura 6.3: Ejemplo de Selección por sustitución.

Una vez terminamos la primera pasada, el algoritmo continúa aplicando fusiones como en FME. Se puede demostrar que, si las claves son aleatorias, las secuencias ordenadas creadas con este algoritmo tienen el doble de longitud que las que podrían producirse usando un método interno (en promedio). El efecto práctico que ganamos con esto es que nos ahorramos una pasada sobre la secuencia: en lugar de iniciar con secuencias ordenadas de tamaño cercano al de la memoria principal, para luego hacer un paso de fusión para producir secuencias del doble del tamaño, podemos empezar directamente con secuencias del doble del tamaño de la memoria, utilizando una cola de prioridad de tamaño M. Si existe algún orden entre las claves, las secuencias serán mucho más largas. Por ejemplo, si ninguna clave tiene más de M claves mayores antes de ella en el archivo, el archivo será completamente ordenado en una sola pasada, y no se necesitará de aplicar fusiones. Esta es la razón práctica más importante por la cual usar este método.

En resumen, el método de selección por sustitución puede ser usado para ambos pasos de ordenar y fusionar de una Fusión Múltiple Balanceada. Para ordenar N registros usando una memoria interna de tamaño M y $P+1$ cintas auxiliares, primero usamos selección por sustitución con una cola de prioridad M para producir secuencias ordenadas iniciales de longitud $2M$ o más, y luego usar selección por sustitución con una cola de prioridad P para hacer $\log_p(N/2M)$ (o menos) pasadas de fusión.

6.1.3. Fusión Polifásica

Uno de los problemas con la fusión múltiple balanceada es que requiere de una cantidad excesiva de cintas auxiliares (o hacer muchas más copias de las necesarias). Para una fusión de P vías debemos usar $2P$ cintas (P como entrada y P como salida) o debemos copiar casi todo el archivo de una sola cinta de salida, distribuyendo esta cinta de salida a las P cintas de entrada, entre cada pasada de fusión (duplicando la cantidad de pasadas). Varios algoritmos de ordenamiento para cintas se han inventado para eliminar virtualmente todas estas copias a través de cambiar la manera en la que los bloques *más cortos* son ordenados y fusionados. El método más conocido es la *Fusión Polifásica.*

La idea básica detrás de la fusión polifásica es distribuir los bloques ordenados producidos por el método de selección por sustitución de manera algo dispareja entre las cintas disponibles, dejando una completamente vacía, y luego aplicar una estrategia de "fusión hasta el vaciado". Cuando alguna cinta de entrada queda vacía, cambia de roles con la cinta de salida. Una de las ventajas que tiene este algoritmo es que nos permite trabajar con un número arbitrario de cintas que dispongamos para el ordenamiento, sin tener la necesidad de copiar registros.

Por ejemplo, supongamos que tenemos tan solo tres cintas, y empezamos con la siguiente configuración inicial de bloques ordenados en cintas. Primero aplicamos selección por sustitución sobre el archivo con una memoria interna que puede almacenar tan solo dos registros. Luego de tres fusiones a dos vías de las cintas 1 y 2 a la cinta 3, la segunda cinta termina vacía y nos quedamos con la siguiente configuración. Después, continuamos con dos fusiones de 2 vías entre las cintas 1 y 3 con salida a la cinta 2, quedando la primera cinta vacía.

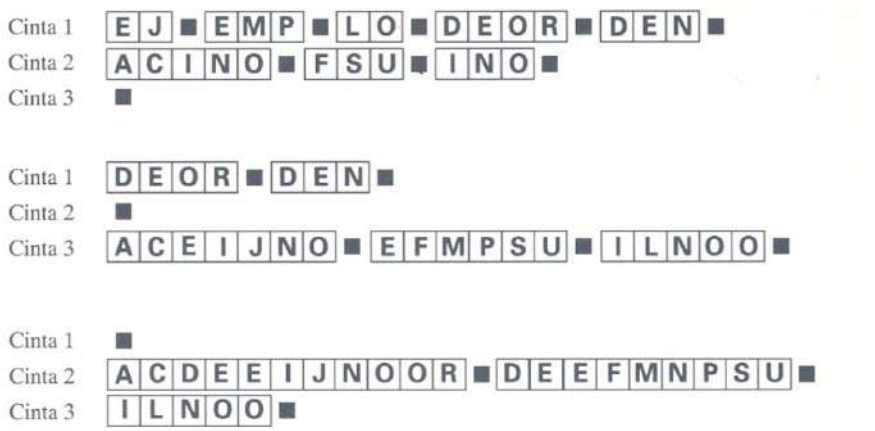

Figura 6.4: Etapas iniciales de una fusión polifásica con tres cintas.

El ordenamiento es completado en dos pasos más. Primero, una fusión a 2 vías entre las cintas 2 y 3 con salida a la cinta 1. Por último, hacemos una fusión a 2 vías entre las cintas 1 y 2 con salida a la cinta 3, terminando con todo el archivo ordenado en la cinta 3.

6.2. Búsqueda Externa

Tener algoritmos de búsqueda externa eficientes es de una inmensa importancia práctica, ya que el acceso a grandes archivos suele representar una fracción bastante significativa del consumo de recursos en una computadora. Vamos a centrarnos en los métodos de búsqueda en discos, porque en los dispositivos secuenciales como las cintas, las búsquedas deben ser secuenciales y no podemos hacer algo mucho mejor que el método trivial de montar la cinta y leerla hasta encontrar el objeto. En cambio, para las memorias secundarias que ofrecen la posibilidad de realizar accesos arbitrarios de forma eficiente, como es el caso de los discos, existen algoritmos y estructuras de datos que nos permiten encontrar un objeto particular entre miles de millones de palabras en tan solo dos o tres accesos.

