Aclaraciones

Este resumen fue hecho con el propósito de volcar todo el conocimiento necesario para el final de algoritmos 2 a medida que estudiaba para el mismo. La motivación fue principalmente, y bajo mi criterio, porque notaba que la información estaba segmentada y necesitaba un poco mas de organización, especialmente los temas de especificación y diseño. Estos dos últimos fueron principalmente extraídos de los apuntes de la cátedra, los cuales contienen la información necesaria para entender perfectamente los temas mas ejemplos. De estos últimos, quise organizarlos a mi modo, de forma resumida, removiendo los ejemplos y dejando solo los aspectos mas teóricos, y agregando algunas clarificaciones para algunos de los temas.

El resumen estará dividido en "capítulos", no porque sea un libro, sino por un tema de orden, ya que los temas quedan agrupados en cuatro categorías generales. Las primeras dos, como dije anteriormente, sus referencias pueden ser encontradas en los apuntes correspondientes a cada tema, las diapositivas de las clases y algunos agregados de mi parte de apuntes o dudas que consulte y resolví. En la tercera sección, las definiciones de las clases de complejidad fue traducida directamente del Cormen[1], como así también las estructuras de datos como ABB, Heap arboles B y arboles Red-Black. Para las estructuras de datos de Splay Trees y Arboles 234 (contenidos dentro de la sección de arboles B), son prácticamente una transcripción de vídeos de clases de la universidad de Berkeley[2] correspondientes a los temas. Finalmente, en la cuarta sección los temas de Dividir y Conquistar y códigos de Huffman, fueron traducidos del Cormen directamente, junto a las definiciones del Teorema Maestro y árbol de recurrencia, los temas de ordenamiento son nuevamente transcripciones de clases de Berkeley[3].

Si bien este resumen intento hacerse a conciencia y con la máxima correctitud y verificación posible, es muy probable que el mismo contenga errores. Es por ello que pido encarecidamente que de encontrarlos sean corregidos o al menos dar una advertencia de los mismos. El código fuente de este archivo, el cual fue producido en LaTeX, estará disponible en mi cuenta de GitHub[4], junto con las instrucciones para modificarlo. Ahí mismo, ademas, se podrá dar notificación de los errores encontrados.

Referencias

- **[1]** Introduction to Algorithms - Thomas H. Cormen
- **[2]** Splay Trees: `http://www.youtube.com/watch?v=G5QIXywcJlY`
- **[2]** 234: `http://www.youtube.com/watch?v=zqrqYXkth6Q`
- **[3]** Sort I: `http://www.youtube.com/watch?v=EiUvYS2DT6I`
- **[4]** `https://github.com/ramaroberto/ResumenFinalAlgo2`

Índice general

Capítulo 1

Tipos Abstractos de Datos

1.1. Introducción y noción

Los tipos abstractos de datos (TADs) son modelos matemáticos que se construyen con el fin de exponer los aspectos relevantes de un problema bajo análisis. La razón por la cual son usados es porque gracias a los mismos es posible realizar un análisis exhaustivo junto con la comprensión cabal del funcionamiento del objeto de estudio, esto se logra utilizando la abstracción como herramienta para lograr la comprensión y mediante conceptos claves como lo son el encapsulamiento, podremos resaltar las cualidades relevantes de lo que queremos analizar.

De cierta forma lo que queremos lograr es capturar lo más fielmente posible, y con precisión matemática, un problema, para el que luego encontraremos una solución. Es importante tener en cuenta que en la etapa de especificación solo debe preocuparnos describir el problema que se intenta resolver, y no sus eventuales soluciones.

1.2. Sintaxis y semántica de un TAD

Todo lenguaje tiene una gramática o sintaxis, las cuales son un conjunto de reglas que indican cómo se escriben las oraciones del lenguaje y las reglas semánticas, que indican cómo deben interpretarse las oraciones válidas del lenguaje. Los lenguajes lógicos no son una excepción a estas.

Sintaxis

Para determinar un TAD, utilizaremos especificaciones de TADs (de la misma forma que se usan axiomas para caracterizar estructuras matemáticas). Para especificar un TAD es necesario definir:

- La **signatura** que exponen las operaciones y que aridad tienen los modelos.
- Los **axiomas** son formulas bien formadas según ciertas reglas (sintácticas) que determinan el comportamiento de las operaciones.

Semántica

La semántica declara con precisión qué cosas son modelos de nuestra teoría, es decir le da un significado a las cosas que se pueden escribir de acuerdo a las reglas sintácticas. Un modelo de nuestra teoría es un ente matemático tal que cumple que cada uno de sus conjuntos corresponden con un género del TAD y cada función con una operación, nosotros definiremos nuestro TAD de forma tal que acomodará los modelos matemáticos que se ajusten a el, de cierta forma la especificación del TAD funciona como un descriptor del modelo matemático. Un modelo matemático determina qué elementos del mundo real estarán reflejados en la especificación y que operaciones se permitirá realizar sobre ellos.

Por otro lado una teoría es consistente cuando en ella no existe una igualdad que afirme que **verdadero** es **falso**. Si introducimos una teoría inconsistente al TAD provocara que no haya un modelo que se ajuste al mismo, y por lo tanto, y burdamente dicho, provocará que el TAD no tenga sentido alguno ya que ningún modelo se ajustara a el y en consecuencia no seremos capaces de modelar nada.

Las instancias de un TAD son las que representan, mediante la abstracción, un objeto de la vida real y cualquier modelo que sea descripto por un TAD especifico representa a todas las instancias posibles del objeto modelado. Es interesante remarcar que cuando nos referimos a instancias podemos estar hablando del mismo objeto y su evolución con respecto al tiempo, esto también es modelable y podremos definir una instancia del objeto para cada instante de tiempo (o al menos para los cambios relevantes que queramos observar mediante el modelo).

1.3. Conceptos a tener en cuenta

1.3.1. Metalenguaje

Muchas veces al intentar describir propiedades acerca de un lenguaje formal no nos alcanza con dicho lenguaje, y es por ello que necesitamos de un metalenguaje para lograrlo. En las especificaciones de TADs, el metalenguaje es utilizado para escribir las restricciones de las funciones y para describir la igualdad observacional.

1.3.2. Restricciones sobre funciones totales

En el formalismo de los tipos abstractos de datos sólo se permite especificar funciones totales, es decir aquellas que están definidas para todo su dominio. Lo que técnicamente estamos haciendo no es restringir el dominio de las funciones sino definirlas sólo para la parte del dominio que nos interesa (por este motivo utilizaremos predicados del metalenguaje para restringirlas), o dicho de otra forma no diremos que valores toma la función cuando sus parámetros no cumplen con la restricción.

En consecuencia, cuando utilizamos una restricción estaremos subespecificando. Las restricciones son una parte fundamental del TAD, con ellas explicitamos los casos para los cuales ciertas operaciones no tienen sentido (por el marco del problema o por una limitación técnica) y aportan claridad y coherencia a una especificación, por ello en este caso estaremos haciendo un uso lícito de la subespecificación. De cierta forma las restricciones nos permiten limitar el universo al cual aplican ciertas operaciones de nuestros TADs.

Es importante remarcar el caso cuando utilizamos una función con su dominio restringido $g(x)$ dentro de la definición de otra función $h(x)$. Cuando sucede esto tendremos que asegurarnos de no llamar a g con parámetros de su dominio que estén restringidos. Para evitar esto y lograr que h este bien definida tendremos dos opciones. La primera consiste en agregar las restricciones necesarias a h para no llamar a g con parámetros restringidos, y la segunda, que es prácticamente lo mismo pero desde otra perspectiva, es manejar la parte conflictiva del dominio de h, es decir aquellos valores de los parámetros de h en donde llamaríamos a g de forma ilegal, mediante un condicional en h para efectuar otras operaciones o hacer un ajuste antes de llamar a g de forma tal de no producir ninguna violación. Dicho de otra forma esto ultimo seria como definir a h como una función partida.

1.3.3. Incertidumbre mediante subespecificación

Las restricciones nos servirán para expresar sobre que valores de los parámetros de una función determinada no daremos ninguna descripción, sin embargo es por ello que también deberemos asegurarnos de que si determinado valor de los parámetros de una operación es válido de acuerdo a las restricciones, este indicando mediante los axiomas, cuál es el resultado para aquellos valores. En algunos casos no diremos exactamente qué resultado da la función, si no que indicaremos qué características tiene ese resultado.

Esto último es un concepto sutil y avanzando que a veces también recibe el nombre de subespecificación y solo comparte el nombre con el descripto en al sección anterior. La intención que reside detrás de el es la de dejar algunos aspectos particulares del TAD sin una definición precisa, lo cual se convierte en un recurso muy útil para manejar algunas incertidumbres de forma práctica. La forma de lograr esto es caracterizando el resultado de una función de forma débil. Un ejemplo especifico de los tipos básicos es la operación **dameUno** del tipo **conjunto**, la cual lo único que nos dice es que nos devolverá un elemento perteneciente al conjunto sin especificar bajo que criterio lo hará. Esto significa que cualquier forma de elección que se decida implementar al conjunto va a ser apropiada, mientras que satisfagan la característica de elegir un elemento del conjunto.

1.3.4. No existe el orden de evaluación

En algunos lenguajes de programación una función puede estar definida en partes y las partes de la misma pueden estar ordenadas mediante un orden de evaluación. Esto de alguna forma simplifica el esquema de evaluación de las funciones, ya que en vez de usar condicionales en la función y sobre los datos, podremos usar múltiples definiciones a las que un dato se ajustara dependiendo si el pattern matching lo detecta como válido y en donde la evaluación terminará con la primer coincidencia, es decir con la primer definición en el orden de evaluación que sea válida para el dato. En estos casos las funciones se suelen ordenar desde los casos mas particulares hacia los casos mas generales.

En la especificación de los TADs la idea del pattern matching todavía estará latente, pero por otro lado el orden de evaluación no existirá. Esto sucede ya que todos los axiomas valen a la vez, no se evalúan en orden, y a causa de ello no deberemos definir un axioma para un caso particular de algún parámetro (como ejemplo, si es un numero cuando el mismo vale cero).

1.4. Estructura de un TAD

Se explicaran a continuación cada uno de los componentes que conforman un tipo abstracto de datos.

1.4.1. Igualdad observacional o Semántica observacional

Definición

La igualdad observacional es un predicado que nos dice cuándo dos instancias del aspecto de la vida real que nuestro TAD está modelando se comportan de la misma manera. El concepto de la igualdad observacional es un concepto semántico (es decir que le da sentido al TAD) y no sintáctico, por lo que es necesario utilizar el metalenguaje para describirla.

Semántica inicial

Para los TADS, una semántica posible es la semántica inicial, que burdamente descripta trata de partir el universo de acuerdo a las ecuaciones que aparecen en el TAD, de esta forma la relación de igualdad que queda definida es una congruencia (significa que las instancias del tipo están en la misma clase de equivalencia y pertenecerán a la misma sin importar que función se les aplique). La desventaja de esto último es que los modelos resultantes no son muy bonitos.

Semántica observacional

A causa de la incomodidad de la semántica inicial se invento la semántica observacional, en la cual hay un conjunto de funciones etiquetadas como observadores básicos que particionan el universo de instancias de acuerdo a ellos. La intención es que estas particiones sean congruencias por lo que no puede haber inconsistencias entre declarar una igualdad entre dos instancias y la información (distinta) que nos puede brindar alguna de sus funciones acerca de ellas.

1.4.2. Observadores básicos

Definición

Los observadores básicos son un conjunto de funciones pertenecientes al TAD que permiten particionar el universo de sus instancias en clases de equivalencia, con la idea de agrupar en cada clase a las instancias que posean un comportamiento similar con respecto al estudio que queremos realizar. Deseamos que el TAD se convierta en una congruencia, es decir, una relación de equivalencia en la que si se aplica cualquier operación a dos instancias de la misma clase los resultados obtenidos formen parte de la misma clase.

Ruptura de congruencia

Si comparamos instancias observacionalmente iguales no debería pasar que al aplicar un observador a ambas obtengamos resultados observacionalmente distintos, de la misma forma que si tenemos instancias observacionalmente distintas no podrá pasar que al aplicar todos los observadores a ambas obtengamos los mismos resultados. En ambos casos el TAD no se comportaría como una congruencia (solamente seria una clase de equivalencia dada por la igualdad observacional).

Minimalismo y axiomatización de observadores

Cuando realizamos la selección de funciones del TAD para agruparlas como observadores es preferible tener un conjunto de observadores minimal, esto es que no deberían existir observadores que sólo identifiquen aspectos de la instancia que ya han sido identificados por otros observadores. Además, es considerado buena práctica axiomatizar los observadores básicos en función de los generadores y no de otros observadores. La axiomatización utilizando otros observadores se reserva, en la práctica, para **otras operaciones**. Notar que, salvo en los casos en los que la operación tenga una restricción con respecto a alguna instancia, la cantidad de axiomas que tendremos será aproximadamente el cardinal del producto cartesiano entre la cantidad de observadores básicos y los generadores.

Sobreespecificación en los observadores

Decimos que una operación esta sobreespecificada cuando hay varias formas de saber cuál es su resultado para unos valores dados de sus parámetros. Si bien esto puede estar definido en forma legal y no romper con el modelo, puede resultar un poco confuso a la hora de conocer un resultado ya que pueden haber varios caminos para obtenerlo. El problema se presenta cuando obtenemos resultados distintos dependiendo si seguimos caminos distintos

1.4.3. Generadores

Definición

Los generadores son un conjunto de funciones que retornan un resultado del género principal del TAD especificado, y que tienen la particularidad de que a partir de una aplicación finita de ellos se pueden generar o construir absolutamente todas las instancias del TAD. Esto es, que no puede existir una instancia del problema que estamos modelando que sea relevante y que no podamos generar su representación a partir de una sucesión de aplicación de los generadores del TAD.

Estructura de generadores

El conjunto de generadores puede ser clasificado de la siguiente manera:

- **Generadores base o no recursivos** son aquellos que no reciben como parámetro ninguna instancia del tipo que están generando, es decir, serán usados como base de los generadores recursivos.
- **Generadores recursivos** son aquellos que reciben como parámetro al menos una instancia del tipo que están generando, esto es, un generador base o una aplicación de un generador recursivo a otra/s instancia/s del TAD (que bien esta misma puede ser una sucesión de aplicaciones de generadores).

Además de recibir como parámetro a instancias del tipo que generan, los generadores pueden recibir como parámetro otros tipos que usaran como información de la instancia (por ejemplo números o strings).

Transparencia referencial

Es importante notar que al aplicar un generador recursivo a una instancia de un TAD no se está modificando la instancia que recibe como parámetro dado que en nuestro lenguaje no existe la noción de "cambio de estado", por lo que realmente se estará haciendo será generar una nueva instancia basada en la anterior, cuyo comportamiento podrá ser descripto mediante la aplicación de los observadores básicos sobre ella. Es definitiva, los resultados de las funciones sólo dependen de sus argumentos.

Importancia en la estructura de los generadores / Inducción estructural

Dado que todas las instancias de un TAD están generadas a partir de un generador base o a partir de la aplicación de un generador recursivo, se vuelve un pilar fundamental a la hora de realizar demostraciones de propiedades sobre los tipos abstractos de datos, ya que nos ofrece un esquema de demostración dividido en dos partes mapeable a una esquema de inducción; en donde la primer parte demostrara la propiedad para todas las instancias generadas por generadores base y la segunda demostrara la propiedad para todas las instancias generadas por generadores recursivos. Este esquema de inducción es conocido como **inducción estructural**.

1.4.4. Otras operaciones

En esta categoría estarán el resto de las operaciones que se necesiten declarar en un TAD incluyendo las operaciones auxiliares que no se exportan. La diferencia primordial entre las operaciones que se encuentren en esta categoría y las operaciones encontradas en la categoría de observadores básicos, es que las operaciones en esta sección no deberán devolver valores distintos cuando se apliquen sobre dos instancias observacionalmente iguales del TAD. Dicho de otra forma, no deberán dar información del TAD que no este cubierta por los observadores básicos, de lo contrario la congruencia del mismo sera imposibilitada.

1.4.5. Géneros, Usa y Exporta

En la sección de géneros se incluirán todos los géneros nuevos que se describen en el TAD. El género es el nombre colectivo con el que se va a hacer referencia a instancias del TAD que estamos definiendo, el cual es diferente al Tipo del TAD. Un tipo es el conjunto de operaciones, axiomas y demas que componen al TAD. En la sección de usa se incluyen los nombres de los TADs que necesitaremos para definir el nuevo tipo, desde el punto de vista formal lo que estamos haciendo es incluir otras teorías en la que estamos definiendo. Por último, la sección exporta servirá para incluir todos los elementos declarados en el TAD que queremos que puedan ser utilizados por otros TADs, por defecto se exportaran los **generadores** y **observadores básicos**.

1.5. Al especificar recordar

Estas son una serie de consideraciones a tener en cuenta en el momento de especificar. Algunas de ellas son buenas prácticas, otras son ideas de formas que deberemos, o es recomendable, hacer ciertas cosas y otras están escritas para remarcar en que cosas no deberemos caer.