Vamos a considerar que el disco está dividido en *páginas*, es decir, bloques contiguos de información que pueden ser accedidos de forma eficiente por el hardware del disco, y cada página va a contener

muchos registros; nuestra tarea es organizar los registros dentro de las páginas y las páginas dentro de los discos, de manera tal que cualquier registro pueda ser accedido tan solo leyendo unas pocas páginas. Vamos a asumir que el tiempo de E/S requerido para leer una página completa domina al tiempo de procesamiento requerido para hacer cualquier cómputo que involucre a esa página. Los métodos que vamos a estudiar son:

- Acceso Secuencial Indexado (ISAM).
- Árboles B.
- Hashing Extensible.

6.2.1. Acceso Secuencial Indexado

El método de Acceso Secuencial Indexado combina una organización *secuencial* de claves con la utilización de índices, que nos permiten obtener en unos pocos accesos a disco el registro buscado. Es un método apropiado para aplicaciones en las que los cambios sobre los datos sean poco frecuentes, es decir, en caso de necesitar estructuras estáticas. El problema que tiene es que incluso agregar una sola clave puede implicar una reorganización completa de los discos.

Los registros van a estar almacenados en orden creciente según sus claves, a lo largo de las distintas páginas y discos. Para evitar tener que revisar cada una de las páginas de cada disco, vamos a mantener un *índice* que nos diga qué claves pertenecen a las páginas de qué disco, permitiendo saber si está presente una clave en una página mirando este índice. Cada disco va a tener un índice propio, que se almacena su primer página. Más específicamente, cada índice tiene una entrada por cada página, que nos indica cuál es la mayor clave que se encuentra en la página anterior (la primer entrada se usa para saber cuál es la mayor clave de la última página del disco anterior).

Estos índices son acompañados por un *índice maestro*, que tiene una entrada por cada disco, indicando cuál es la última clave que contiene. Por ejemplo, el índice maestro nos diría que el disco 1 contiene claves $k \leq E$, el disco 2 claves $E \leq k \leq O$, etc. Este índice maestro suele ser lo suficientemente chico como para que fácilmente se guarde en la memoria principal, por lo que podemos garantizar que la mayoría de los registros pueden ser encontrados en tan solo dos accesos a disco: uno para traerse el índice de algún disco apropiado y otro para traerse la página que contiene al registro. Es posible que tengamos que hacer accesos adicionales cuando no podemos saber si el registro está en una página o en la siguiente (en el ejemplo, si la clave es E, no tenemos manera de saber de antemano si el registro está en el disco 1 o en el disco 2).

6.2.2. Árboles B

Un método que funciona bien en el escenario dinámico, en el que no solo accedemos a los registros, sino que también los modificamos, es la utilización de los árboles B para organizar el disco. Consideremos un árbol binario de búsqueda grande, e imaginemos que se ha almacenado en un disco. Si buscamos sobre este árbol de la manera en la que veníamos viendo para algoritmos internos, tendríamos que hacer cerca de $\log n$ accesos a disco antes de completar la búsqueda. Ahora, supongamos que dividimos en lugar de utilizar árboles binarios, utilizamos árboles 6-arios; si accedemos una página a la vez, entonces necesitaríamos tan solo un tercio de los accesos que teníamos antes, por lo que la búsqueda sería tres veces más rápida. El cambio esencial entre un árbol binario y un árbol B es que en estos últimos podemos agrupar muchas más páginas en un solo nodo. Por ejemplo, si tenemos un árbol B con una ramificación de 128 vías en cada nodo, podríamos encontrar cualquier clave en una tabla de un millón de entradas en tan solo tres accesos a disco.

El objetivo que tenemos es minimizar la altura del árbol. Para que el árbol no sea demasiado alto, lo que vamos a hacer es tener mucha ramificación, para lo cual necesitamos poder almacenar muchos registros y punteros en cada nodo. Además, para evitar tener que acceder a más de una página por nivel del árbol, queremos limitar el espacio ocupado por cada nodo a unas pocas páginas (preferentemente una sola). Para ver cómo funcionan los árboles B, primero vamos a ver una variante más sencilla de los mismos, y luego vamos a ver cómo podemos generalizar este caso particular para árboles de cualquier aridad. (La asignación de nodos a páginas de discos requiere de estrategias sofisticadas que no vamos a

ver, pero podemos suponer que en el primer nivel se elige el disco, en el segundo la pista y en el tercero la página.)

Los Árboles 2-3-4 son un caso particular de los Árboles B, en donde los nodos pueden contener $1, 2$ o 3 claves. Un i-nodo contiene $i-1$ claves y de este desprenden i subárboles. La idea es que si en un nodo se tienen dos claves $c_1 < c_2$, el primer subárbol contenga claves menores que c_1, el segundo subárbol contenga claves entre c_1 y c_2, y el tercer subárbol contenga claves mayores que c_2. Una forma de reducir la cantidad de accesos a disco, es mantener la raíz del árbol en memoria, ya que que toda búsqueda va a empezar por la raíz.