1.5.1. No axiomatizar sobre casos restringidos

A la hora de axiomatizar una función con restricciones no se ha de realizar ningún tipo de consideración para "controlar" que los argumentos cumplan efectivamente las restricciones ya que cuando la función es usada todos los argumentos siempre cumplen las restricciones que impusimos, y es por ello que asi debe considerarse cuando los axiomatizamos.

1.5.2. No axiomatizar sobre generadores de otros tipos

Si bien no es algo que este completamente mal como en el punto anterior, la axiomatización de operaciones sobre generadores de otros tipos puede ocasionar que la igualdad observacional del tipo usado sea violada, algo que nunca se podrá dar si en su lugar utilizamos los observadores básicos. Es por ello que es preferible que al realizar las axiomatizaciones se efectúen en función de los observadores del tipo usado y no sobre los generadores.

1.5.3. 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 (si es que no nos interesa conocer nada de ella puntualmente) para modelar correctamente el objeto de estudio. Muchas veces podremos tener cadenas de condiciones - acciones - consecuencias en donde las acciones y consecuencias se den de forma automática y la consecuencia de una sea la condición de otra cadena de este tipo. En estos casos solamente hara falta modelar lo suficiente para saber cuando se cumple la primera condición de la cadena y en base de eso podremos definir alguna operación que modele solamente la ultima consecuencia de la satisfacción de la condición, pasando por alto todas aquellas acciones o consecuencias de la vida real que ocurren en el medio y no nos interesa modelar.

1.5.4. Recursión y recursión mutua

La idea detrás de una definición recursiva es ir simplificando la instancia hasta el punto en donde no se puede simplificar más, el llamado caso base. Allí el axioma se resuelve directamente sin usar el concepto que está definiendo, lo importante es que para resolver el caso base nos basta con saber qué tiene que devolver la función para ese caso particular, lo cual es relativamente sencillo.

La auto-referencia a la definición que se está dando se realiza en el caso recursivo, donde se descompone el objeto sobre el cual se esta definiendo y se aplica la definición a esas simplificaciones del mismo. En nuestras definiciones recursivas deberemos garantizar que eventualmente se llegará al caso base para todos los valores sobre los que se encuentra definida (mejor dicho, no restringida) la operación. La manera mas habitual de garantizar esto es que en cada caso recursivo se logre disminuir la complejidad de los parámetros involucrados.

La auto-referencia a las definiciones puede darse de manera indirecta. Lo que encontramos en estos casos es recursión mutua; donde una definición no hace auto-referencia directamente sino que lo hace a través de otra definición. La recursión mutua puede darse en más de dos niveles. En estos casos las consideraciones respecto de la disminución de la complejidad se verán un poco mas complicadas pero seguirán vigentes. Cuando planteamos una recursión debemos concentrarnos en resolver cada caso particular correctamente sin preocuparnos por los otros. Luego si cada caso se resuelve bien, el conjunto también sera correcto.

1.5.5. Interfaces gruesas

Se define como interfaz gruesa a la situación que se da cuando se proveen mas datos que los necesarios en una determinada función. Un indicador de que estamos cayendo en esto es el uso excesivo de los observadores dentro de la axiomatización de una función. Es decir, si no utilizamos toda la instancia que tenemos, sino que proyectamos sistemáticamente una de las características de la instancia, vale preguntarse si no correspondería tener solo esa característica en primer lugar.

1.6. Inducción estructural

La inducción estructural nos servirá para demostrar teoremas o propiedades sobre nuestros tipos mediante el uso de sus axiomas y la inducción como herramienta para demostrar su validez para un dominio coordinable con todo $\mathbb{N}$. Para hacerlo se podrán seguir una serie de pasos:

- **Convencernos que es cierto** Si bien no es un paso realmente necesario, es importante para nosotros. Si la propiedad es cierta a simple vista entonces no tendremos problemas en buscar una demostración a la misma, pero cuando su veracidad no es tan fácilmente visible es importante darnos cuenta de porque vale, ya que de lo contrario tendremos problemas al demostrarlo o al creer que la demostración es correcta.
- **Plantear la propiedad como predicado unario** Básicamente esto consiste en quitar el cuantificador que liga a la variable sobre la que vamos a realizar inducción. Por ejemplo si tenemos algo como $(\forall s : secu(\alpha))$ $(Long(Duplicar(s)) = 2 \cdot Long(s))$ el predicado unario resultante seria $P(s) \equiv (Long(Duplicar(s)) = 2 \cdot Long(s))$, de forma tal que la expresión inicial nos quedaría $(\forall s : secu(\alpha))P(s)$, que seria equivalente.
- **Plantear el esquema de inducción** El esquema de inducción consiste en plantear los casos base que debemos probar así como los pasos inductivos. Este esquema es propio del tipo, ya que se deriva de su conjunto de generadores, por lo tanto para cualquier propiedad que se quiera probar sobre un TAD dado, el esquema de inducción será el mismo. Para ejemplo anterior, el esquema de inducción quedaría de la forma $(\forall s : secu(\alpha))P(s) \implies (\forall a : \alpha)P(a \bullet s)$ en donde $P(s)$ es la hipótesis inductiva y $(\forall a : \alpha)P(a \bullet s)$ es la tesis inductiva.
- **Demostración** Para probar la validez de la propiedad probaremos primero el caso base para cada uno de los generadores base y luego el paso inductivo con cada uno de los generadores recursivos, tal como lo sugiere el esquema de inducción.

1.6.1. Fundamento teórico

La inducción completa es una instancia particular de la inducción estructural, es decir que la inducción estructural es una generalización de la inducción completa para otros tipos de datos mas allá de los números naturales. La inducción estructural tiene su fundamento teórico sobre el principio de inducción bien fundada, para el cual es necesario previamente un orden bien fundado.

- **Orden bien fundado** Decimos que $\prec$ define un buen orden sobre un conjunto A (o equivalentemente, que tiene un buen orden fundado), sii $\prec$ es un orden total sobre A y todo $X \subseteq A$ tal que $X \neq \emptyset$ tiene un elemento que es mínimo de acuerdo a $\prec$. Es decir, si hay un orden total definido sobre A y además todo subconjunto de A tiene un mínimo, entonces tendremos un buen orden fundado. De cierta forma se pide que el orden total definido sea consistente.
- **Orden total** Decimos que $\prec$ define un orden total sobre el conjunto A, sii define un orden parcial y además tiene comparabilidad (o tricotomía). Esto ultimo es $\forall a, b \in A$ se cumple que $a \prec b \vee b \prec a$. Por ejemplo $\leq$ es un orden total en N.
- **Orden parcial** Decimos que $\prec$ define un orden parcial sobre un conjunto A, sii $\prec$ es una relación reflexiva, antisimétrica y transitiva. Si se quita la reflexividad se habla de un orden parcial débil. Por ejemplo $<$ es un orden parcial débil en N.

Construcción de un orden bien fundado

Si los elementos de un conjunto son numerables podremos realizar una construcción de un orden bien fundado realizando un mapeo con los números naturales. Como las instancias de cualquier TAD son numerables, siendo T las instancias del TAD que estamos mapeando y N el conjunto de naturales definiremos la función $f : T \to N$ tal que $x \prec_f y \iff f(x) \leq f(y)$. De esta forma $\prec_f$ sera el orden bien fundado sobre T. En el caso particular de los TADs sabemos que los mismos son definidos de forma inductiva. Por ello, f puede ser definida de la siguiente forma:

- Si x es un elemento base de T, entonces $f(x) = 0$
- Si x se construye a partir de los elementos $x_1, ..., x_n$, entonces $f(x) = 1 + \text{máx}(f(x_1), ..., f(x_n))$

Principio de inducción bien fundada

Una vez definido un orden bien fundado sobre el conjunto podemos hablar del **principio de inducción bien fundada**. Para tener esto ultimo $\prec$ debe definir un buen orden sobre el conjunto A, P debe ser un predicado sobre A y P debe cumplir

1. P debe valer para todos los elementos mínimos de A de acuerdo a $\prec$, es decir P debe valer para todos los elementos base.
2. Se debe cumplir que $(\forall a \in A)[(\forall b \in A | b \prec a) P(b) \implies P(a)]$. Es decir, que para todo $a \in A$, cuando vale $P(b)$ para todos los $b \in A | b \prec a$, entonces vale $P(a)$.

entonces $(\forall\ a \in A)\ P(a)$

Demostración:

- Supongamos que $\prec$ define un orden bien fundado sobre A, que P cumple (1) y (2) pero que no vale para todos los elementos de A.
- Como $\prec$ es un orden bien fundado, el conjunto $\{a \in A | \neg P(a)\}$ tiene un elemento mńimo, que llamaremos m.
- Si m es un mínimo para A, entonces contradice (1).
- Si no lo es, entonces tiene predecesores. Como m era el mínimo elemento que no cumplía P, todos sus predecesores sí lo cumplen. Pero eso contradice (2). $\square$

Esquema de inducción estructural

- Llamaremos $g_1, ..., g_k$ a los generadores del tipo T que no toman como parámetro una instancia de T, es decir que estos serán los generadores base.
- Llamaremos $g_{k+1}, ..., g_n$ a los que si toman una instancia de T, es decir que estos serán los generadores recursivos.
- El primer paso para la inducción es probar el caso base, es decir $P(g_1) \wedge ... \wedge P(g_k)$ debe ser verdadero.
- Luego probaremos el paso inductivo, esto es $(\forall i : T)[P(i) \implies P(g_{k+1}(i))] \wedge ... \wedge (\forall i : T)[P(i) \implies P(g_n(i))]$. Esto es, para cada uno de los generadores recursivos, pruebo el paso inductivo con todas las instancias posibles como precedente de forma tal de obtener todas sus posibles variantes (por simplificar no se incluyo la variación de los argumentos de los generadores). De esto podremos concluir que $(\forall i : T) P(i)$.

Capítulo 2

Diseño jerárquico de tipos de datos abstractos

2.1. Introducción y noción

En la etapa de especificación de problemas, lo único que hemos hecho es detallar qué debemos hacer, pero no nos hemos preocupado por cómo hacerlo, es decir que el objetivo era describir el comportamiento del problema a resolver, pero no interesaba determinar cómo lo resolveríamos. Esto significa que al especificar estamos describiendo el problema, recién al diseñar comenzamos a resolverlo.

2.1.1. Diseño

Al diseñar, centraremos nuestro interés tanto en el ámbito en el que será usado el tipo abstracto de datos como en los aspectos que se necesitan optimizar de este tipo, los cuales pueden estar dados por requerimientos explícitos de eficiencia temporal o espacial. Sobre la base de esta información, a la que llamaremos **contexto de uso**, diseñaremos nuestro tipo aprovechando las ventajas que el contexto nos ofrezca y cuidando de responder a los requisitos que nos plantea. Es importante remarcar que un tipo se define por sus funciones antes que por sus valores. La forma en que los valores se representan es menos importante que las funciones que proveen para manipularlos. Los generadores de los tipos describen la forma abstracta de construir elementos, nunca la forma de construirlos o representarlos físicamente.

En esta etapa, al buscarle representaciones menos abstractas al modelo especificado, es donde realmente comenzaremos a aprovechar el nivel de abstracción. Cuanto más abstracto sea el modelo, mas opciones de diseño tendremos disponibles en cada paso. Básicamente nuestra metodología de diseño partirá entonces de un modelo abstracto no implementable directamente en un lenguaje imperativo de programación, y aplicará iterativamente sobre dicho modelo sucesivos pasos de refinamiento hasta llegar a estructuras que si son implementables. En cada una de estas sucesivas iteraciones estaremos realizando, de cierta forma, una desabstracción.

Es importante tener en cuenta que en la especificación estaremos centrados en un paradigma funcional y en la etapa de diseño nos centraremos en paradigma imperativo, por lo que tendremos un cambio de paradigma además de la desabstracción del modelo, lo que nos conllevara ciertas dificultades. Uno de los objetivos del lenguaje de diseño es justamente permitir un cambio de paradigma que resulte ordenado.

2.1.2. Jerárquico

Cada iteración de desabstracción de este proceso definirá un nivel de nuestro diseño. Por su parte, cada uno de estos niveles tendrá asociado uno o más módulos de abstracción, que indicaran cómo se resuelven las operaciones de un módulo utilizando otras operaciones de módulos del nivel inmediato inferior. Cada uno de estos módulos de abstracción resultantes de cada iteración, será implementable en un lenguaje de programación, obteniendo de tal forma un diseño estratificado en niveles donde los módulos de un cierto nivel son usuarios de los servicios que les brindan los del nivel inmediato inferior y no conocen (ni usan) a los módulos de otros niveles. Un módulo dará a conocer los servicios que provee a través de una declaración de las operaciones que exporte junto con la aridad de cada una de ellas, se darán a conocer las precondiciones (estado esperado de la máquina antes de ejecutarse la operación) y las postcondiciones (como incidirá la ejecución en el estado anterior). Esta información estará incluida en la interfaz del módulo.

Esta separación o encapsulamiento en niveles provocara que cualquier cambio de implementación de nivel n será transparente al nivel superior $n+1$, siempre que el nivel n mantenga su interfaz. Esto es, que el módulo exporte al menos las mismas funciones que se exportaban antes y la funcionalidad provista por las mismas no cambié, aunque puede haber mejorado su performance. Podremos verificar la validez del cambio de diseño viendo que la veracidad de las precondiciones y postcondiciones del nivel rediseñado se mantiene con respecto a la versión anterior.

2.2. Lenguaje de diseño

Este es el lenguaje que utilizaremos para diseñar nuestros módulos. El mismo, si bien sera un pseudocódigo, estará muy basado en lenguajes que se utilizan en la vida real, de esta forma la transición desde el diseño a los mismos será casi inmediata.

2.2.1. Paradigma imperativo

Para especificar formalmente el problema a resolver escribíamos el tipo abstracto de datos siguiendo un paradigma funcional. Sin embargo al diseñar, debemos realizar un cambio de paradigma para poder expresar nuestra representación del modelo en un lenguaje imperativo, el cual se ajusta a los lenguajes de programación mas usados. En esta sección discutiremos los principales aspectos que deberemos tener en cuenta al afrontar tal cambio.

Valores vs. Objetos

Las aridades de las operaciones que definimos en la especificación para los tipos están en una notación funcional, esto quiere decir que supone que las mismas construyen un objeto nuevo y lo devuelven a aquel que las llamo. Una característica de esta notación es la transparencia referencial, esto es que una expresión siempre da el mismo resultado sin importar su contexto. En este paradigma los datos sólo tienen sentido en cuanto sean argumentos o resultados de funciones.

Al contrario del paradigma funcional los datos en el paradigma imperativo son tratados como entidades independientes de las funciones que los utilizan. Es usual que se trabaje con una instancia de un objeto que se va modificando y cuyos valores anteriores no interesen. Por lo tanto, por cuestiones de optimización y uso, no tiene sentido construir cada vez un objeto nuevo para devolverlo como resultado de una función, sino que en cambio se modificara el objeto original.

Parámetros que se modifican

El mapeo de los parámetros de las funciones del tipo en las operaciones del módulo no siempre es uno a uno. De hecho en el paradigma imperativo se acostumbra a modificar los parámetros como parte de respuesta del algoritmo y a devolver en el valor de retorno de la función, algún estado que informe si la operación se completo de forma correcta o si hubo algún problema. Esto nos brinda una mayor versatilidad a la hora de diseñar las interfaces, ya que una misma función puede devolver varios tipos de resultados sin necesidad de hacerlo mediante una tupla y además dar alguna información acerca de la ejecución.

2.2.2. Tipos disponibles

Los tipos listados a continuación serán considerados tipos básicos y no serán diseñados: bool, nat, int, real, char, string, género, puntero$\langle tipo_dato \rangle$, arreglo[$nat$] de $tipo\_dato$, tupla$\langle campo_1\ tipo_dato \times ... \times campo_n\ tipo_dato \rangle$, arreglo_dimensionable de $tipo\_dato$.

2.2.3. Declaración de operaciones y pasaje de parámetros

Para declarar las operaciones se le asignara un nombre, se describirán los argumentos y los tipos de cada uno como así también el tipo de dato del valor de retorno de la función. El pasaje de parámetros puede pertenecer a 3 tipos:

- **entrada** El valor es usado como dato pero no es posible modificarlo (en C++ la equivalencia seria el const). Se denota anteponiendo al nombre de la variable en el tipo de la operación el símbolo "**in**".
- **salida** El valor se genera en la operación, y se almacena en el parámetro formal con el que se invocó a la función pero no es usado como dato. Se lo denota con "**out**". La variable resultado (la que devuelve la función) pertenece a esta categoría.
- **entrada-salida** Combina los conceptos anteriores, se lo denota con "**in**/**out**".

Recordemos que en el paradigma imperativo todos los valores son pasados por referencia, exceptuando a los tipos primitivos (bool, nat, int, real, char, puntero) que son pasados por valor o copia*. Al ser pasados por referencia y al ser del tipo de entrada-salida o salida y efectuar una asignación sobre el mismo, el valor con el que fue llamada la función es sobreescrito y el nuevo valor será el que mantendrá la variable luego de salir de la función, esto quiere decir que la variable que pasamos como parámetro de la función es efectivamente modificada por mas que nos encontremos dentro del scope de la función.