Cuando hacemos una inserción, tenemos que preservar el invariante de los árboles 2-3-4. Para ello, lo que se hace es buscar en el árbol la posición que le correspondería a la nueva clave. Si el nodo es un 2-nodo (o un 3-nodo), simplemente agregamos la clave al nodo, quedándonos con un 3-nodo (o un 4-nodo). En caso de que se trate de un 4-nodo, no podemos agregar la clave al nodo (decimos que el nodo está *saturado*). Por lo tanto, lo que vamos a hacer es dividir el 4-nodo en dos 2-nodos, y vamos a pasar una de sus claves hacia arriba en el árbol. En la Fig. 6.5 podemos ver cómo dividimos un 4-nodo, y la idea es que podemos representar a un 4-nodo como un árbol binario de tres elementos (se cumple el mismo invariante y tenemos la misma cantidad de nodos externos).

Supongamos que tenemos que insertar las claves A S E A R C H I N G E X A M P L E en un árbol inicialmente vacío (en ese orden). Empezamos con un 2-nodo (A), luego un 3-nodo (A S), luego un 4-nodo (A E S). Cuando tenemos que poner una segunda A en el 4-nodo, el algoritmo divide el 4-nodo formando un árbol binario antes de insertar la clave A, de manera tal que ahora haya un espacio para A en el fondo del árbol.

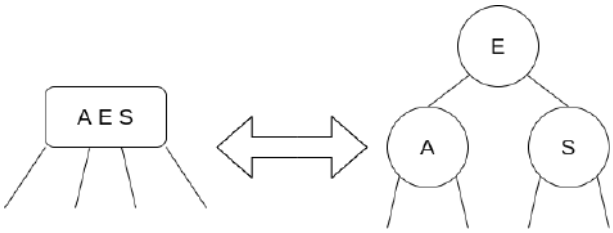

Figura 6.5: Equivalencia entre 4-nodo y árbol binario

En definitiva, cuando tenemos un 4-nodo, este se divide en dos 2-nodos para hacer lugar a la nueva clave, reubicando la clave del medio al nivel de arriba. El problema que nos falta resolver es qué hacemos cuando tenemos que dividir un 4-nodo cuyo padre es también un 4-nodo. Un método sería repetir el procedimiento sobre el padre de manera recursiva, hasta llegar a algún nivel que no tenga su nodo saturado o hasta llegar a la raíz (y simplemente agregar un nuevo nivel como vimos para el caso de A E S). Este tipo de árboles 2-3-4 se conocen como *bottom-up* 2-3-4 trees.

Otra posibilidad es asegurarse que el padre de cualquier nodo que veamos en el camino de bajada no sea un 4-nodo. Si alguno de ellos fuera un 4-nodo, lo dividimos de forma preventiva. Específicamente, cada vez que nos encontremos con un 2-nodo conectado a un 4-nodo, lo deberíamos transformar en un 3-nodo conectado a dos 2-nodos; y cada vez que nos encontremos en el camino de bajada a un 3-nodo conectado a un 4-nodo, deberíamos transformarlos en un 4-nodo conectado a dos 2-nodos. Esta alternativa funciona mejor en la práctica, principalmente para accesos concurrentes a los datos (además de que evita recorrer dos veces los mismos nodos).

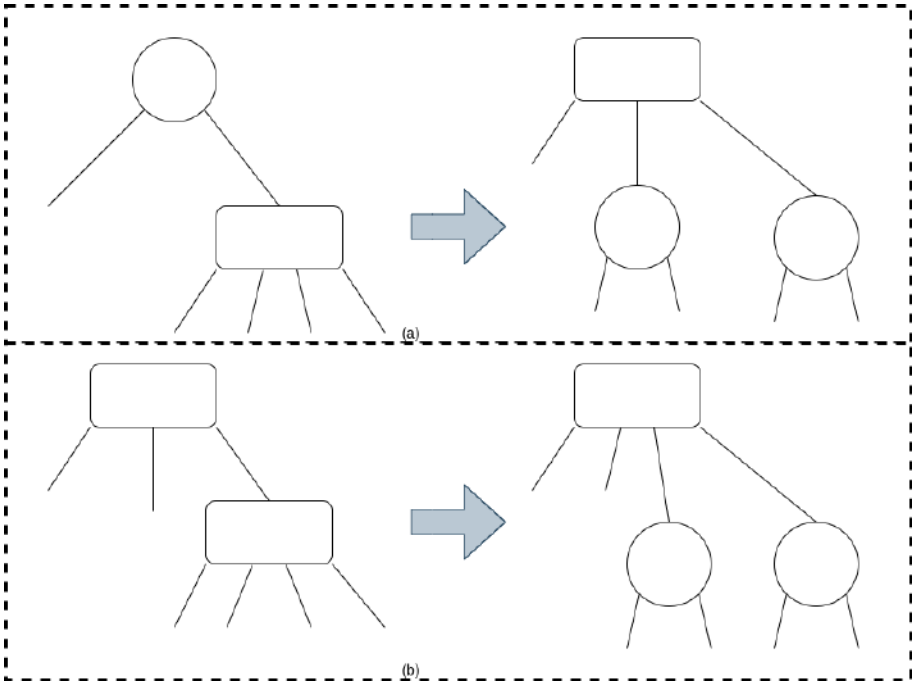

Figura 6.6: (a) 2-nodo conectado a 4-nodo. (b) 3-nodo conectado a 4-nodo.

Cada una de las transformaciones suben una de las claves de un 4-nodo a su padre en el árbol, reestructurando los enlaces de forma acorde. Notemos que no tenemos que preocuparnos explícitamente acerca de si el padre es un 4-nodo, ya que nuestras transformaciones aseguran que a medida que pasamos por cada nodo en el árbol, terminamos con que ninguno es un 4-nodo. Cuando la raíz se vuelve un 4-nodo, vamos a dividirlo en tres 2-nodos, como hicimos en el primer caso A E S, haciendo que el árbol crezca un nivel para arriba. Como los 4-nodos se dividen en el camino de bajada, estos árboles se llaman *top-down* 2-3-4 trees.

Lo interesante es que, incluso sin habernos preocupado por el balanceo, el árbol resultante queda **perfectamente balanceado**, es decir, la distancia de la raíz a cualquier nodo hoja es siempre la misma. Esto es una propiedad muy importante, porque implica que el tiempo requerido para una búsqueda, inserción y borrado es siempre proporcional a $\log n$. Para ver que el árbol siempre queda perfectamente balanceado, simplemente tenemos que darnos cuenta que las operaciones que realizamos no tienen ningún efecto sobre la distancia entre ninguna hoja a la raíz, exceptuando el caso en el que dividimos la raíz (incrementando la distancia de todos los nodos a la raíz en uno).

Los algoritmos top-down que vimos para los árboles 2-3-4 se extienden fácilmente para manejar nodos con más claves, con la diferencia de que cada nodo puede tener hasta $M - 1$ claves y M subárboles. La búsqueda procede de una manera análoga a la de los árboles 2-3-4: para movernos de un nodo al siguiente, primero encontramos el intervalo adecuado para la clave en el nodo actual y luego salimos por el eje correspondiente. Continuamos de esta manera hasta que llegamos al nodo hoja correspondiente, para luego insertar la nueva clave en el último nodo interno alcanzado. Al igual que en los árboles top-down 2-3-4, es necesario dividir los nodos que están saturados en el camino de bajada del árbol: en cualquier momento en el que veamos un k-nodo unido a un M-nodo, lo reemplazamos por un $k + 1$-nodo unido a dos $M/2$-nodos. Esto garantiza que cuando se llega al piso del árbol haya espacio para insertar un nuevo nodo.

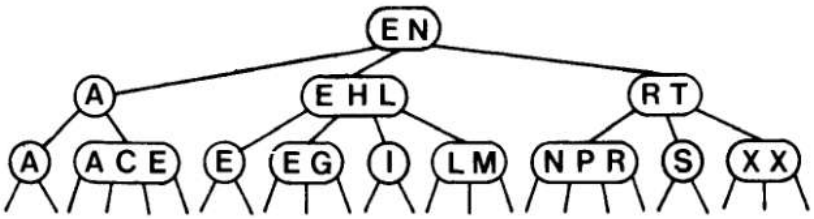

Figura 6.7: Ejemplo de árbol-B construido para $M = 4$ y el archivo E X T E R N A L S E A R C H I N G E X A M P L E

Este árbol tiene 13 nodos, cada uno correspondiente a una página del disco. Recordemos que los discos tenían capacidad de almacenar tres páginas de cuatro registros cada una. Vale la pena enfatizar que en las páginas debemos almacenar tanto los registros como los enlaces a los siguientes nodo, por lo que la cantidad total de espacio aprovechado va a depender del tamaño relativo entre los registros y los enlaces.

Optimizaciones: Muchos nodos en el último nivel del árbol B descripto contienen muchos enlaces sin usar que podrían ser eliminados marcando a estos nodos hoja de alguna manera. Es posible optimizar esta estructura utilizando valores de M mucho más altos para los nodos internos del árbol si en lugar de almacenar los registros, tan solo almacenaran las claves, al igual que hacíamos en el método de acceso secuencial indexado (almacenando todos los registros en los nodos hoja). Por ejemplo, suponiendo que podemos colocar hasta 7 claves y 8 enlaces en una página, podríamos usar $M = 8$ para los nodos internos y $M = 5$ para los nodos hoja. Esta idea nos dejaría con el siguiente árbol:

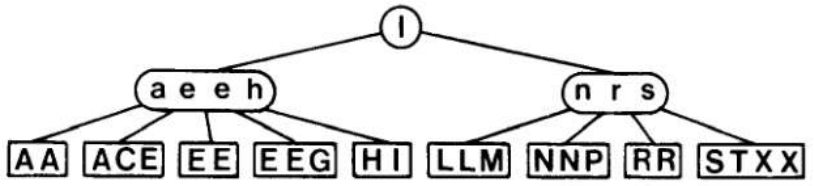

Figura 6.8: Optimización de árbol B

El efecto en una situación real sería mucho más dramático, ya que el factor de ramificación del árbol aumenta según la relación entre el tamaño de un registro y el tamaño de una clave, que suele ser grande. De esta manera, tendríamos dos valores de M: uno para los nodos internos que determina el factor de ramificación del árbol (M_I) y uno para los nodos hoja que determina la posición de cada registro en el disco (M_B). Eligiendo estos valores de forma adecuada, se puede obtener un árbol B cuya altura sea de tan solo 3 niveles. Esta variante de los árboles B se conoce como árboles B^+. Recordemos que como todas las búsquedas involucran el acceso al nodo raíz, este se suele mantener en la memoria principal, de manera tal que podamos encontrar típicamente cualquier elemento en el archivo en tan solo 2 accesos al disco.