***** En el caso de los arreglos dimensionables y estáticos, se pasan por referencia, y en el caso de las tuplas, cada una de sus componentes se pasa por referencia o por copia según sea un tipo primitivo o no.

2.2.4. Asignación y aliasing

La expresión $A \leftarrow B$ (siendo A y B variables de un mismo tipo), denota la asignación del valor B a la variable A. Esto funciona del mismo modo que el pasaje de parámetros. Si A y B pertenecen a un tipo primitivo A pasara a ser copia de B, y si no son tipos primitivos, luego de haber efectuado la asignación, A y B harán referencia a la misma estructura física, es decir A sera un alias de B y viceversa (de ahí el nombre).

2.2.5. Relación entre el diseño y la especificación, precondiciones y postcondiciones

Al describir la interfaz de un módulo para cada una de las operaciones deberemos indicar cuáles son sus restricciones y qué efectos produce en el estado de la máquina. Para describir esto haremos uso de la especificación del tipo abstracto de datos asociado al módulo, lo que en consecuencia nos obligará a describir la relación que existe entre las variables del diseño y el tipo abstracto de datos especificado. Queremos poder describir en la interfaz cual es el resultado final luego de aplicada una operación teniendo en cuenta los parámetros con los cuales se la invoca y las relaciones entre ellos.

Cuando nos pasa esto tenemos el problema de que los tipos y operaciones a las que queremos hacer referencia están en el mundo de los TADs y por otro lado, las funciones sobre las que queremos describir su entrada y resultado están en el mundo del diseño. Por lo que de cierta forma queremos comparar elementos que no están definidos en base a axiomas, con elementos que si lo están. Para subsanar esta dificultad, existirá la función $\widehat{\bullet}$.

Llamaremos G_I al conjunto de géneros del paradigma imperativo y G_F al conjunto de géneros del paradigma funcional. Sub-indexaremos con I a los géneros de G_I y con F a los de G_F. Es decir, disponemos de una función que dado un género del paradigma imperativo nos da su "equivalente" en el paradigma funcional, la misma quedara definida como:

$$\widehat{\bullet} : G_I \rightarrow G_F$$

Mediante el uso de esta función podremos establecer de forma clara el mapeo entre las operaciones del módulo y las funciones de la especificación cuando escribamos las precondiciones y postcondiciones de cada operación del módulo.

2.3. Metodología de diseño

Nuestro objetivo es obtener un diseño jerárquico y modular. Para realizar esto hay varias formas pero presentaremos un método que tiene las nociones de los distintos niveles en la jerarquía. Cada uno de los niveles tendrá asociado un módulo de abstracción. Para ser más precisos, habrá distintos tipos abstractos de datos que deberemos diseñar, a cada uno de ellos le corresponderá un módulo de abstracción. A grandes rasgos, el método se compone de los siguientes pasos:

- Elección del tipo abstracto de datos a diseñar.
- Implementación del módulo de abstracción para el tipo abstracto de datos elegido.
- Iteración o finalización.

2.3.1. Elección del tipo a diseñar

El orden en el cual se diseñan los tipos es arbitrario. Sin embargo es una buena práctica comenzar por los tipos más importantes, pues éstos serán los principales generadores de requerimientos de eficiencia para los tipos menos importantes o de niveles inferiores. De cierta forma estamos aplicando el esquema top-down. Es importante notar que el proceso de diseño posee una natural ida y vuelta. Por ejemplo, la redefinición de las funciones de un tipo puede obligarnos a reveer la sección representación de un tipo que basa su diseño en éste. Si lo vemos desde el punto de vista explícitamente jerárquico, la redefinición de las operaciones de un tipo de un nivel, puede obligarnos a redefinir a un modulo de un nivel superior.

Cuando diseñamos un módulo, no necesariamente debemos diseñar todos los tipos que usamos en la especificación. Esto quiere decir que a veces sucede que en la especificación realizamos ciertas tareas utilizando tipos que en la etapa de diseño, al no ser el objetivo la descripción del problema y serlo la resolución del mismo, podemos no necesitar. Es decir que si tenemos un tipo que no se exporta directamente y solamente necesitamos algunas de sus operaciones, podemos quizás evitar usarlo y reemplazarlo por algo mas ligero que cumpla nuestros requerimientos. Esto expresa que la manera en que se axiomatizan los tipos en el TAD no es importante a la hora de realizar el diseño, sino que solamente es importante lo que dichos axiomas significan.

2.3.2. Implementación del módulo de abstracción para el tipo elegido

Una vez elegido el tipo a diseñar, crearemos su módulo de abstracción correspondiente. El módulo de abstracción deberá describir de forma clara y concisa las operaciones que podrá realizar, como así también los efectos de las mismas sobre los datos, las complejidades temporales y espaciales que tomarán su uso y cuales de las operaciones podrán ser utilizadas externamente, es decir, cuales se exportarán. Además de dar una descripción externa del mismo poseerá otra sección en la cual se explicitará la implementación de cada una de las operaciones (incluyendo operaciones privadas auxiliares) como así también la estructura de datos interna utilizada para llevar al cabo las tareas.

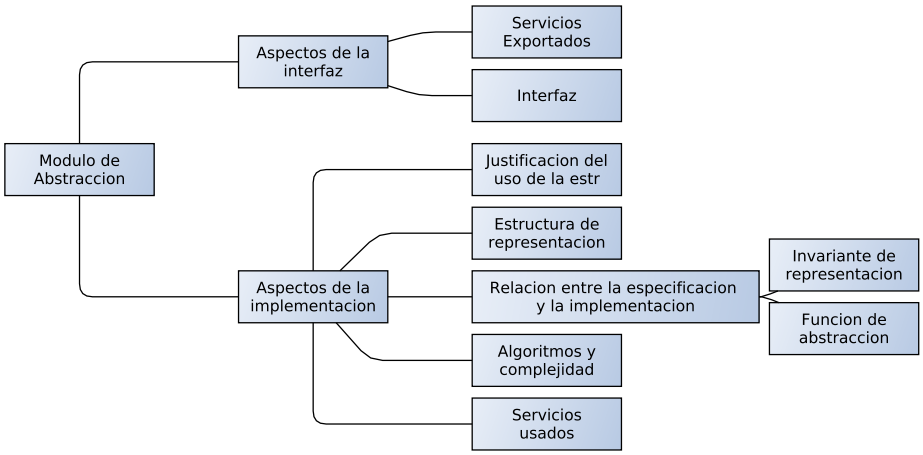
2.3.3. Iteración o finalización.

En este punto tenemos un diseño que puede contener tipos para los que no tenemos una propuesta de diseño. En realidad son otros problemas a resolver de nivel de abstracción menor al original. Por lo tanto, debemos volver a repetir el método con los nuevos tipos a diseñar.

La iteración prosigue hasta llegar a tipos que tengamos diseñados en nuestra biblioteca o sean primitivos. Por otro lado, la reutilización de tipos ahorra tiempo de diseño pero ya que es posible reutilizar tipos que fueron diseñados con criterios distintos a los que deseamos, podríamos perder parte de la eficiencia buscada lo que será tolerable siempre y cuando no rompa las restricciones planteadas por el contexto de uso.

2.4. Módulo de abstracción

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. En la **representación** se elige, bajo algún criterio, una forma de representación utilizando otros módulos y se resuelven las operaciones del módulo en función de su representación. Dicho de otra forma, se eligen las estructuras de datos internas que representarán el modulo y se implementan los algoritmos que la interfaz expresa. Un módulo de diseño debe tener la siguiente estructura:



- **Especificación** Puede omitirse si es uno de los TADs provistos por la cátedra, o incluirse sólo los cambios si es una extensión de un TAD ya conocido.
- **Interfaz** La cual contendrá los servicios exportados, órdenes de complejidad, aspectos de aliasing, efectos secundarios, etc. En definitiva, todo lo que el usuario necesite saber.
- **Representación** Estructura interna, invariante, función de abstracción y algoritmos.
- **Servicios usados** Órdenes de complejidad, aspectos de aliasing, etc. requeridos de los tipos de soporte que utilizamos en nuestro módulo.

2.4.1. Aspectos de la Interfaz

En este paso tomamos las decisiones concernientes al cambio de paradigma. Una forma de lograr esto es redefinir las aridades de las funciones adaptándolas a un lenguaje imperativo explicitando los requerimientos (precondiciones) para la aplicación de cada operación y los efectos que tiene sobre el estado de la máquina (postcondiciones). Para escribir las precondiciones y las postcondiciones usaremos un lenguaje de descripción de estados aprovechando la especificación del tipo a diseñar.

Durante la redefinición de aridades no debemos limitarnos a cambiar una operación por un procedimiento como así también a respetar exactamente la cantidad de parámetros que tiene una operación. Siempre podremos decidir usar más de un procedimiento para reemplazar a una operación del tipo abstracto como así también, y contrariamente, tendremos la posibilidad de unir varias funciones del tipo abstracto en una sola del diseño. Además de poder evitar totalmente el mapeo uno a uno podremos incluir funciones que no tengan sentido desde el punto de vista abstracto pero sean útiles dentro del paradigma imperativo, por ejemplo funciones para copiar instancias del tipo o funciones como "comprimir" cuyos efectos sean visibles solo desde la eficiencia temporal y espacial de las operaciones, pero no signifiquen un cambio en el valor representado. Todas estas decisiones dependen del contexto de uso.

Servicios exportados / Operaciones exportadas

En esta sección deben estar expresados los detalles acerca de nuestro módulo que resulten indispensables a sus usuarios. Es imprescindible tocar temas como la complejidad temporal de los algoritmos y cuestiones de aliasing y efectos secundarios de las operaciones. Además, pueden exhibirse comentarios (a modo informativo) con respecto a la implementación del módulo que, aunque tengan menor importancia, sean de interés para el usuario. Los puntos a recalcar que debe tocar esta área son los siguientes:

- Deducir y documentar la **complejidad temporal** de cada operación es fundamental dentro de esta area, ya que de no haber información de la misma nuestro módulo no podría ser utilizado por ningún otro módulo cuyo contexto de uso restringiera tal aspecto. Dicho de otra forma, un usuario que deba responder a requerimientos de eficiencia temporal y deba usar nuestro módulo, no podrá saber si esta cumpliendo con ellos si no documentamos los mismos.
- Conocimiento del **aliasing** es de vital importancia para el uso correcto y eficiente del módulo. Si no informásemos las cuestiones de aliasing referentes a las operaciones podría pasar que un usuario de nuestro módulo modifique datos sin saberlo por estar utilizando algún alias (en vez de una copia) generado a partir del uso de de una de nuestras operaciones, lo que provocaría un problema muy grave en el desarrollo del programa.
- Es importante indicar en las operaciones que **eliminan elementos de la estructura** si dichos elementos seguirán existiendo o serán eliminados. Esto afecta a los usuarios que tengan referencias a dichos elementos. Si se los elimina, las referencias a ellos dejarán de ser válidas, en caso contrario seguirán vigentes pero ya no estarán en la estructura, por lo que será el usuario el encargado de liberarlas al momento de implementar su módulo ya que de no hacerlo podrá producir una potencial perdida de memoria.

Contexto de uso y requerimientos de eficiencia de los servicios prestados

Como dijimos al inicio, el contexto de uso está dado por el entorno en donde vaya a ser usado el programa del cual se derivarán, a través de un análisis de las funciones que serán más usadas, los requerimientos de eficiencia y además qué estructuras de datos se ajustan mejor al contexto en el que se encuentra el diseño. No se puede conocer si un diseño es adecuado si no se aclara precisamente el contexto de uso. La idea es que la principal justificación para el resultado obtenido en cada iteración de diseño es el contexto de uso que se le impuso al diseñador.

2.4.2. Aspectos de la Implementación

El objetivo de esta sección es definir la forma en que representaremos el tipo que estamos diseñando en esta iteración (o nivel). La elección de una forma de representación está dada por la elección de una o más estructuras, las cuales deberán estar debidamente justificadas. Además de elegir la estructura de representación, es necesario definir cuál es la relación entre la estructura de representación y el tipo representado. Por último, se deberán proveer los algoritmos que operan sobre la estructura y que resuelven cada una de las operaciones.

La estructura de representación de las instancias de los tipos sólo será accesible (modificable, consultable) a través de las operaciones que se hayan detallado en la interfaz del módulo de abstracción respectivo. Las operaciones no exportadas también tendrán acceso a esta información, pero sólo podrán ser invocadas desde operaciones del mismo módulo, es decir la visibilidad de las mismas fuera del módulo será nula.

Elección de la estructura de representación

La elección de la estructura esta fundamentada sobre las operaciones que nos interesa optimizar y el contexto de uso en el que serán utilizadas. No solo tomamos en cuenta la complejidad temporal para definir un criterio de optimización sino que también tomamos en cuenta el espacio de disco, espacio en memoria, reusabilidad, claridad, sencillez de la implementación, homogeneidad de los algoritmos, entre otros.

Las variables en un programa referencian valores. Será imposible el acceso a la representación interna de éstos y esto redundará en la modularidad de nuestro diseño y en el ocultamiento de la información. El ocultamiento nos permite hacer invisibles algunos aspectos que serán encapsulados. Esto es útil para aumentar el nivel de abstracción y diseñar código que sea más fácilmente modificable, mantenible y extensible. Al acceder a los objetos sólo a través de su interfaz no nos atamos a su implementación, sólo a su funcionalidad. Cualquier cambio de implementación en un tipo que no altere la funcionalidad no nos obligará a rediseñar los tipos superiores.

Relación entre la implementación y la abstracción

Para poder relacionar el mundo de los TADs con el mundo del diseño haremos uso de dos funciones que nos facilitaran esta tarea. Ambas funciones tendrán su dominio en las instancias de representación, estas son todas las formas que tendremos de instanciar la estructura de datos que elegimos en la representación sin restricción alguna. Como podremos imaginar, muchas de estas formas de instancias la representación no tendrán sentido alguno para el tipo que estamos representando, mas que nada si las variables de nuestra representación tienen alguna correlación (por ejemplo, cantidad de elementos y la lista con dichos elementos), es aquí en donde **invariante de representación** cobrará importancia, ya que es el que nos indicara que instancias de la representación tienen sentido para el tipo abstracto que estamos representando y cuales no. Luego, la **función de abstracción** servirá para llevar una instancia de representación al mundo de los TADs, es decir a una instancia del tipo abstracto de datos que estamos representando.

Invariante de representación

$$Rep : \text{genero\_de}\widehat{\text{\_representacion}} \rightarrow boolean$$

El invariante de representación es un predicado que nos indica si una instancia de la estructura de representación es válida para representar una instancia del tipo representado. De cierta forma, es el conocimiento sobre la estructura que necesitan las distintas operaciones para funcionar correctamente y que garantizan las mismas al finalizar su ejecución. Además funciona como un concepto coordinador entre las operaciones. En él quedan expresados las restricciones de coherencia de la estructura, surgidas de la redundancia de información que pueda haber. Su dominio es la imagen funcional del tipo que estamos implementando, lo cual es necesario para que podamos "tratar" los elementos del dominio en lógica de primer orden.

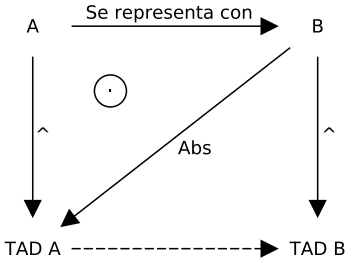
El invariante debe ser cierto tanto al comienzo de las **operaciones exportadas** del tipo como al final de las mismas. Por lo tanto, las operaciones del tipo deberán preservarlo, aunque quizás no sea cierto en algún estado intermedio del algoritmo que implemente alguna operación. Las **operaciones auxiliares** o internas, es decir aquellas que no aparecen en la interfaz y por lo tanto son invisibles a factores externos del módulo, no tienen la necesidad conservar el invariante, será nuestra responsabilidad conocer que modificaciones le realizan a la estructura interna (de hacerlas) para luego restaurar el invariante.

De cierta forma, el invariante de representación fuerza a las operaciones a cumplir ciertos ordenes de complejidad y a preservar la coherencia en la estructura de representación. Además al ser un predicado que debe cumplirse gran parte del tiempo, podremos deducir propiedades del mismo que podremos usar como parte de nuestras demostraciones de correctitud y complejidad.

Función de abstracción

$$Abs : \text{genero\_de}\widehat{\text{\_representacion}}\ g \rightarrow \text{genero\_del\_tipo\_}\widehat{\text{abstracto}}\text{\_representado} \quad \{Rep(g)\}$$

La función de abstracción es una herramienta que permite vincular una estructura de representación con el valor abstracto al que representa, es decir con el TAD vinculado al módulo de abstracción. Tiene por dominio al conjunto de instancias que son la imagen abstracta del tipo de representación (al igual que el invariante de representación) y que además verifican el invariante de representación, por esto mismo la función sera sobreyectiva. La función devuelve una imagen abstracta de la instancia del tipo representado (aquella instancia que estamos pretendiendo representar del tipo abstracto) y diremos que T representa a A si $Abs(\widehat{T}) =_{obs} A$, en donde T es una instancia de la representación y A una instancia del TAD. Informalmente la función de abstracción cumple la función de verificar que las funciones que alteran la estructura de representación realicen lo que está especificado en el TAD y no otra cosa.