Existen otras optimizaciones que se pueden hacer a los árboles B. Por ejemplo, una variación importante bajo el contexto de búsqueda externa es la idea de *overflow*. La idea es mejorar el algoritmo de inserción a través de resistir la tentación de dividir a los nodos de forma tan frecuente; para evitar la división, se hace una rotación local en la que le pasamos claves a un nodo hermano. Esto lleva a una mejor utilización del espacio en los nodos (asegura que se usan 2/3 de la capacidad de cada nodo, en lugar de 1/2), que suele ser de gran importancia cuando trabajamos con aplicaciones de búsqueda sobre discos de gran escala (aunque ralentiza las inserciones). Esta variante se conoce como árboles B^*.

6.2.3. Hashing Extensible

Una alternativa a los árboles B es el método de *hashing extensible*. Este método garantiza que cualquier búsqueda no requiera más de dos accesos a disco. Al igual que en los árboles B, los registros son almacenados en páginas que cuando se llenan, se dividen en dos partes; al igual que en acceso secuencial indexado, vamos a mantener un índice en disco que nos permite encontrar la página que contiene el registro asociado a nuestra clave de búsqueda.

Para decidir a dónde va cada clave, se utiliza una sucesión de funciones hash $h_1, h_2, \ldots h_m$ tales que h_i es una función hash binaria $f : K \to \{0, 1\}$, que dada una clave cualquiera del conjunto de claves posibles, devuelve 0 o 1. Además, vamos a contar con un *índice* (o directorio), el cual nos va a permitir saber en qué página está cada registro (vamos a indexarlo utilizando las funciones $h_1, h_2, \ldots, h_d$, con $d \leq m$). Este índice es dinámico, y se va a ir modificando a medida de que insertemos y borremos registros repartidos en una colección de discos. Cuando el directorio no cabe en una sola página, vamos a utilizar un nodo raíz que mantendremos en memoria principal, que funcionaría como *índice maestro*, indicando dónde están las páginas del directorio (siguiendo el mismo esquema de direccionamiento). Los registros se almacenan de forma ordenada según sus claves (como las claves se ordenan en memoria principal, no resulta costoso).

Para ver cómo funciona el método de hashing extensible, vamos a considerar cómo es que maneja las inserciones sucesivas del siguiente ejemplo: E X T E R N A L S E A R C H I N G E X A M P LE, usando páginas con capacidad de cuatro registros y dos discos. También supongamos que tenemos $h_1, h_2, \ldots, h_5$ funciones de hash y que al aplicar estas funciones sobre las claves del ejemplo obtenemos lo siguiente:

- $h(E) = 00101, h(X) = 11000, h(T) = 10100$,
- $h(R) = 10010, h(N) = 01110, h(A) = 00001$,
- $h(L) = 01100, h(S) = 10011$, etc.,

donde $h(\bullet)$ representa la secuencia dada por $h_1, h_2, \ldots, h_5$.

Inicialmente, tenemos un índice de una sola entrada, que contiene un puntero a la página que almacenará los primeros registros. Los primeros cuatro registros entran en una sola página, por lo que nos queda la siguiente estructura:

- Disco 1: 20.
- Disco 2: EETX.

El directorio almacenado en el disco 1 nos dice que todos los registros están en la página 0 del disco 2 (porque la única entrada que tiene apunta a esa página).

Al intentar insertar el siguiente elemento, nos encontramos con que la página está llena. Lo que vamos a hacer es dividirla en dos páginas, y así poder agregar la clave R = 10010 a alguna de las nuevas páginas. La estrategia para dividir la página es simple: ponemos todos los registros con tales que $h_1(k) = 0$ en una página y en la otra ponemos todos los registros con $h_1(k) = 1$. Esto requiere que dupliquemos el tamaño del directorio y que movamos la mitad de las claves de la página 0 del disco a una nueva página, dejando la siguiente estructura:

- Disco 1: 20 21
- Disco 2: EE RTX

Ahora el directorio en el disco 1 nos dice que todos los registros tales que $h_1(k) = 0$ están en la página 0 del disco 2, mientras que todos registros tales que $h_1(k) = 1$ están en la página 1 del disco 2. Continuamos agregando los registros N = 01110 y A = 00001. Como $h_1(N) = h_1(A) = 0$, ambos registros van a parar a la página 0 del disco 2:

- Disco 1: 20 21
- Disco 2: AEEN RTX

El siguiente elemento a agregar es L = 01100, pero la página 0 del disco 2 está llena, por lo que otra división es necesaria. Como esta página se corresponde con las claves tales que $h_1(k) = 0$, dividimos la página en dos: una para las claves que empiezan con 00 y otra para las claves que empiezan con 01. Además, vamos a duplicar el tamaño del directorio, quedándonos la siguiente estructura:

- Disco 1: 20 21 22 22
- Disco 2: AEE LN RTX

De esta manera, podemos acceder a cualquier registro en tan solo dos accesos a disco: primero accedemos a la página que contiene al índice, lo indexamos usando $h_1(k)$ y $h_2(k)$, obteniendo la entrada con un puntero a la página que contiene el registro, para finalmente traernos la página con el registro buscado.

En general, si en determinado momento necesitamos d funciones de hash para encontrar algún registro, vamos a tener un *directorio* de 2^d entradas (una por cada patrón posible), que apuntan a las páginas que contienen a los registros. Notemos que una página puede ser apuntada por más de una entrada del directorio (como vimos en el ejemplo con las entradas $h(k) = 10$ y $h(k) = 11$). Estos punteros suelen ser mucho más pequeños que los registros, por lo que no estamos desperdiciando demasiada memoria.