A será el modulo en cuestión que estaremos diseñando y B la estructura de representación del mismo. El circulo con el punto dentro hace alusión al hecho de que la conmutación del triangulo conserva la sanidad. Esto significa que la aplicación de la función de abstracción a B deberá dar el mismo resultado que la aplicación de $\widehat{\bullet}$ a A.

Tendremos dos formas de describir la función de abstracción. La primera de ellas, en función de sus observadores básicos. Dado que los observadores identifican de manera unívoca al objeto del cual hablan, si nosotros describimos el valor que tienen los observadores básicos una vez aplicados al objeto, estaremos describiendo desde un punto observacional pero sin ambigüedad el objeto representado. Otra forma de describir la función de abstracción es utilizando los generadores del tipo de representación. La elección de elegir una forma u otra de hacerlo depende de la comodidad y declarativiidad.

Dicho de otra forma, el objetivo de la función de abstracción es poner en consonancia el tipo de soporte del modulo abstracto con los observadores del tipo que esta siendo representado. Una de las formas de realizar esto es caracterizando el valor de todos los observadores del TAD en base a valores de la instancia de la estructura de representación, de esa forma obtiene una instancia del TAD en consonancia con el modulo abstracto.

Algoritmos

En este paso se implementarán las operaciones del tipo a diseñar en términos de operaciones de los tipos soporte. Deben aparecer los algoritmos de cada una de las operaciones, sean éstas de la interfaz o auxiliares. En el caso de las funciones auxiliares, es recomendable incluir junto a sus algoritmos, sus precondiciones y postcondiciones. En el diseño de los algoritmos hay que tener en cuenta que las operaciones deben satisfaces sus pre/postcondiciones, y además deben satisfacer los requerimientos de eficiencia surgidos en el contexto de uso.

Servicios usados

Aquí es donde indicaremos qué responsabilidades le dejamos a los tipos soporte que usamos. Son las pautas y requerimientos que se extraen del diseño de este tipo para el diseño de los tipos de la representación. Luego pasarán a ser las interfaces y los contextos de uso y requerimientos de eficiencia para los módulos de soporte de los tipos usados en la representación.

Capítulo 3

Complejidad y Estructuras de Datos

3.1. Complejidad

3.1.1. Notación asintótica

Las notaciones que usamos para describir la complejidad temporal asintótica de un algoritmo están definidos en términos de funciones cuyos dominios son el conjunto de los números naturales $N = \{0, 1, 2, ...\}$. Estas notaciones son convenientes para describir el tiempo en el peor caso de una función $T(n)$, la cual usualmente esta definida solo en tamaños de entrada enteros. Sin embargo a veces encontramos conveniente abusar de la notación asintótica en variadas formas. De todas formas, deberemos cerciorarnos de entender precisamente el significado de la notación, para que cuando hagamos abuso de la misma no estemos usándola de forma errónea. Usaremos la notación asintótica primariamente para describir los tiempos que insumen los algoritmos, sin embargo la notación asintótica aplica a funciones.

Conjunto / Notación Asintótica Θ

Para una función $g(n)$ dada, notaremos a $\Theta(g(n))$ como el conjunto de funciones $f(n)$ que a partir de un numero $n_0 > 0$ sus valores pueden ser acotados entre $c_1 g(n)$ y $c_2 g(n)$ para algún $c_1, c_2 \in \Re_{>0}$. Descripto formalmente quedaría de la siguiente manera:

$$\Theta(g(n)) = \{f(n) : (\exists\ c_1, c_2, n_0 \in \Re_{>0})\ (\forall n \mid n_0 \leq n)\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Una función pertenece al conjunto $\Theta(g(n))$ si existen constantes positivas c_1 y c_2 tal que $f(n)$ puede ser "sandwicheada" entre $c_1 g(n)$ y $c_2 g(n)$, para un n suficientemente grande. Como $\Theta(g(n))$ es un conjunto, podremos escribir "$f(n) \in \Theta(g(n))$" para indicar que $f(n)$ es un miembro de $\Theta(g(n))$, a veces escribiremos "$f(n) = \Theta(g(n))$" para expresar exactamente lo mismo. Cuando una función $f(n)$ es acotada superiormente por otra función $c \cdot g(n)$ para algún c y a partir de algún n, diremos que $f(n)$ esta **acotada asintóticamente** por $g(n)$ o que $f(n)$ tiene un **comportamiento asintótico** a $g(n)$. En el caso de $\Theta(g(n))$ diremos que $g(n)$ es una **cota asintóticamente ajustada** para $f(n)$.

Propiedades de Θ:

1. Para cualquier función f se tiene que $f \in \Theta(f)$.
2. $\Theta(f) = \Theta(g) \iff f \in \Theta(g) \iff g \in \Theta(f)$.
3. Sí $f \in \Theta(g) \land g \in \Theta(h) \implies f \in \Theta(h)$.
4. Sí $f \in \Theta(g) \land f \in \Theta(h) \implies \Theta(g) = \Theta(h)$.
5. Regla de la suma:
   Sí $f_1 \in \Theta(g) \land f_2 \in \Theta(h) \implies f_1 + f_2 \in \Theta(g + h)$.
6. Regla del producto:
   Sí $f_1 \in \Theta(g) \land f_2 \in \Theta(h) \implies f_1 * f_2 \in \Theta(g * h)$.
7. Sí existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, según los valores que tome k:

a) Sí $k \neq 0$ y $k < \infty$ entonces $\Theta(f) = \Theta(g)$.

b) Sí $k = 0$ entonces $\Theta(g) \neq \Theta(f)$.

Conjunto / Notación Asintótica O

El conjunto Θ asintóticamente acota una función inferior y superiormente. Cuando solo tenemos una **cota asintótica superior**, usaremos el conjunto O. Para una función dada $g(n)$, denotaremos por $O(g(n))$ al conjunto de funciones tales que:

$$O(g(n)) = \{f(n) : (\exists\ c, n_0 \in \Re_{>0})\ (\forall n \mid n_0 \leq n)\ 0 \leq f(n) \leq c \cdot g(n)\}$$

Es decir que para todos los valores de n a la derecha de n_0, el valor de la función $f(n)$ está por debajo de $c \cdot g(n)$. Escribiremos $f(n) = O(g(n))$ para indicar que una función $f(n)$ es un miembro del conjunto $O(g(n))$. Notar que $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$, ya que la noción de Θ es mucho mas fuerte que la noción de O. Esto es, escrito en forma de teoría de conjuntos, que vale la inclusión $\Theta(g(n)) \subseteq O(g(n))$

Propiedades de O:

1. Para cualquier función f se tiene que $f \in O(f)$.
2. $f \in O(g) \implies O(f) \subset O(g)$.
3. $O(f) = O(g) \iff f \in O(g) \wedge g \in O(f)$.
4. Sí $f \in O(g) \wedge g \in O(h) \implies f \in O(h)$.
5. Sí $f \in O(g) \wedge f \in O(h) \implies f \in O(min(g, h))$.
6. Regla de la suma:
   Sí $f_1 \in O(g) \wedge f_2 \in O(h) \implies f_1 + f_2 \in O(max(g, h))$.
7. Regla del producto:
   Sí $f_1 \in O(g) \wedge f_2 \in O(h) \implies f_1 * f_2 \in O(g * h)$.
8. Sí existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, según los valores que tome k:
   a) Sí $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$.
   b) Sí $k = 0$ entonces $f \in O(g)$, es decir, $O(f) \subset O(g)$, pero sin embargo se verifica que $g \notin O(f)$.

Conjunto / Notación Asintótica Ω

De forma tal como la notación O proporciona una cota asintótica en una función, Ω provee una **cota asintótica inferior**. Para una función dada $g(n)$, notaremos a $\Omega(g(n))$ como el conjunto de funciones tales que:

$$\Omega(g(n)) = \{f(n) : (\exists\ cn_0 \in \Re_{>0})\ (\forall n \mid n_0 \leq n)\ 0 \leq c \cdot g(n) \leq f(n)\}$$

Cuando decimos que el tiempo de un algoritmo es $\Omega(g(n))$, significa que no importa que particular entrada de tamaño n para cada valor de n, el tiempo que tardara el algoritmo con dicha entrara sera de al menos un numero constante de veces $g(n)$, para un n suficientemente grande. Equivalentemente, esto nos da una cota temporal inferior para el mejor caso del algoritmo. De las definiciones de las notaciones asintóticas, es fácil ver que para cualquier dos funciones $f(n)$ y $g(n)$, tendremos que $f(n) = \Theta(g(n))$ si y solo si $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$. Usando una notación de teoría de conjuntos esto seria equivalente a decir que $\Omega(g(n)) \cap O(g(n)) = \Theta(g(n))$, y si $f(n) \in \Omega(g(n)) \wedge f \in O(g(n))$ entonces $f(n)$ pertenecerá a ambos conjuntos, por lo que en particular pertenecerá a la intersección $\Theta(g(n))$.

Propiedades de Ω:

1. Para cualquier función f se tiene que $f \in \Omega(f)$.
2. $f \in \Omega(g) \implies \Omega(f) \subset \Omega(g)$.
3. $\Omega(f) = \Omega(g) \iff f \in \Omega(g) \wedge g \in \Omega(f)$.

4. Sí $f \in \Omega(g) \wedge g \in \Omega(h) \implies f \in \Omega(h)$.
5. Sí $f \in \Omega(g) \wedge f \in \Omega(h) \implies f \in \Omega(max(g,h))$.
6. $f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$
7. Regla de la suma:
   Sí $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \implies f_1 + f_2 \in \Omega(g+h)$.
8. Regla del producto:
   Sí $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \implies f_1 * f_2 \in \Omega(g*h)$.
9. Sí existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, según los valores que tome k:
   a) Sí $k \neq 0$ y $k < \infty$ entonces $\Omega(f) = \Omega(g)$.
   b) Sí $k = 0$ entonces $g \in \Omega(f)$, es decir, $\Omega(g) \subset \Omega(f)$, pero sin embargo se verifica que $g \notin \Omega(f)$.

Conjunto / Notación Asintótica No-Ajustada o

La cota superior asintótica provista por O puede o no puede ser ajustada asintóticamente, por ejemplo la cota $2n = O(n^2)$ no lo es. Usaremos la notación o (o chica) para definir una cota superior que no sea ajustada asintóticamente. Para una función dada $g(n)$, definiremos formalmente a $o(g(n))$ como el conjunto de funciones tales que:

$$o(g(n)) = \{f(n) : (\forall c \in \Re_{>0})(\exists\ n_0 \in \Re_{>0})\ (\forall n \mid n_0 \leq n)\ 0 \leq f(n) < c \cdot g(n)\}$$

Las definiciones de O (o grande) y o (o chica) son sumamente similares. La diferencia principal yace en que $f(n) = O(g(n))$, la cota $0 \leq f(n) \leq c \cdot g(n)$ se mantiene para alguna constante $c > 0$ mientras que en $f(n) = o(n)$, la cota $0 \leq f(n) < c \cdot g(n)$ se mantiene para todas las constantes $c > 0$. En este sentido, la notación o es más fuerte que la notación O: sí $f(n) \in o(g(n)) \implies f(n) \in O(g(n))$, pero no vale la vuelta.

Intuitivamente, en la notación o, la función $f(n)$ es dominada asintóticamente por $g(n)$ a medida que n tiende a infinito, para cualquier constante c.

Conjunto / Notación Asintótica No-Ajustada ω

Analógicamente, ω es a Ω como o es a O. Usaremos la notación ω para referirnos a una cota inferior que no sea ajustada asintóticamente. Una forma de definirla en función a o sera que $f(n) \in \omega(g(n))$ si y solo si $g(n) \in o(f(n))$. De todas formas formalmente definiremos a ω como el conjunto de funciones tal que dada una función $g(n)$ es:

$$\omega(g(n)) = \{f(n) : (\forall c \in \Re_{>0})(\exists\ n_0 \in \Re_{>0})\ (\forall n \mid n_0 \leq n)\ 0 \leq c \cdot g(n) < f(n)\}$$

A medida que n tiende a infinito la relación entre $f(n)/g(n)$ se vuelve arbitrariamente grande, es decir que tiende a infinito.

3.1.2. Funciones de Complejidad temporal comunes

A continuación se listan las principales categorías de complejidad temporal, en orden creciente:

1. $O(1)$ **Complejidad constante**: es independiente de los datos de entrada.
2. $O(lg\ lg\ n)$ **Complejidad sublogarítmica**
3. $O(lg\ n)$ **Complejidad logarítmica**: suele aparecer en determinados algoritmos con iteración o recursión (por ejemplo, búsqueda binaria). Todos los logaritmos, sea cual sea su base, son del mismo orden, por lo que se representan en cualquier base.
4. $O(n^c)$ **Complejidad sublineal** $(0 < c < 1)$
5. $O(n)$ **Complejidad lineal**: suele aparecer en bucles simples cuando la complejidad de las operaciones internas es constante o en algunos algoritmos con recursión.

6. $O(n \cdot lg\ n) = O(lg\ n!)$ **Complejidad lineal logarítmica**: en algunos algoritmos de "Divide & Conquer" (por ejemplo, Mergesort).
7. $O(n^2)$ **Complejidad cuadrática**: aparece en bucles o recursiones doblemente anidados.
8. $O(n^3)$ **Complejidad cúbica**: en bucles o recursiones triples.
9. $O(n^k)$ **Complejidad polinómica** ($k \geq 1$).
10. $O(2^n)$ **Complejidad exponencial**: suele aparecer en subprogramas recursivos que contengan dos o más llamadas recursivas.
11. $O(n!)$ **Complejidad factorial**
12. $O(n^n)$ **Complejidad potencial exponencial**

3.2. Estructuras básicas

3.2.1. Lista

3.2.2. Cola

La estructura Cola se puede pensar como un arreglo para guardar los elementos más un natural para saber la cantidad. Las colas se caracterizan por ser una estructura de tipo FIFO (First in, First out. O en castellano: Primero que entra, primero que sale). Una implementación efectiva de una Cola puede hacerse usando **Aritmetica Circular**, alias: función módulo. Idea:

1. Tengo un arreglo de MAX_CANTIDAD elementos.
2. En una variable *cant* guardo la cantidad de elementos actuales en la cola.
3. En una variable *primero* guardo la posición en el arreglo del primer elemento.
4. Encolar: agrega el nuevo elemento a la posición $(primero+cant)\,\%MAX\_CANTIDAD$ del arreglo e incrementa *cant* en uno.
5. Desencolar: setea $primero = (primero + 1)\,\%MAX\_CANTIDAD$ y decrementa *cant* en uno.

Siendo las complejidades resultantes:

- **Próximo** $O(1)$
- **Desencolar** $O(1)$
- **Encolar** $O(1)$
- **Búsqueda** $O(n)$

Una clara limitación de ésta implementación es que solo puede almacenar hasta MAX_CANTIDAD elementos al mismo tiempo. Se puede resolver esto y agregar algunas ventajas utilizando **Memoria Dinámica**.

3.2.3. Pila

3.2.4. Arreglo dimensionable

3.3. Árboles

3.3.1. Árbol binario de búsqueda

Un árbol binario de búsqueda es un árbol basado en nodos binarios el cual puede ser representado por una estructura de punteros en donde cada nodo es un objeto. En adición a la clave y a los datos, cada nodo contiene los atributos izquierda, derecha y p, los cuales son punteros que apuntan a su hijo izquierdo, hijo derecho y padre, respectivamente. Si un hijo o un padre esta perdido el atributo apropiado contiene el valor NIL. El nodo raíz es el único nodo cuyo padre es NIL.

Las claves en un árbol binario de búsqueda siempre están guardadas de forma tal de satisfacer la **invariante del árbol binario de búsqueda**, el cual se define como: "Sea x un nodo en un árbol de búsqueda binario. Si y es un nodo en el sub-árbol izquierdo de x, entonces $y.clave \leq x.clave$. Si y es un nodo en el sub-árbol derecho de x, entonces $y.clave \geq x.clave$"

Búsqueda

La búsqueda en un árbol binario se logra utilizando el invariante presentado anteriormente. El procedimiento comienza en la raíz y por cada nodo x que encuentra compara la clave k con $x.clave$, si las dos claves son iguales, la búsqueda termina. Si k es mas chico que $x.clave$, la búsqueda continua por el sub-árbol izquierdo de x, dado que el invariante del árbol binario de búsqueda implica que k no puede ser guardado en el sub-árbol derecho. Simétricamente, si k es mas grande que $x.clave$, la búsqueda continua por el sub-árbol derecho. Los nodos encontrados durante la recursión forman un camino simple desde la raíz, por lo que el tiempo de corrida de una búsqueda en un árbol binario es de $O(h)$, en donde h es la altura del árbol la cual puede ser n si tenemos un árbol totalmente degenerado, que de cierta forma terminaría siendo una lista enlazada.

Además, siempre podremos encontrar un elemento en un árbol binario cuya clave sea un mínimo o un máximo siguiendo los punteros izquierdos o derechos desde la raíz hasta que encontremos un NIL.

Inserción

Las operaciones de inserción y borrado causa que la dinámica del conjunto representado por un árbol binario de búsqueda cambie. La estructura de datos debe ser modificada para reflejar este cambio, pero de forma tal que el invariante se conserve.

Para insertar un nuevo valor v en un árbol binario de búsqueda T, crearemos un nuevo nodo z de tal forma que $z.clave = v$, $z.izq = NIL$ y $z.der = NIL$. Modificara T y alguno de los atributos de z de tal forma que inserte z en la posición apropiada del árbol. Para lograr insertar el nuevo nodo el procedimiento guardará en una variable p un puntero al nodo actual en donde se encuentra mientras que ejecuta una búsqueda en el árbol. Una vez que llega a un nodo NIL durante la búsqueda, asigna $z.p = p$ y compara $z.clave$ con $p.clave$, si es mayor asigna $p.izq = z$ y si es menor $p.der = z$, de esta forma el invariante queda restaurado. El tiempo de esta operación se encontrara en el orden de $O(h)$, siendo que h puede ser n en el peor caso el tiempo será de $O(n)$, sin embargo el caso promedio suponiendo una distribución uniforme en la entrada de los valores sera de $O(lg(n))$.

Borrado

La estrategia general para eliminar un nodo z de un árbol binario de búsqueda T tiene 3 casos básicos, pero como veremos uno de ellos es algo engañoso.

- Si z no tiene hijos, entonces simplemente lo removeremos modificando su padre $z.p$ reemplazando el puntero a z por NIL.
- Si z solo tiene un hijo, entonces **elevaremos** dicho hijo para tomar la posición de z modificando el padre $z.p$ reemplazando el puntero a z por el puntero al único hijo de z.
- Si z tiene dos hijos, entonces encontraremos el mínimo y en el sub-árbol derecho de z (es decir, su sucesor), para tomar la posición de z en el árbol. Como y era el mínimo del sub-árbol derecho de z, $y.izq$ sera NIL y si y tiene un hijo izquierdo lo elevaremos al padre de y. Luego, con y libre hijos, lo reemplazaremos por z y le asignaremos los sub-árboles derecho e izquierdo, como hijos de y. Esto mismo se puede hacer si tomamos como y el máximo del sub-árbol izquierdo de z.

Balanceo perfecto

Vimos que los ABB funcionan razonablemente bien en el caso promedio, pero no dan garantías. Todos los algoritmos que vimos tienen un peor caso lineal. En éste sentido, los arreglos también son lineales en el peor caso, pero ocupan menos memoria.

Pero si miramos en detalle, veremos que los algoritmos más bien son $O(h)$, donde h es la altura del árbol. Entonces, sí distribuyésemos los nodos del ABB de manera "pareja", de manera tal que el árbol tuviese la mínima altura y estuviese siempre parejo, obtendríamos unas complejidades mucho mejores.

Teorema: un árbol binario perfectamente balanceado de n nodos (no nil, es decir $n > 0$) tiene altura $\lfloor log_2 n \rfloor + 1$.

Supongamos que cada nodo tiene 0 o 2 hijos. Llamemos n_i a la cantidad de nodos internos (más la raiz) y n_h a la cantidad de hojas.

Propiedad: Si $n > 1$, $n_h = n_i + 1$

El caso base es trivial. $n = 3$ (raiz con dos hijos) $\implies n_h = 2 \wedge n_i = 1$ (la raiz). Entonces se cumple la propiedad.

Para el paso inductivo, supongamos que vale para n_h y n_i. Ahora, agreguemos un nivel al arbol. Las hojas se duplican, $n'_h = 2n_h$ y los nodos internos crecen en tantos como antes tenía hojas: $n'_i = n_i + n_h$, por hipótesis inductiva: $n'_i = (n_h - 1) + n_h = 2n_h - 1 = n'_h - 1$. □

Corolario: al menos la mitad de los nodos son hojas.

Luego, sabemos que (1) $n' = n'_i + n'_h$ y (prop) sí $n > 1$, $n_h = n_i + 1$. Imaginemos que podamos las hojas: nos queda un árbol (llamemos n a la cantidad de nodos) con las mismas propiedades, 1 menos de altura (llamémosla h), la mitad de los nodos y ahora todas las ramas de la misma longitud. ¿Cuántas veces más podemos podarlo? O dicho de otra forma: ¿Cuántos niveles se pueden agregar desde el comienzo para tener un árbol de altura h?

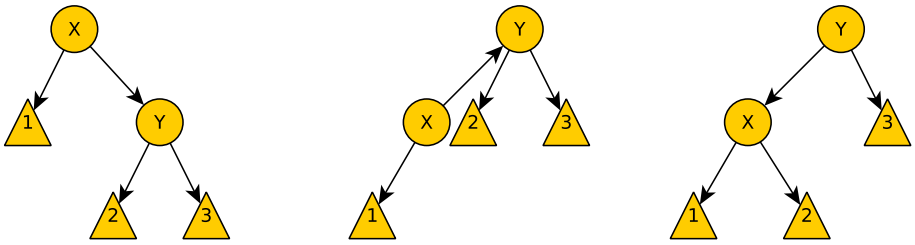
Al agregar un nivel, la cantidad de nodos se duplica, porque $n'_h = n'_i + 1$, pero $n'_i = n$, entonces $n'_h = n + 1$. Reemplazando en (1) nos queda que $n' = n + (n + 1) + 1$. Entonces $n = 1.2 \ldots 2 = 2^h = 2^{log_2 n}$. Por ende, $h = log_2 n$ y la altura del árbol era $h + 1$. □

Conclusión, sí tuviésemos árboles perfectamente balanceados todas nuestras operaciones serían $O(log\ n)$. Pero mantener un balanceo perfecto es muy costoso, por lo que no es factible. Sin embargo, podemos tener un balanceo "casi" perfecto, haciendo que todas las ramas tengan "casi" la misma longitud, donde ese "casi" lo vamos a interpretar de la siguiente manera: la longitud entre dos ramas cualesquiera de un nodo difiere a lo sumo en 1.

Ésta clase de ABBs, son los llamadas árboles AVL.

Rotaciones

Las rotaciones son operaciones locales en un árbol de búsqueda que cambian la estructura de punteros del mismo preservando el invariante del árbol binario, son mayormente usadas en las distintas implementaciones de árboles balanceados para ajustar los factores de balanceo del mismo. Usaremos dos tipos de rotaciones, rotaciones hacia la izquierda y rotaciones hacia la derecha. Cuando hacemos una rotación hacia la izquierda en un nodo x, asumimos que su hijo derecho y no es NIL; x podrá ser cualquier nodo en el árbol cuyo hijo derecho no es NIL. La rotación hacia la izquierda "pivotea" alrededor del enlace de x a y y hace a y la nueva raíz del sub-árbol, dejando como hijo izquierdo de y a x y trasladando a el hijo izquierdo de y al lugar del hijo derecho de x.



Si nos imaginamos la rotación como bolillas e hilos, "tiraríamos" y hacia arriba a la izquierda, haciendo que x baje y quede a la altura del hijo izquierdo de y. Luego, como el hijo izquierdo de y se encontraba en el sub-árbol derecho de x, sera mas grande que x, por lo que lo podremos asignar como hijo derecho de x.

Recorrido de árboles

El recorrido de árboles, en ingles *Tree traversal*, se refiere al proceso de visitar de manera sistemática, exactamente una vez, a cada nodo de un árbol. Tales recorridos están clasificados en el orden en el cual son visitados los nodos:

Preorden (o "preorder"): (raíz, izquierdo, derecho). Comenzando con el nodo raíz:

- Visitar la raiz.
- Recorrer el subárbol izquierdo en preorden.
- Recorrer el subárbol derecho en preorden.

Inorden (o "inorder"): (izquierdo, raíz, derecho). Comenzando con el nodo raíz:

- Recorrer el subárbol izquierdo en inorden.
- Visitar la raiz.
- Recorrer el subárbol derecho en inorden.

Postorden (o "postorder"): (izquierdo, derecho, raíz). Comenzando con el nodo raíz:

- Recorrer el subárbol izquierdo en postorden.
- Recorrer el subárbol derecho en postorden.
- Visitar la raiz.

Complejidades

- **Búsqueda** $O(n)$ (caso promedio $O(lg(n))$)
- **Inserción** $O(n)$ (caso promedio $O(lg(n))$)
- **Borrado** $O(n)$ (caso promedio $O(lg(n))$)
- **Espacio** $O(n)$

3.3.2. Heap

La estructura de datos del heap (binario) es un arreglo de objetos que representa una Cola de Prioridad y que podremos ver como un árbol binario casi-completo. Cada nodo del árbol corresponde a un elemento del arreglo y el árbol esta completamente lleno en todos los niveles excepto en el ultimo, el cual esta lleno desde la izquierda hasta algún punto. Un arreglo A que representa un heap es un objeto con dos atributos, $A.lenght$ que nos dirá la cantidad de elementos en el arreglo y $A.heap\_size$ que representara cuantos elementos en el heap están guardados en el arreglo A. Esto es, que a pesar que $A[1..A.length]$ contenga números, solo los elementos en $A[1..A.heap\_size]$ (en donde $0 \leq A.heap\_size \leq A.length$) son elementos validos del heap.

La raíz del árbol estará situada en $A[1]$, y dado el índice i de un nodo, fácilmente podremos computar los índices de su padre, hijo derecho e izquierdo. El cómputo del índice del padre estará definido de la forma $Padre(i) = \lfloor i/2 \rfloor$, mientras que los índices de los hijos podrán ser computados de la forma $Izquierda(i) = 2i$ y $Derecha(i) = 2i + 1$, en donde $i \in [1..A.heap\_size]$. Hay dos tipos de heaps binarios, max-heaps y min-heaps. En ambos casos, los valores de los nodos satisfacen el invariante del heap, el cual depende del tipo de heap con el que trabajemos. En un max-heap, el invariante del max-heap nos dice que para cada nodo i que no sea la raíz se debe cumplir que $A[Padre(i)] \geq A[i]$, esto es que el valor de un nodo debe ser como máximo el valor de su padre. Entonces, el máximo en un max-heap estará guardado en la raíz, y el sub-árbol con raíz en un nodo no podrá contener valores mas grandes que el nodo en si mismo. Un min-heap esta organizado de la forma opuesta, el invariante de min-heap nos dice que para cada nodo i distinto de la raíz se debe cumplir $A[Padre(i)] \leq A[i]$, por lo que el elemento mínimo de un min-heap estará en la raíz.

Viendo el heap como un árbol, definiremos la altura de un nodo en un heap como el numero de aristas en el camino simple mas largo desde el nodo mismo hasta una hoja, y definiremos la altura del heap como la altura de su raíz. Ya que un heap de n elementos esta basado en un árbol binario completo, su altura pertenecerá a $\Theta(lg(n))$.

Mantener el invariante del heap

De aquí en mas, asumiremos que estaremos hablando de un max-heap, ya que el min-heap es análogo. Para mantener el invariante del max-heap en un arreglo, llamaremos al procedimiento **Heapify**. Su entrada sera el arreglo A y un índice i del arreglo. Cuando es llamado, Heapify asume que los sub-árboles binarios con sus raíces en $Izquierda(i)$ y $Derecha(i)$ serán max-heaps, pero que $A[i]$ puede ser mas chico que que sus hijos, violando así el invariante. Heapify deja que el valor en $A[i]$ "decante" en el max-heap, de forma tal que el sub-árbol con raíz en i cumpla el invariante del heap y luego recursivamente se asegurara que el invariante sea cumplido en el sub-árbol en donde el valor $A[i]$ haya decantado.

Puntualmente lo que realiza Heapify es, estando situado en i, compara el valor con los valores de los hijos de i. Si ninguno de los valores de los hijos es mas grande que el valor de i lo deja como esta, en el caso contrario intercambia el valor de i con el valor de su hijo mas grande. Si bien en el nodo original el invariante se cumple, el invariante puede ser violado en el sub-árbol cuya raíz fue intercambiada por lo que se realiza una llamada recursiva utilizando como i el nuevo índice donde el valor fue intercambiado.

El tiempo de Heapify en un sub-árbol de tamaño n con su raíz en un nodo i sera el tiempo $\Theta(1)$ para ajustar las relaciones entre los elementos $A[i]$, $A[Izquierda(i)]$ y $A[Derecha(i)]$, mas el tiempo que tome correr Heapify en un sub-árbol con su raíz en alguno de los hijos del nodo i (asumiendo que la llamada recursiva ocurra). El peor caso ocurrirá cuando el nivel inferior del árbol se encuentre exactamente a la mitad de estar lleno, por lo que cada uno de los sub-árboles de los hijos tendrán como mucho $2n/3$ elementos. Podremos describir el tiempo de Heapify con la recurrencia $T(n) \leq T(2n/3) + \Theta(1)$. Si tuviésemos $T(n) = T(2n/3) + \Theta(1)$, la recurrencia se ajusta al segundo caso del teorema maestro, por lo que $T(n) \subseteq \Theta(lg(n))$ pero como tenemos una relación de menor o igual en la recurrencia original, $T(n) \subset O(lg(n))$.

Construyendo un heap (Algoritmo de Floyd)

Podremos usar el procedimiento Heapify de forma bottom-up para convertir un arreglo $A[1..n]$, en donde $n = A.length$, en un max-heap. Los elementos en el sub-arreglo $A[\lfloor n/2 \rfloor + 1..n]$ son todas hojas del árbol, y por lo tanto cada uno es un heap de 1 elemento por el cual podremos comenzar. El procedimiento para construir un heap va por cada uno de los nodos del árbol restantes comenzando por el índice mas grande, es decir $i = \lfloor A.length/2 \rfloor.,1$, y ejecuta Heapify en cada uno de ellos. Una cota grosera para este procedimiento es $O(nlg(n))$, ya que hará n llamadas a Heapify. Sin embargo, una cota mas ajustada puede ser dada basándonos en que la altura de un árbol heap de n elementos tiene altura $\lfloor lg(n) \rfloor$ y como mucho $\lceil n/2^{h+1} \rceil$ nodos de cualquier altura h, la cual corresponderá con $O(n)$.

Extraccion de máximo e inserción

El máximo en un max-heap sera $A[1]$, el cual una vez extraído no será eliminado del arreglo sino que será intercambiado por el ultimo elemento del arreglo y luego se decrementara $A.heap\_size$ en una unidad, dejando fuera del scope al ultimo elemento del arreglo que será aquel que acabamos de extraer. El único problema con esto es que estaremos rompiendo el invariante del heap, pero para solucionarlo solo nos bastara con aplicar $Heapify(A, 1)$ ya que el resto de los sub-heaps seguirán siendo validos. Como todas las operaciones son $\Theta(1)$ exceptuando Heapify, la cual se usará una sola vez, el tiempo total del algoritmo sera de $O(lg(n))$.

Para insertar un nuevo elemento incrementaremos la variable $A.heap\_size$ en una unidad y asignaremos el nuevo valor a la ultima posición valida para el heap, es decir $A[heap\_size] = valor$. Luego, intercambiaremos su valor con el padre hasta que el invariante del heap se cumpla.

En resumen

Los heaps son árboles balanceados en donde la clave de cada nodo es mayor o igual que la de sus hijos y en donde todo sub-árbol es un heap. Adicionalmente puede o no cumplir que por la forma de su implementación sea izquierdista, esto quiere decir que el árbol estará todo lleno exceptuando su ultimo nivel, el cual se llenara de izquierda a derecha.

Su representación podrá estar dada por cualquier tipo de estructura para árboles binarios, pero si bien sera mas eficiente si se lo representa en un arreglo, pasaremos a una representación estática por lo que podremos perder tiempo agrandando un arreglo. Sea un nodo v, la posición dentro de un heap se calcula:

- Si v es la raíz, $P(v) = 1$
- Si v es un nodo que no es la raíz, su padre u sera $P(v) = \lfloor P(u)/2 \rfloor$. Sea i un índice de un nodo: $Padre(i) = \lfloor i/2 \rfloor$.

- Si v es un hijo izquierdo de u $P(v) = 2P(u)$. Sea i un índice de un nodo: $Derecho(i) = 2i$.
- Si v es un hijo derecho de u $P(v) = 2P(v) + 1$. Sea i un índice de un nodo: $Izquierdo(i) = 2i + 1$.

Complejidades

Las complejidades de las operaciones del heap quedaran de la forma:

- **Ver el máximo o mínimo / Próximo** $O(1)$
- **Extraer el máximo o mínimo / Desencolar** $O(lg(n))$
- **Ingresar nuevo elemento / Encolar** $O(lg(n))$
- **Construir heap a partir de arreglo / Heapify** $O(n)$

3.3.3. Árbol balanceado en altura (AVL)

Un árbol AVL es un árbol binario de búsqueda que esta balanceado en altura. Esto es, que por cada nodo x, la altura de de los sub-árboles izquierdo y derecho de x difieren en a lo sumo 1. Para implementar un árbol AVL mantendremos un atributo extra en cada nodo $x.fdb$ que representara el factor de balanceo del nodo x. El factor de balanceo para un nodo x se calculara de la forma $FDB(x) = altura(der(x)) - altura(izq(x))$.