Cuando insertamos un elemento en la estructura, su inserción puede involucrar alguna de las siguientes operaciones adicionales, además de la búsqueda en sí de la página en la que se pretende insertar:

1. Si hay espacio en la página, el nuevo registro simplemente es insertado.
2. Si no hay espacio en la página, esta se debe dividir en dos.
3. Si el directorio tiene más de una entrada apuntando a esa página, una de las entradas puede reutilizarse para apuntar a la nueva página. Sino, es necesario duplicar el tamaño del directorio.

Si por causa de un borrado la unión de dos páginas hermanas entra en una única página, lo que se hace es unirlas en una sola página. En la práctica, para evitar posibles secuencias de borrados e inserciones que terminen en divisiones y uniones sucesivas, lo que se hace es considerar que una página se llenó cuando se llega al 80 % de capacidad, y se considera que dos páginas hermanas se deben unir cuando la unión ocupe el 40 % de la capacidad de una página.

Este método presenta una alternativa a los árboles B y al acceso secuencial indexado, ya que usa exactamente dos accesos a disco para cada búsqueda (al igual en el acceso indexado), mientras que retiene la capacidad de inserciones eficientes (al igual que los árboles B).

Capítulo 7

Alternativas a los AVL

En este apartado, vamos a estudiar otras implementaciones eficientes de diccionarios, que no se basan en mantener una estructura balanceada. La primera estructura que vamos a ver se conoce como *Skip List*, y tienen la particularidad de que es una estructura randomizada, es decir, su operaciones son **algoritmos probabilísticos**. Este tipo de algoritmos renuncian al *determinismo*: la propiedad de que siempre que le demos la misma entrada, se obtiene el mismo resultado. Al incorporar el azar a los algoritmos, perdemos en predecibilidad, pero ganamos en eficiencia y elegancia. Dentro de la familia de los algoritmos probabilísticos, podemos diferenciar a los algoritmos del tipo Las Vegas: dan siempre el resultado correcto, pero su tiempo de ejecución $T(n)$ es una variable aleatoria. Notemos que una de las versiones que vimos de Quick Sort es un algoritmo Las Vegas, ya que el pivote se elige al azar y se asegura que al finalizar la ejecución, el arreglo termina ordenado. Otra tipo de algoritmos probabilísticos son los del tipo Montecarlo: los resultados tienen cierta probabilidad de ser falsos, pero siempre tardan lo mismo.

La segunda estructura que vamos a ver, conocida como *Splay Tree*, tiene la particularidad de que no nos da una buena cota sobre el costo *individual* de una operación en el peor caso, sino que nos da una cota sobre el costo de una *secuencia* de operaciones (en el peor caso). A este tipo de análisis se lo conoce como análisis del *costo amortizado*. Además, ofrece una mayor eficiencia en el caso de que los accesos a las claves no sea uniforme, es decir, que haya claves que sean más frecuentemente accedidas que otras.

7.1. Skip Lists

Veamos ahora una estructura de datos randomizada llamada *Skip Lists*. Utiliza listas enlazadas ordenadas para implementar diccionarios. El problema que teníamos con las listas enlazadas era que, sin importar que la lista esté o no ordenada, no podíamos escapar la necesidad de recorrer *secuencialmente* la lista, por lo que las operaciones de búsqueda, inserción y borrado nos quedaban en $O(n)$ en el peor caso.

La idea de las Skip Lists es tener una forma de avanzar más rápido sobre la lista, sin tener que recorrer secuencialmente toda la lista. Para ello, lo que se hace es ir agregando nuevos *niveles* de punteros que apuntan a elementos cada vez más espaciados (exponencialmente). Con esto en mente, lo que vamos a hacer es darle al $c \cdot 2^i$-ésimo nodo (con c impar) un puntero a los nodos en $2^0, 2^1, \ldots, 2^i$ posiciones más adelante en la lista.

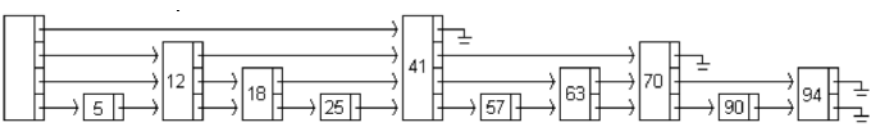

Figura 7.1: Ejemplo de Skip List.

De alguna manera, lo que nos estaría quedando son $\log n$ listas paralelas, de manera tal que por cada *nivel* que subimos en las listas, la distancia entre nodos aumenta de manera exponencial, donde la distancia es entendida como cantidad de elementos intermedios. Para facilitar el acceso al i-ésimo nivel, lo que se hace es tener un primer nodo vacío, que tenga un puntero hacia el primer nodo de cada nivel. Si bien no estamos aumentando la cantidad de nodos de la lista, sí estamos aumentando la cantidad de punteros. Sin embargo, esto no es tan problemático, porque la cantidad total de punteros nos queda:

$$\begin{aligned}\lim_{k\to\infty}\sum_{i=0}^{k}\frac{n}{2^i} &= \lim_{k\to\infty} n\cdot\sum_{i=0}^{k}\left(\frac{1}{2}\right)^i \\ &= \lim_{k\to\infty} n\cdot\frac{1-\overbrace{\left(\frac{1}{2}\right)^{k+1}}^{\to 0}}{1-\frac{1}{2}} \\ &= \lim_{k\to\infty} n\cdot\frac{1}{0{,}5} = 2n\end{aligned}$$

donde n es la cantidad de elementos y k es la cantidad de niveles de la lista ($k = \log n$). Por lo tanto, solo necesitamos el doble de punteros que en una lista tradicional. Para realizar una búsqueda, a lo sumo se necesitan de 2 comparaciones por nivel, y como tenemos $\log n$ niveles, podemos concluir que el costo de la operación de búsqueda nos queda en $O(\log n)$. El problema principal con esta estructura es la rigidez que tiene frente a inserciones y borrados. Este tipo de Skip Lists se conocen como Skip Lists Perfectas, justamente por el hecho de que por cada nivel tenemos exactamente $2, 4, 8, \ldots, 2^k$ punteros.

Para solucionar el problema de la rigidez, lo que se hace es someter a un proceso aleatorio la forma en la que se reconstruye la Skip List. La inserción se divide en dos pasos. Primero ubicamos la posición en la lista que le corresponde al elemento, y luego le asignamos su *nivel* de forma probabilística. Para ello, tiramos una moneda de tal que si sale *cara*, promociona de nivel (y volvemos a iterar); si sale *cruz*, termina el procedimiento. Notemos que todos los nodos empiezan siendo de nivel 1. En el caso promedio, sin tener que asumir propiedades probabilísticas sobre el input, intuitivamente nos va a quedar una Skip List con $O(\log n)$ niveles.

La ventaja con respecto a las estructuras de árboles AVL es que son mucho más simples de programar, extender y modificar. Por este motivo, suelen venir con mejores optimizaciones de implementación (por ejemplo, implementar sus algoritmo de forma iterativa en lugar de forma recursiva). Además, las Skip Lists suelen tener un menor factor constante que las versiones poco optimizadas de los árboles AVL [7].

7.2. Splay Trees

Un Splay Tree es un árbol binario de búsqueda *auto-ajustante* en el que todas las operaciones básicas de diccionario se pueden implementar con un costo amortizado de $O(\log n)$, donde por costo amortizado nos referimos a que el tiempo por operación promedio sobre una secuencia de operaciones para el peor caso. La eficiencia de los splay trees no viene dado por una restricción explícita estructural, como sucede en los árboles balanceados, sino que su eficiencia se debe a la aplicación de una simple heurística de reestructuración, llamada *splaying*, que se aplica cada vez que el árbol es accedido. Además, se mantiene a los elementos más frecuentemente accedidos lo más alto posible en el árbol. Extensiones de la operación de splaying dan formas simplificadas de otras dos estructuras de datos: los árboles de búsqueda lexicográficos o multidimensionales y los link/cut trees[5].

La motivación de esta estructura es la observación de que los distintos tipos de árboles balanceados, si bien tienen una cota en el peor caso de $O(\log n)$, se pueden mejorar en términos de tiempo de acceso, para el caso en el que el patrón de acceso no es uniforme, y en términos de espacio (los Splay Trees no requieren de información de balanceo). Cuando algunas claves son accedidas de forma mucho más frecuente que otras, nos gustaría que las importantes estén relativamente cerca de la raíz (especialmente si el árbol es muy grande). Una estructura que nos *garantiza* esta propiedad son los árboles binario óptimos (ver sección 6.2.2 de [9]).

ABB óptimos: Recordemos que los árboles ABB cumplen con el invariante de que todo elemento en el subárbol derecho de un nodo es mayor este, y todo elemento en el subárbol izquierdo es menor. Si supiéramos las frecuencia de acceso a cada elemento, podríamos armar un árbol en el que los elementos más accedidos estén más cerca de la raíz. Esto se puede hacer en tiempo $O(n^2)$ a partir de un algoritmo basado en la técnica de Programación Dinámica [8]. Sin embargo, estar en una situación en la que sabemos la frecuencia de acceso de cada elemento y que esta no varíe a lo largo del tiempo lo vuelve un planteo poco realista.

Frente al problema de falta de dinamismo en los ABB óptimos, los Splay Trees tratan de tender todo el tiempo al ABB óptimo en todo momento. Estos son más simples de implementar que los árboles AVL, al no tener la necesidad de verificar condiciones de balanceo y no requieren de memoria adicional (por ejemplo, para almacenar factores de balanceo). Decimos que son estructuras *auto-ajustantes*, porque se modifican así mismas cuando operamos sobre ellas. El ajuste que vamos a realizar es que cada vez que accedemos a un elemento, lo movemos hasta la raíz, mediante la aplicación rotaciones. Esta operación es conocida como *splaying*.

Para realizar un splay un árbol binario en un nodo interno x, empezamos en x y recorremos el camino hacia la raíz, en un orden que depende de la estructura del árbol. La operación de splay consiste en repetir el siguiente paso hasta que $p(x)$ quede indefinido, siendo $p(x)$ el padre de x y $p^2(x)$ su abuelo:

a) Zig: si x tiene un padre pero no un abuelo, rotamos en $p(x)$.

b) Zig-Zig: si x tiene un abuelo y x y $p(x)$ son ambos hijos izquierdos o hijos derechos, rotamos en $p^2(x)$ y luego en $p(x)$.

c) Zig-Zag: si x tiene un abuelo y x es un hijo izquierdo y $p(x)$ un hijo derecho, o viceversa, rotamos en $p(x)$ y luego en el nuevo padre de x (su antiguo abuelo).