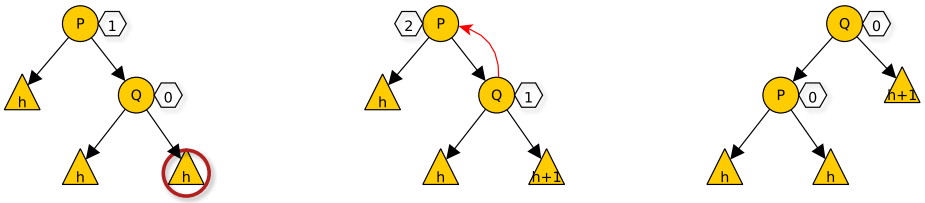
Inserción

La inserción de un nuevo nodo en un principio se realiza de la misma manera que en un ABB normal. Luego de haber insertado el nodo se realizan dos acciones:

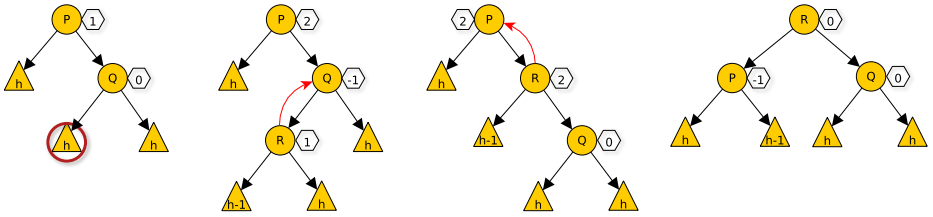
- Se recalculan los factores de balanceo de forma bottom-up, es decir comenzando por el nodo recién insertado hasta la raíz. Esto funciona ya que los factores de balanceo de otros nodos que no pertenezcan a la rama del nodo recién insertados no se verán afectados por la inserción.
- Si durante el recalculo aparece un factor ± 2 en alguno de los pasos, habrá que rebalancear el nodo antes de continuar.

Durante el rebalanceo de un nodo aparecerán distintos casos para los cuales deberemos aplicar distintas rotaciones. La notación de cada uno de los casos va de la forma D_1D_2, siendo x el nodo que necesitamos rebalancear D_2 indicará el hijo de x en cuyo sub-árbol D_1 fue hecha la inserción del nuevo elemento. Es decir, si tenemos un caso RL, significa que estando parados en un nodo x la inserción fue hecha en el sub-árbol derecho del hijo izquierdo de x. De cierta forma la notación se traduce desde abajo hacia arriba en un árbol.

Estos casos son LL y RR, a los cuales será suficiente con aplicarles rotaciones simples para solucionarlos y LR y RL a los cuales habrá que aplicarles una rotación para dejarlos en alguno de los casos anteriores y una segunda rotación para terminar de balancearlos. Tanto LL a RR y LR a RL son casos simétricos unos de otros.



Caso RR En la primera figura se observa el estado inicial, dentro de los hexágonos se encuentra el FDB de cada nodo. En la segunda figura se puede observar el estado luego de la inserción y FDB desbalanceado del nodo P. Ejecutando una rotación hacia la izquierda entre P y Q se restaura, como se ve en la tercera figura (la izquierda de Q pasa a ser P, y lo que era la izquierda original de Q ahora es la derecha de P). El caso LL es simétrico a este caso.



Caso LR En la primera figura se observa el estado inicial. En la segunda figura se puede observar el estado luego de la inserción con el hijo izquierdo R de Q expandido, la primera rotación se ejecutará hacia la derecha entre R y Q. La tercer figura refleja el estado luego de la primer rotación, en este estado se ejecutara la segunda rotación que será hacia la izquierda entre P y R. La cuarta figura refleja el estado final con el nodo balanceado. El caso RL es simétrico a este caso.

La rotaciones en el caso de inserción no influyen en los antepasados de x, por lo que su balance no será modificado, y por lo tanto a lo sumo necesitaremos realizar dos rotaciones para rebalancear el árbol luego de una inserción. El tiempo que tomara entonces una inserción sera $O(2lg(n)) \subseteq O(lg(n))$, ya que deberemos buscar el lugar para insertar el nodo y luego recalcular todos los factores de balanceo de la rama (de ser necesario se rebalanceará alguno de los nodos).

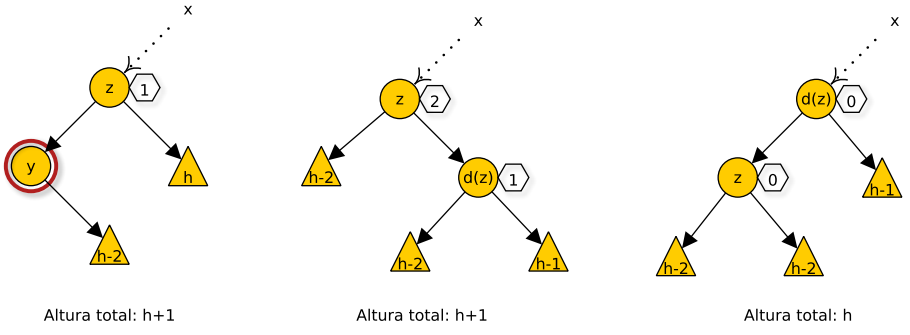
Eliminación

El borrado en un AVL en un principio se realiza de la misma forma que en un árbol binario, luego de haberlo hecho se ejecuta un recalculo de los factores y un rebalanceo sobre la rama que se vio afectada por el mismo. Una diferencia importante entre la inserción y el borrado es que a la hora de realizar los rebalanceos en la inserción los antepasados de un nodo x desbalanceado no se verán influenciados mientras que durante un rebalanceo luego de un borrado si lo harán.

Esto último se debe a que durante la inserción estaremos agregando una hoja nueva, y por la naturaleza del AVL el balanceo de un sub-árbol T con su raiz en x ocasiona que la altura de T se acorte. Esto significa que de cierta forma mediante el balanceo estaremos cancelando el incremento de altura en el sub-árbol T ocasionado por la inserción, por lo que los antepasados de x no sufren modificaciones. En el caso de una eliminación, el balanceo ocasionará también que la altura del sub-árbol se acorte pero no habrá compensación por una inserción, lo que decantara en una modificación de los factores de balanceo de los antepasados de x, pudiendo esto llegar a provocar $lg(n)$ balanceos. La complejidad de la operación de eliminación sera entonces de $O(2lg(n)) \subseteq O(lg(n))$ ya que tendremos que buscar el nodo a eliminar, actualizar los factores de balanceo y, en el peor caso, balancear toda la rama afectada (podremos actualizar los factores y rebalancear sin iterar nuevamente).

Peor caso en eliminación

Para lograr caracterizar un peor caso en el rebalanceo luego de la eliminación de un nodo, definiremos a T como un árbol AVL en donde todos los nodos que no son hojas tendrán factor de balanceo 1. Esto es que para todo nodo x la diferencia entre la altura del sub-árbol derecho y el sub-árbol izquierdo sera 1, es decir que $FDB(x) = altura(der(x)) - altura(izq(x)) = 1$. Luego, si removemos una hoja y del sub-árbol $izq(x)$, sea z el padre de y tal que $izq(z) = y$ y $izq(y) = NIL$, ocasionara que el FDB de z se incremente de 1 a 2, ya que habremos acortado el sub-árbol izquierdo de z en 1 unidad.



En la primera figura se puede apreciar el nodo y a ser eliminado, junto con su padre z y el FDB de z en el hexágono, la altura total es de $h+1$. En la segunda figura se puede apreciar el estado luego de la eliminación del nodo y junto con el hijo de derecho de z expandido y los FDB correspondientes a cada nodo recalculados. Finalmente, en la tercera figura se observa el estado luego de la rotación, lo que genera un decremento en la altura del sub-árbol.

Para compensar esto ejecutaremos una rotación hacia la izquierda entre los nodos z y $der(z)$, lo que dejara a z como hijo izquierdo de $der(z)$ y al hijo izquierdo de $der(z)$ como hijo derecho de z. Luego de la rotación los FDB de ambos nodos quedarán en 0 pero la altura del árbol se verá reducida en 1, lo que provocará que el padre original de z (el abuelo de y), cambie su FDB de 1 a 2. De la misma forma deberemos continuar balanceando el resto de los nodos hasta llegar a la raíz, en donde tendremos que $FDB(izq(raiz)) = 0$, $FDB(der(raiz)) = 1$ y $FDB(raiz) = 2$. Para resolver esta ultima situacíon podremos ejecutar una rotación hacia la derecha entre $izq(raiz)$ y $raiz$, o una rotación hacia la izquierda entre $der(raiz)$ y $raiz$. Finalmente habremos realizado exactamente $lg(alt(y))$ rotaciones, en donde $alt(y)$ es distancia a la que se encontraba y de la raíz, es decir su altura o nivel. Dentro de este esquema, si seleccionamos a y como uno de los nodos a nivel $\lfloor lg(n) \rfloor$ del árbol, entonces tendremos el peor caso de balanceo, por lo que la complejidad de balanceo luego de una eliminación será de $O(lg(n))$.

Complejidades

- **Búsqueda** $O(lg(n))$
- **Inserción** $O(lg(n))$
- **Borrado** $O(lg(n))$
- **Espacio** $O(n)$

3.3.4. B

Los árboles B son árboles balanceados de búsqueda diseñados para trabajar bien en discos rígidos u otros dispositivos de almacenamiento secundario. Los árboles B son similares a los árboles RB, pero son mejores minimizando las operaciones I/O. Varios sistemas de base de datos utilizan árboles B o variantes para guardar información.

Los árboles B difieren de los árboles RB en el sentido de que los nodos de los árboles B pueden tener muchos hijos, desde unos pocos hasta miles. Esto es, que el "factor de ramificación" (branching factor) de un árbol B puede ser bastante grande, sin embargo usualmente depende en las características de la unidad de disco usada. Este tipo de árboles son similares a los árboles B en el aspecto de que cada $n-$nodo tiene altura $O(lg(n))$. La altura exacta de un árbol B puede ser considerablemente menos que la de un árbol RB por su factor de ramificación, por lo que la base del logaritmo que expresa su altura puede ser mucho mayor.

Los árboles B generalizan los árboles binarios en una forma natural. Si un nodo interno de un árbol B x contiene $x.n$ claves, entonces tendrá $x.n + 1$ hijos. Las claves en un nodo x sirven como puntos de división separando los rangos de claves manejados por x en $x.n + 1$ sub-rangos, cada uno manejado por un hijo de x. Cuando buscamos una clave en un árbol B, haremos una decisión entre $x.n + 1$ caminos basada en comparaciones con las $x.n$ claves guardadas en el nodo x.

Un árbol B tiene las siguientes propiedades:

1. Cada nodo x tiene los siguientes atributos:
   a) $x.n$, el numero de claves actualmente guardadas en el nodo x.
   b) Las $x.n$ claves, $x.key_1, x.key_2, ..., x.key_{x.n}$, guardadas en orden creciente de forma que $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$.
   c) $x.leaf$, un valor booleano que es verdadero si x es una hoja y falso si es un nodo interno.
2. Cada nodo interno x además contiene $x.n + 1$ punteros $x.c_1, x.c_2, ..., x.c_{x.n+1}$ a sus hijos. Los nodos hojas no tienen hijos, por lo que sus atributos c_i estarán indefinidos.
3. Las claves $x.key_i$ separan los rangos de las claves guardadas en cada sub-árbol. Si k_i es cualquier clave guardada en el sub-árbol cuya raíz es $x.c_i$, entonces $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq ... \leq x.key_{x.n} \leq k_{x.n+1}$.
4. Todas las hojas tienen la misma profundidad, la cual será la altura del árbol h.
5. Los nodos tienen cotas inferiores y superiores en el numero de claves que pueden contener. Expresaremos estas cotas en términos de un entero fijo $t \geq 2$ llamado el grado mínimo del árbol B:
   a) Cada nodo distinto a la raíz debe tener al menos $t - 1$ claves. Cada intervalo de nodos distinto que la raiz debe tener al menos t hijos. Si el árbol no esta vacío, la raíz debe tener al menos una clave.

b) Cada nodo debe contener como mucho $2t-1$ claves. Entonces, un nodo interno puede tener a lo sumo $2t$ hijos. Diremos que el nodo esta **lleno** si contiene exactamente $2t-1$ claves.

Arboles 234

El árbol B mas simple ocurre cuando $t=2$. Cada nodo interno puede tener 2, 3 o 4 hijos, por lo que tendremos un árbol $2-3-4$. En la practica, valores muchos mas grandes de t nos darán árboles B con una altura menor. Gracias a la noción de cotas podremos dar una cota para la altura de un árbol B de la forma $h \leq log_t((n+1)/2)$, en donde $n \geq 1$, para cualquier árbol B de $n-$claves de altura h y grado mínimo $t \geq 2$.

Inserción en Arboles 234

Durante la inserción cada vez que encontremos un 3-nodo x empujaremos al nodo del medio hacia su padre z y partiremos el 2-nodo restante en dos 1-nodo. Para nodos distintos de la raíz podremos hacer esto ya que el nodo z, por el cual ya habremos pasado, tendrá a lo sumo dos claves. Si es la raíz en la que nos encontramos, crearemos una nueva raíz a la que podamos pasar la clave del medio. Es importante remarcar que los nodos se partirán a medida que los descubrimos, si se creo un 3-nodo a partir de la acción de empujar una clave hacia su padre no haremos nada. Las razones por las cuales partimos los 3-nodos son asegurarnos de que haya lugar para la nueva clave en la hoja y para hacer lugar para cualquier clave que sea empujada hacia arriba. La única forma en la que el árbol incrementará su profundidad es creando una nueva raíz a causa de una inserción.

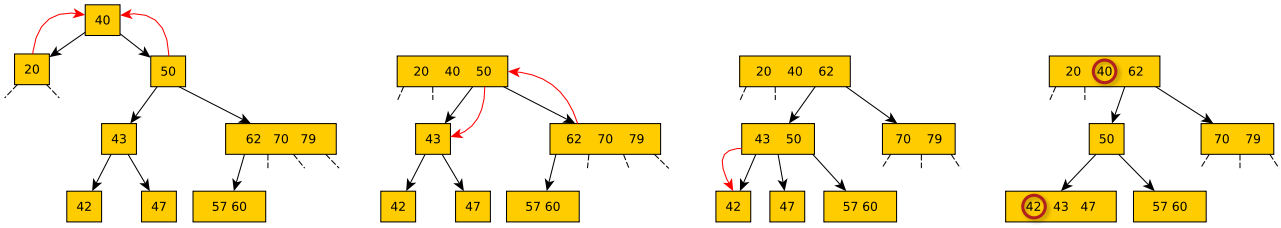
Eliminación en Arboles 234

Para la eliminación, en primer lugar buscaremos la clave a eliminar. Si la clave se encuentra en una hoja la removeremos del árbol y si es un nodo interno la "eliminaremos" reemplazándola con la siguiente clave mas alta (que se encontrará en un nodo hoja). Durante la eliminación se traerán claves desde el padre o los hermanos de forma tal de asegurarnos de tener al menos dos claves en el nodo donde se efectuara la eliminación (o de donde obtendremos el reemplazo). De cierta forma estaremos haciendo una estrategia inversa a la inserción donde simplemente empujábamos las claves.

A medida que busquemos la clave a eliminar o su reemplazo, nos encontraremos en nodos que contengan solo una clave que deberemos lograr que tengan dos (para que al remover alguna de ellas, no nos quede un nodo vacío). Para lograr esto tendremos distintas estrategias según los posibles tres casos en donde nos encontremos:

1. Intentar robar una clave de los hermanos inmediatamente adyacentes. Si alguno de sus hermanos tiene mas de una clave ejecutara una rotación para obtenerlo.
2. Si no hay hermano adyacente al que podamos robarle una clave, robaremos una clave al padre (esto sera posible al menos que sea la raíz) y fusionaremos la misma con el hermano siguiente u anterior (dependiendo de la desigualdad) formando un 3-nodo.
3. Si no hay hermano adyacente al que podamos robarle una clave, el padre es la raíz y tiene solo una clave, entonces fusionaremos en un 3-nodo los hermanos y la raíz, obteniendo así un nuevo 3-nodo raíz. Este sera el único caso en donde la altura del árbol se verá disminuida en una unidad.

Mostraremos los tres casos mediante un ejemplo al cual se le removerá la raíz.



Como queremos remover el 40, el objetivo sera reemplazarlo por el 42 que se encuentra en una de las hojas del árbol. Una vez parados en el 1-nodo que contiene la clave 50, veremos que aplica el tercer caso ya que el padre es la raíz, y el único hermano inmediato tiene solo una clave, por lo que haremos una fusión como se aprecia en la segunda figura. Luego descenderemos por la rama dirigida hacia el 1-nodo que contiene el 43, veremos que en esta instancia el primer caso es el que se ajusta, por lo cual realizaremos una rotación para terminar con un 2-nodo como se aprecia en

la tercera figura. Finalmente en la tercera figura habremos descendido por el nodo que contiene a 42 pero todavía no podremos usarlo para reemplazar a 40 ya que es un 1-nodo. En esta situación el caso que se ajustara será el segundo, ya que el único hermano inmediato es el 47, por lo que bajaremos el 43 y lo fusionaremos con el 47, llegando así a la cuarta figura en donde podremos eliminar la clave de la raíz y reemplazarlo por el 42.