En definitiva, el efecto de splaying es mover a x hasta la raíz del árbol, permitiendo mantener a los elementos más accedidos lo más alto posible en el árbol.

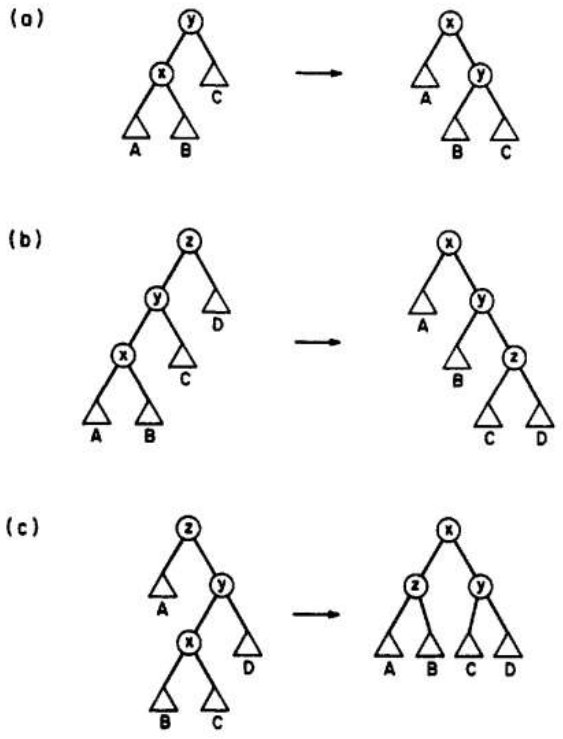

FIG. 4.9. *Cases of splay step. Each case has a symmetric variant (not shown).* (a) *Terminating single rotation.* (b) *Two single rotations.* (c) *Double rotation.*

Notemos que en cualquiera de los tres casos, si aplicamos una operación de splaying sobre un ABB, el árbol resultante sigue siendo un ABB.

Luego de acceder o insertar un elemento i, hacemos una operación de splay en i. Después de borrar un elemento i, hacemos un splay sobre el nodo que era su padre antes del borrado. En cada caso el tiempo de la operación es proporcional a la longitud del camino sobre el que se realiza el splay. Si bien hemos descripto a la operación de splay como bottom-up, existen variantes top-down, en donde efectuamos la operación de splay durante la bajada hasta i. Si bien no se puede garantizar que cada operación cueste $O(\log n)$, sí se puede garantizar un costo de $O(m \log n)$ para m operaciones sucesivas, por lo que podemos decir que las operaciones de Splay Tree tienen costo **amortizado** de $O(\log n)$.

En general, las ventajas que presenta los splay trees sobre los árboles balanceados es que resultan mucho más eficientes si el patrón de uso es sesgado (algunas claves se acceden mucho más que otras), utilizan menos espacio y los algoritmos son más sencillos. Sin embargo, esta estructura presenta dos posibles desventajas. En primer lugar, requieren de más ajustes locales, incluso durante los accesos, por lo que son poco eficientes para aplicaciones multi-threading. Además, las operaciones individuales podrían ser demasiado costosas, lo que sería un problema en aplicaciones real-time.

Listas auto-ajustantes: Lo que queremos es tener una lista que mantenga más a mano a los elementos más recientemente accedidos. Esto se puede hacer aplicando una política Move-to-front (MTF), en la que se mueve el elemento accedido a la cabeza de la lista. Se puede demostrar que el costo total para una secuencia de m operaciones en una lista MTF es de a lo sumo el doble que el de cualquier implementación de diccionario usando listas.

Bibliografía

[1] URL: https://stackoverflow.com/questions/30713269/why-would-one-use-a-heap-over-a-self-balancing-binary-search-tree.

[2] *Advantages of Trie Data Structure*. 2021. URL: https://www.geeksforgeeks.org/advantages-trie-data-structure/.

[3] Thomas H. Cormen et al. *Introduction to Algorithms*. The MIT Press. McGraw-Hill, 2001.

[4] Bratley P. Brassard G. *Fundamental of Algorithmics*. International series of monographs on physics. Prentice Hall, 1995.

[5] R. E. Tarjan D. D. Sleator. «Self-Adjusting Binary Search Trees». En: *Commun. ACM* 32.3 (jul. de 1985), págs. 652-686. URL: https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf.

[6] *Data structures and network algorithms*. Society for Industrial y Applied Mathematics, 1987.

[7] En: *Commun. ACM* 33.3 (1990). Ed. por Peter J. Denning, págs. 668-676. ISSN: 0001-0782. URL: https://homepage.cs.uiowa.edu/~ghosh/skip.pdf.

[8] D. E. Knuth. «Optimum binary search trees». En: *Acta Informatica* 1.1 (mar. de 1971), págs. 14-25. ISSN: 1432-0525. DOI: 10.1007/BF00264289. URL: https://doi.org/10.1007/BF00264289.

[9] D. E. Knuth. *The art of computer programming*. Vol. 3. Addisson Wesley Professional, 1998.

[10] K. Wayne R. Sedgewick. *Algorithms*. Addison Wesley, 1983.

[11] J. D. Ullman V. Aho J. E. Hopcroft. *Data structurs and algorithms*. Addison Wesley, 1985.