Complejidades

Supongamos un Árbol 2-3-4 con altura d, entonces es fácil ver que dicho árbol tiene entre 2^d y 4^d hojas, por lo tanto la cantidad de nodos totales (lo llamaremos n) es $n \geq 2^{d+1} - 1$. Luego, aplicando logaritmo a ambos lados y dando vuelta la desigualdad, $d \in O(log\ n)$.

Ahora, el tiempo que tardamos en visitar cada nodo $\in O(1)$. Entonces, las operaciones sobre el árbol tienen como costo la cantidad de nodos que visitamos, o sea: $O(d) = O(log\ n)$.

- **Búsqueda** $O(lg(n))$
- **Inserción** $O(lg(n))$
- **Borrado** $O(lg(n))$
- **Espacio** $O(n)$

3.3.5. Splay trees

Este tipo de árbol tiene su inspiración en los ABB óptimos. Los ABB óptimos son árboles cuyos elementos están ubicados mas cerca o lejos de la raíz según la frecuencia con la que son consultados (similar a la codificación de Huffman), de esta forma optimizando los tiempos de las operaciones. El problema de este tipo de árboles son la rigidez, el desconocimiento de la frecuencia de las operaciones y la variabilidad que pueden sufrir estas ultimas. Los Splay Trees son ABB óptimos aproximados que se modifican según la frecuencia de los últimos elementos consultados, de esta forma mantendremos algo cercano a un ABB óptimo que se irá optimizando para los elementos que consultemos mas frecuentemente.

En estos árboles, los cuales utilizan exactamente la misma representación interna que un ABB común sin ningún agregado, todas las operaciones toman $O(lg(n))$ en promedio. Cuando nos referimos a en promedio no nos referimos al promedio de los ABB o al Quicksort, no hay aleatorización en los Splay Trees. De hecho, una operación simple puede tomar $\Theta(n)$ en el peor caso en donde n es la cantidad de elementos en el árbol. Lo que podemos garantizar entonces es que cualquier secuencia de k operaciones, comenzando con un árbol vacío, y con el árbol no creciendo mas que n elementos, entonces la secuencia a lo sumo toma $O(k \cdot lg(n))$ en el peor caso. Esto significa que podremos tener algunas pocas operaciones lentas pero la mayoría serán mucho mas rápidas, por lo que tendremos un balance en promedio de $lg(n)$ o mejor para cada operación. O dicho más precisamente, los costos amortizados por operación son de $lg(n)$.

Otra ventaja en cuanto a otras estructuras es que los Splay Trees son mucho mas fáciles de programar y además otorgan acceso mas rápido a los datos recientemente accedidos, es decir que de cierta forma los Splay Trees tienen "memoria" en este aspecto. Esto último puede darnos una mayor performance en comparación a los Árboles 234 si tenemos un árbol grande y solamente accedemos a algunos de los elementos ignorando la mayoría de ellos, ya que como estaremos recurriendo frecuentemente a los mismos elementos los accesos en el Splay Tree se darán en $O(1)$.

Los Splay Trees, al igual que otros árboles balanceados, mantienen su balance mediante rotaciones. El balance de los Splay Trees no es perfecto, es por eso que si bien no nos garantizan una cota de $lg(n)$ en los peores casos los algoritmos utilizados para mantener cierto balance son mucho mas sencillos y rápidos que en estructuras de datos balanceadas mas rígidamente. Además, no nos garantizaran una cota en el peor caso, pero si nos garantizan la cota en la mayoría de las operaciones, lo que en la practica y para la mayoría de los casos los hace igualmente buenos.

Optimalidad de los Splay trees

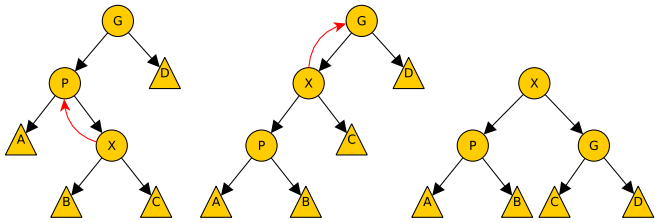
- Teorema de Optimalidad Estática: asintóticamente, los Splay Trees son tan eficientes como *cualquier* ABB fijo.
- Teorema de Optimalidad Dinámica: asintóticamente, los Splay Trees son tan eficientes como *cualquier ABB que se modifique* a través de rotaciones.

Búsqueda

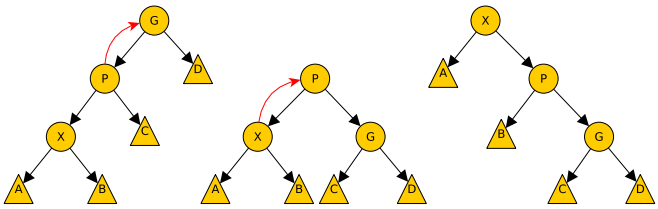
Dada una clave k, empezaremos a buscar la clave de la misma manera que lo hacemos en un ABB hasta que encontremos la clave k o hasta llegar a un NIL (en el caso de que la clave no sea existente). Sea x el nodo en donde la búsqueda terminó, contenga o no k, elevaremos x a través de una secuencia de rotaciones, de forma tal que x se vuelva la raíz, llamaremos a esta operación **splay**. Las razones por las cuales haremos esto serán, en primer lugar, para mantener los elementos mas consultados en la raíz o cerca de ella, de forma tal de que la consulta de estos sea verdaderamente rápida la próxima vez, y en segundo lugar, como este tipo de árboles puede desbalancearse (lo que es usualmente un estado temporal), si nos encontramos en un caso en donde exploramos hasta la profundidad del mismo, las rotaciones evitaran que esto pase de nuevo.

Las rotaciones se darán en tres casos:

1. **Zig-zag** X es el hijo izquierdo de un hijo derecho (LR) o su caso espejo (RL). Siendo P el padre de X y G su abuelo, para resolver este caso haremos una rotación hacia la izquierda entre P y X y luego una rotación hacia la derecha entre X y G. En el caso espejo realizaremos una rotación hacia la derecha y luego una hacia la izquierda, dejando a X como la raíz del sub-árbol.



2. **Zig-zig** Este caso tiene una diferencia muy sutil con el caso anterior. Si X es el hijo izquierdo de un hizo izquierdo (LL) o su caso espejo (RR). Siendo P el padre de X y G su abuelo, para resolver este caso haremos primero una rotación entre P y G hacia la derecha, y luego ejecutaremos otra rotación hacia la derecha entre X y P, dejando a X como la raíz del sub-árbol.



3. **Zig** Este es un caso final que se origina al tener a X a una distancia original impar de la raíz en el momento del acceso. Para solucionar esto, si X es el hijo izquierdo haremos una rotación hacia la derecha entre X y la raíz, o si X es el hijo derecho haremos una rotación hacia la izquierda.

Los casos 1 y 2 se repetirán hasta que X sea la raíz del árbol o un hijo de la raíz, lo que dará lugar al caso 3. En los casos donde tengamos un árbol totalmente degenerado hacia un lado (de forma tal que sea equivalente a una lista), la búsqueda nos costara tiempo $O(n)$ pero las rotaciones Zig-zig lograrán acortar la altura del árbol a la mitad, de forma tal de que las siguientes operaciones de consulta sean mucho mas eficientes. Esto ultimo se logra gracias a la forma de la rotación Zig-zig, la cual rota el abuelo y su padre antes que el hijo y su padre, si esto si hiciese en orden inverso obtendríamos un árbol totalmente degenerado hacia la dirección opuesta original.

Max/Min, Inserción y Eliminación

Las demás operaciones del árbol usaran la operación de splay de formas similares ya que es importante para el Splay Tree ejecutar un splay cada vez que realizamos una operación sobre el mismo. De esta forma, el árbol se mantiene actualizado y balanceado según las ultimas operaciones.

- En el caso de **max/min**, una vez encontrado la max/min clave del árbol, haremos un splay del nodo que lo contiene.

- En el caso de la **inserción**, una vez insertado el elemento haremos splay sobre el elemento.
- En el caso de la **eliminación**, una vez eliminado de la misma forma que haríamos en un ABB común, sea x el nodo eliminado del árbol, ejecutaremos splay sobre el padre de x. Si la clave que queríamos eliminar no existe, aun deberemos ejecutar un splay sobre algo, por lo que lo haremos donde termino la búsqueda, tal como en la primera operación.

3.3.6. Tries

Los Tries son arboles (K+1)-arios para alfabetos de K elementos, o lo que es lo mismo, cada nodo puede tener a lo sumo K+1 hijos. Su nombre viene de la palabra "Re**trie**ve".

Motivaciones

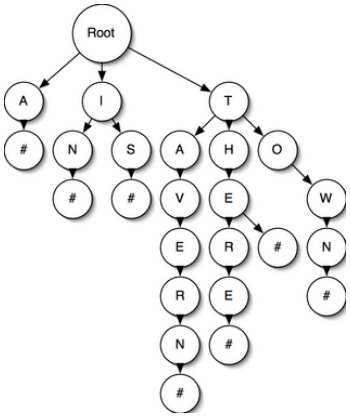
- Tiempo menos dependiente de la cantidad de claves.
- Rendimiento razonable en el peor caso.
- Rápidos en la práctica.
- Adecuados para claves de tamaño variable.

La idea principal del Trie es que los ejes del árbol representan componentes de las claves, y cada subárbol representa al conjunto de claves que comienza con las etiquetas de los ejes que llevan hasta él. De esta forma, se busca no hacer comparaciones de claves completas, sino de partes de ellas. Es decir, sí las claves son strings, trabajar con caracteres.

Haciendo ciertas suposiciones, la búsqueda o inserción en un árbol de n claves de b bits necesita en promedio $log\ n$ comparaciones de clave completa, y b en el peor caso.

Propiedades

- La estructura del trie es la misma independientemente del orden en el que se insertan las claves.
- A lo sumo una comparación de clave completa (cuando llega a una hoja).



Desventajas

- Algunas implementaciones pueden requerir mucha memoria.
- Las operaciones sobre las componentes de las claves pueden no ser fáciles, o ser muy ineficiente en algunos lenguajes de alto nivel.

Complejidades

Para un Trie construido a partir de n claves de b bits:

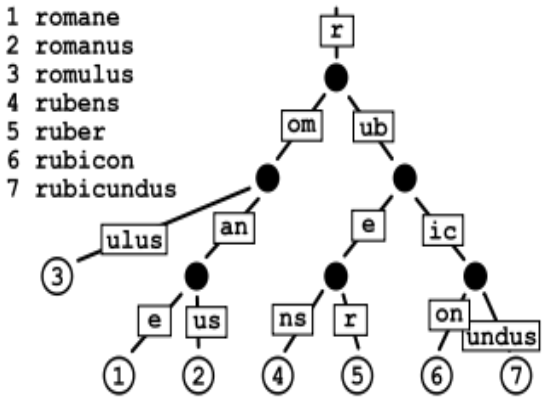
- **Búsqueda** $O(b)$, promedio (haciendo ciertas suposiciones) $\Theta(log\ n)$.
- **Inserción** $O(b)$, promedio (haciendo ciertas suposiciones) $\Theta(log\ n)$.

Tries compactos

Consiste en colapsar las cadenas de caracteres o bits que llevan hacia hojas. Ejemplo: en un trie con dos claves "Algo" y "Algoritmos", el camino que lleva desde la primer 'o' hasta la 's' puede ser compactado en un solo nodo para ahorrar espacio.

Tries Patricia

Son Tries más compactos que los mencionados anteriormente. *Patricia* es un acrónimo de: Practical Algorithm To Retrieve Information Coded in Alphanumeric. La idea es colapsar todas las cadenas, no solo las que llevan a hojas (un eje ahora puede representar una cadena). Ejemplo: en un trie con dos claves "Algoritmo" y "Algoritmos", el camino que lleva desde el nodo 'A' hasta la última 'o' puede ser compactado en un solo nodo para ahorrar espacio.



3.3.7. Red-Black

Los árboles RB son unos de los tantos esquemas en donde son "balanceados" para asegurar que las operaciones tomen $O(lg(n))$ en el peor caso. Un árbol RB es un árbol binario de búsqueda con un almacenamiento de un dato extra en cada nodo, su color, el cual puede ser rojo o negro. Restringiendo los colores de los nodos en cualquier camino simple desde la raíz a una hoja, los árboles RB se aseguran que no haya camino mas largo que el doble que cualquier otro, de esta forma el árbol esta aproximadamente balanceado.

Cada nodo del árbol ahora contiene los atributos color, clave, izquierda, derecha y p. Si un hijo o el padre de un nodo no existe, el puntero correspondiente al atributo del nodo contiene el valor NIL. Un árbol RB tiene las siguientes propiedades:

1. Todos los nodos son rojos o negros
2. La raíz es negra
3. Cada hoja (NIL) es negra
4. Si un nodo es rojo, entonces ambos hijos son negros.
5. Por cada nodo, todos los caminos simples desde el nodo a hojas de descendientes tienen el mismo numero de nodos negros.

Como conveniencia para lidiar con las condiciones borde en el código del árbol RB, usaremos un único centinela para presentar NIL. Para un árbol RB T, el centinela $T.nil$ es un objeto con los mismos atributos que cualquier otro nodo en el árbol. Si atributo de color es negro, y sus otros atributos tomaran valores arbitrarios. Usaremos este centinela de forma tal que podamos tratar un hijo NIL de un nodo x como un nodo ordinario cuyo padre es x. A pesar de que en cambio podríamos agregar un nodo centinela distinto para cada NIL en el árbol, de forma tal que el padre de NIL este bien definido, ese enfoque nos resultara en una perdida de espacio.

Llamaremos al numero de nodos negros de un camino simple desde x, pero no incluyéndolo, hasta una hoja la black-height de un nodo, denominada por $bh(x)$. Por la propiedad 5, la noción de black-height estará bien definida, dado que todos los caminos simples descendientes desde el nodo tendrán la misma cantidad de nodos negros.

Lema (cota de altura)

Un árbol RB con n nodos internos tiene como mucho altura $2 \cdot lg(n+1)$.

Prueba Empezaremos mostrando que el sub-árbol con su raíz en cualquier nodo x contiene al menos $2^{bh(x)} - 1$ nodos internos, esta sera nuestra hipótesis inductiva y probaremos la propiedad utilizando inducción en la altura de x.

Para el caso base, si la altura de x es 0, entonces x debe ser una hoja ($T.nil$), y el sub-árbol con su raíz en x efectivamente contiene al menos $2^{bf(h)} - 1 = 2^0 - 1 = 0$ nodos internos. Para el paso inductivo, consideraremos un nodo x que tiene una altura positiva y es un nodo interno con 2 hijos (recordar que estamos considerando a los NIL como hijos). Cada hijo tiene una black-height de $bh(x)$ o $bh(x) - 1$, dependiendo en si su color es rojo o negro, es decir si suma o no a la black-height. Dado que la altura de un hijo de x es menor que la altura de x en si mismo, podremos aplicar la hipótesis inductiva para concluir que cada hijo tiene al menos $2^{bh(x)-1} - 1$ nodos internos. Entonces, el sub-árbol con su raíz en x contiene al menos $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodos internos, lo que prueba la propiedad.

Para concluir la prueba del lema, sea h la altura del árbol y de acuerdo con la propiedad 4, al menos la mitad de los nodos en un camino simple desde la raíz a una hoja, sin incluir la raíz, deben ser negros (ya que no hay restricción sobre un nodo negro con un hijo negro, y para tener la mínima cantidad de negros deberán estar alternados con rojos). En consecuencia, la black-height de la raíz debe ser al menos $h/2$, entonces $n \geq 2^{h/2} - 1 \implies n + 1 \geq 2^{h/2}$. Si aplicamos logaritmo a ambos lados nos queda que $lg(n+1) \geq h/2 \implies h \leq 2 \cdot lg(n+1)$, lo que prueba el lema.

Búsqueda, máximo y mínimo

Como consecuencia directa del lema las operación de búsqueda, y por lo consecuente las de máximo y mínimo, tendrán tiempo $O(lg(n))$ en el peor caso, en un árbol RB.

Inserción

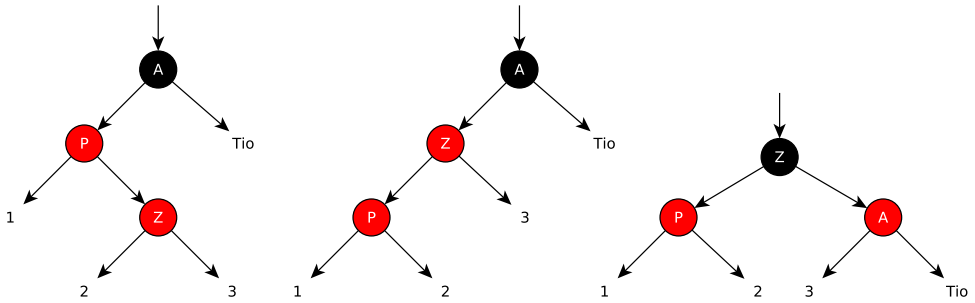
Podremos insertar un nodo nuevo en un árbol RB de n nodos en tiempo $O(lg(n))$. Para lograrlo, haremos algo muy parecido a la inserción del árbol binario de búsqueda. Insertaremos el nuevo nodo z tal como lo haríamos en un árbol binario de búsqueda normal, y luego le daríamos color rojo. Para garantizar que las propiedades del árbol RB se mantengan, realizaremos un procedimiento de reacomodación que recoloreará y ejecutara rotaciones sobre los nodos existentes.

En la inserción de z se mantendrán las propiedades 1 y 3, ya que no estaremos introduciendo un nuevo color y ya que z tendrá dos hijos NIL que estarán pintados de negro. Por lo que, las únicas propiedades que pueden llegar a violar serán la propiedad 2, que requiere que la raíz sea negra, y la propiedad 4, que dice que un nodo rojo no puede tener un hijo rojo. Ambas posibles violaciones son a causa de que z sea pintado rojo.

Deberemos considerar seis casos para la reacomodación, la cual se realizara en un ciclo hasta que el árbol sea correcto, pero tres de los seis casos son simétricos a los otros tres, por lo que realmente solamente necesitaremos considerar tres casos. Distinguiremos el caso 1 del caso 2 y 3 observando el color del hermano del padre de z, dicho de otra forma, el tío de z. Es aquí donde los 6 casos pasan a ser 3. Si $z.p$ es el hijo derecho, entonces el tío sera $z.p.p.izq$, de lo contrario sera $z.p.p.der$. Para trabajar sobre los 3 casos distintos, haremos de cuenta que $z.p$ es el hijo izquierdo, después de todo si es el derecho solamente habrá que hacer un cambio de palabras. Si el color del tío y es rojo, entonces ejecutaremos el caso 1, de otra forma el control pasa al caso 2 y 3. En los 3 casos, el abuelo $z.p.p$ es negro, dado que el padre $z.p$ es rojo, y la propiedad 4 es solo violada entre z y $z.p$.

- **Caso 1: El tío y de z es rojo** Como $z.p.p$ es negro, podremos pintar $z.p$ y y negros, solucionando el problema entre z y $z.p$, y podremos pintar $z.p.p$ de negro par mantener la propiedad 5. Deberemos repetir la reacomodación definiendo a $z = z.p.p$. El puntero de z se mueve dos niveles mas arriba en el árbol.
- **Caso 2: El tío y de z es negro y z es un hijo derecho** En este caso utilizaremos una rotación hacia la izquierda para transformar la situación en el caso 3, en donde z sera el hijo izquierdo. Dado que z y $z.p$ son rojos, la rotación no afecta el black-height de los nodos o la propiedad 5. z ahora sera $z.p$.
- **Caso 3: El tío y de z es negro y z es un hijo izquierdo** Si entramos en el caso 3 directamente o a través del caso 2, el tío y de z es negro, ya que de otra forma hubiésemos ejecutado el caso 1. En este caso ejecutaremos una rotación hacia la derecha entre el padre $z.p$ y su abuelo $z.p.p$, lo que dejará al abuelo de z como hermano

de z. Haciendo un intercambio de colores entre el padre de z y el ahora hermano de z, lograremos restablecer la propiedad 4 y además nos aseguramos de mantener la propiedad 5.



z, p y a representan el nodo nuevo z, su padre y su abuelo, respectivamente. En el primer momento estaremos en el caso 2 y una vez ejecutada la rotación pasaremos al segundo momento. Si bien no cambiaremos los nombres por claridad, el puntero z ahora estará apuntando a p. En este momento estaríamos en el caso 3, haciendo una rotación hacia la derecha mas pasaríamos al ultimo estado que cumplirá todas las propiedades.

Dado que la altura de un árbol RB de n nodos pertenece a $O(lg(n))$, la inserción principal del nuevo nodo z tomará tiempo $O(lg(n))$. Cuando ejecutamos la reacomodación, solo es necesario ejecutarla nuevamente cuando ocurre el primer caso, y cuando sucede el puntero de z se mueve dos niveles hacia arriba del árbol. Por lo tanto el numero de veces máximo que deberemos ejecutar la reacomodación sera $O(lg(n))$, por lo que la inserción tomará en total $O(lg(n))$.

Eliminación

Eliminar un nodo de un árbol RB puede ser un poco mas complicado que insertar un nodo. El procedimiento para eliminar un nodo de un árbol RB esta basado en la eliminación del árbol binario de búsqueda.

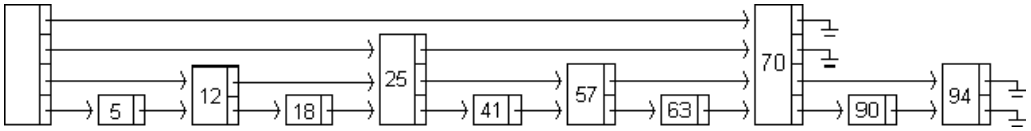
Complejidades

- **Búsqueda** $O(lg(n))$
- **Inserción** $O(lg(n))$
- **Borrado** $O(lg(n))$
- **Espacio** $O(n)$

3.4. Skip lists

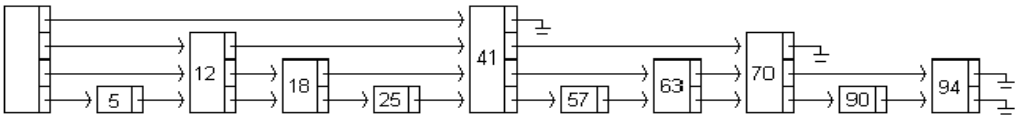
Como vimos en la sección de Estructuras Básicas, las listas son muy simples pero ineficientes. Implementar un Diccionario en dicha estructura nos daría complejidades $O(n)$ (búsqueda, inserción y borrado).

Las Skip lists buscan tomar ventaja de esta simplicidad, pero agregando una forma de avanzar "más rápido" los elementos de la lista. En vez de que cada nodo de la lista tenga un puntero al siguiente, tendrán múltiples punteros, dependiendo de su posición en la lista, lo que nos permitirá dar saltos más grandes a la hora de buscar/insertar un elemento. Llevado al límite, cada 2^i-ésimo nodo posee una referencia al nodo 2^i posiciones más adelante en la lista:



Lo que nos da una búsqueda en tiempo $O(log_2(n))$ (muy similar a la búsqueda binaria). Notese que $1/2$ de los nodos son de nivel 1, $1/4$ de nivel 2, y $1/2^i$ son de nivel 2^i. El problema principal de éste modelo es la rigidez en el caso dinámico (muy difícil de mantener en el caso de borrados), además de que el número total de punteros necesarios se duplica.

Para mejorar la idea anterior, podemos "aleatorizar" el requerimiento, y asignar el nivel de cada nodo probabilísticamente (por ejemplo, subiéndole el nivel mientras una moneda siga saliendo "cara"). De ésta forma, el costo de búsqueda (haciendo un análisis completo) sigue siendo $O(log(n))$ en promedio (pero sin hacer hipótesis probabilísticas sobre el input). Y se pueden mejorar las constantes con una moneda cargada: $p(cara) = 1/e$



Complejidades

- **Búsqueda**: promedio $O(log(n))$, peor caso $O(n)$
- **Inserción**: promedio $O(log(n))$, peor caso $O(n)$
- **Borrado**: promedio $O(log(n))$, peor caso $O(n)$
- **Espacio** promedio $O(n)$, peor caso $O(n * log(n))$

3.5. Tablas hash

Muchas aplicaciones requieren un conjunto dinámico que solo soporte las operaciones de diccionario inserción, búsqueda y eliminación. Una tabla de hash es una estructura de datos efectiva para implementar diccionarios, que generaliza el concepto de arreglo. Si bien buscar un elemento en una tabla de hash puede tardar tanto como buscar un elemento en una lista enlazada ($\Theta(n)$ en el peor caso), en la practica la tabla de hash tiene una muy buena performance. Bajo suposiciones razonables, el tiempo promedio de buscar un elemento en una tabla de hash es de $O(1)$.

3.5.1. Direccionamiento directo

Se asocia a cada valor de la clave un índice de un arreglo. Por lo tanto, tenemos búsqueda, inserción y borrado en $O(1)$. Por ejemplo, clave=número de DNI (número entre 0 y 10^8).

El único problema: **mucho** desperdicio de memoria.

3.5.2. Conceptos básicos

Representaremos un diccionario como una tupla $\langle T, h \rangle$. Donde:

- T es un arreglo con $N = tam(T)$ celdas.
- $h : K \to \{0, \ldots, N-1\}$ es la función hash (o de hash, o de hashing).
- K conjunto de claves posibles.
- $\{0, \ldots, N-1\}$ conjunto de las posiciones de la tabla (a veces llamadas *pseudoclaves*)
- La posición del elemento en el arreglo se calcula a través de la función h.

Se dice que una **función de hash es perfecta** sí: $\forall\ k_1 \neq k_2 \implies h(k_1) \neq h(k_2)$. Y ésto requiere $N \geq |K|$, lo que es raramente razonable en la práctica, ya que en general $N < |K|$, y muy habitualmente $N << |K|$

La consecuencia de esto último es que hay **colisiones** (y son mucho más frecuentes que lo intuitivo). Es decir, es posible que $h(k_1) = h(k_2)$ aún con $k_1 \neq k_2$. Para resolver esto se utilizan distintos métodos, que se diferencian por la forma de ubicar a los elementos que dan lugar a colisión, siendo las dos familias principales:

- Direccionamiento cerrado o Concatenación: a la iésima posición de la tabla se le asocia la lista de los elementos tales que $h(k) = i$
- Direccionamiento abierto: todos los elementos se guardan en la tabla.

Llamaremos P a la **distribución de probabilidad de las claves**, tal que $P(k)$ = "probabilidad de la clave k". Para disminuir la probabilidad de colisiones, nos gustaría que los elementos se distribuyan en el arreglo de manera uniforme. O lo que es lo mismo, pedir Uniformidad Simple:

$$\forall\ j \sum_{k \in K : h(k) = j} P(k) = 1/|N|$$

Pero es muy díficil construir funciones que satisfagan Uniformidad Simple, ya que P es generalmente desconocida.

3.5.3. Direccionamiento cerrado o Concatenación

A la i-ésima posición de la tabla se le asocia la lista de los elementos tales que $h(k) = i$. Por ejemplo, para una tabla de 5 celdas: $h(k) = k \ mod\ 5$

Complejidades

- **Búsqueda**: lineal en la lista asociada a la posición $h(k)$: O(longitud de la lista asocidada a $h(k)$)
- **Inserción**: al principio de la lista asociada a la posición $h(k)$: $O(1)$
- **Borrado**: búsqueda en la lista asociada a la posición $h(k)$: O(longitud de la lista asocidada a $h(k)$)

Luego, bajo la hipótesis de Uniformidad Simple de la función de hash, en promedio:

- Una búsqueda fallida requiere $\Theta(1 + \alpha)$

- Una búsqueda exitosa requiere $\Theta(1 + \alpha/2)$
- Con $n = \#$elementos en la tabla, y $\alpha = n/|T|$ (factor de carga).

Por lo que sí $n \sim T$, obtenemos $O(1)$

3.5.4. Direccionamiento abierto

A diferencia del Direccionamiento cerrado, todos los elementos se almacenan en la tabla, y las colisiones también se resuelven dentro de la tabla.

Ideas básicas:

- Sí la posición calculada está ocupada, hay que buscar una posición libre.
- La función de hash pasa a depender también del número de intentos realizados para encontrar una posición libre, es decir: $h(k, i)$ para el i-ésimo intento, y h debe generar todas las posiciones de T.
- Los distintos métodos de direccionamiento abierto se distinguen por el método de barrido que utilizan.
- Puede pasar que al buscar una posición libre nos de *overflow* (la tabla se llena y hay que borrar algún elemento antes de poder insertar uno nuevo).

Fenómenos de Aglomeración

- Aglomeración primaria: sí dos secuencias de barrido tienen una colisión, siguen colisionando, y los elementos se aglomeran por largos tramos. Ejemplo: $h(k, i) = (h'(k) + i)\ mod\ 101$, en una lista de 101 elementos, con la secuencia de inserciones: $\{2, 103, 104, 105, \dots\}$. Los elementos se aglomeran en un solo bloque.
- Aglomeración secundaria: sí dos secuencias de barrido tienen una colisión en el primer intento, sigue habiendo colisiones.

Barrido lineal

Para recorrer toda la tabla usamos:

$$h(k, i) = (h'(k) + i)\ mod\ |T|$$

donde $h'(k)$ es una función de hashing. Luego, se recorren todas las posiciones en la secuencia: $T[h'(k)], T[h'(k) + 1], \dots, T[|T|], 0, 1, \dots, T[h'(k) - 1]$. Tiene posibilidad de aglomeración primaria.

Barrido cuadrático

Usamos:

$$h(k, i) = (h'(k) + c_1 * i + c_2 * i^2)\ mod\ |T|$$

donde $h'(k)$ es una función de hashing, c_1 y c_2 constantes. Tiene posibilidad de aglomeración secundaria.

Hashing doble

Éste barrido depende también de la clave:

$$h(k, i) = (h_1(k) + i * h_2(k))\ mod\ |T|$$

donde $h_1(k)$ y $h_2(k)$ son funciones de hashing. Reduce los fenómenos de aglomeración secundaria, y no tiene aglomeración primaria.

Hashing dinámico o extensible

Capítulo 4

Algoritmos

4.1. Eliminación de la recursión

La motivación de la eliminación de la recursión es básicamente optimizar funciones recursivas mediante distintas formas. Para hacerlo se intentara cumplir ciertos objetivos como no repetir los cálculos (generalmente en problemas que presenten sub-estructura óptima), no recorrer una estructura mas de una vez (o intentar hacerlo la menor cantidad posible de veces) y finalmente eliminar la recursión, ya que si bien una función iterativa y una recursiva podrán tener complejidades asintóticamente iguales, en la practica las funciones iterativas son mas oṕtimas por diferencias de constantes (básicamente por no tener llamadas recursivas).

Inmersión de rango Agregar un nuevo parámetro de salida a la función, es decir si la salida original era un entero, ahora sera una dupla de enteros. La idea del nuevo parámetro que agregamos es darnos "algo mas" que lo estrictamente necesario de forma tal que nos sirva para calcular mas eficientemente otro resultado y así reducir la complejidad. Esta técnica también es conocida como **generalización de funciones**.

Inmersión de dominio En este caso agregaremos un parámetro de entrada que va conteniendo resultados intermedios. De esta forma podremos optimizar la complejidad de problemas que presenten subestructura óptima.

Inmersion de genero Se amplía el dominio de algún parámetro de la función. Por ejemplo de naturales a enteros.

Recursion a cola Esta sera una clase de funciones recursivas lineales en donde su ultimo paso es la llamada recursiva y además es única. La propiedad de estas es que pueden ser transformadas automáticamente a funciones iterativas.

Recursivas lineales no a cola Son aquellas que tienen una única llamada recursiva (por eso lineales) pero esta no es la ultima operación. Una función de este tipo puede ser aquella que tenga un condicional para separar entre el caso base y el caso recursivo. Este tipo de funciones pueden también ser convertidas automáticamente.

Recursivas no lineales Serán aquellas que tienen mas de una llamada recursiva. Pueden ser **múltiples** en donde contienen varias llamadas pero no es la ultima operación o **anidadas**, las cuales tienen mas de una llamada recursiva pero anidadas y además la llamada es la ultima operación.

Folding/Unfolding El desplegado consiste en reemplazar la definición de la función por su expresión. El plegado puede pensarse como la inversa de la anterior. Se trata de reemplazar la definición de una expresión por una llamada a función.

Precomputar valores Por ejemplo, sí mi programa va a intentar factorizar, tal vez me convenga tener almacenados los n primeros números primos.

Almacenar computos caros Sí ya computé resultados caros, puede tener sentido almacenarlos (en memoria o en disco) en lugar de recomputarlos.

4.2. Dividir y Conquistar

La metodología consiste en **dividir** el problema en un numero de subproblemas que son instancias mas chicas del mismo problema, **conquistar** los subproblemas resolviéndolos recursivamente y cuando sean suficientemente chicos resolverlos de una forma directa y **combinar** las soluciones de los subproblemas en la solución para el problema original.

Cuando los subproblemas son suficientemente grandes para resolverlos recursivamente, llamaremos al **caso recursivo**. Una vez que los subproblemas se vuelvan suficientemente chicos, diremos que la recursión "toco fondo" y que habremos llegado al **caso base**. A veces, en adición a los subproblemas que son instancias mas chicas del mismo problema, deberemos resolver problemas que no son exactamente lo mismo que el problema original. Consideraremos resolver dichos subproblemas como una parte del paso de combinación.

4.2.1. Recurrencias

Las recurrencias van a la par con el paradigma de DyC, ya que nos dan una forma natural de caracterizar los tiempos de los algoritmos que hagan uso del paradigma. Una recurrencia es una ecuación o inecuación que describe una función en términos de su valor en entradas mas chicas. Las recurrencias pueden tomar muchas formas. Por ejemplo, un algoritmo recursivo podrá dividir los subproblemas en tamaños desiguales, tales como 2/3 a 1/3. Si los pasos de dividir y combinar toman tiempo lineal, tal algoritmo nos dará la recurrencia $T(n) = T(2n/3) + T(n/3) + \Theta(n)$. Los subproblemas no están necesariamente acotados a ser una fracción constante del tamaño del problema original. Por ejemplo, si en cada paso disminuyéramos 1 solo elemento, nos quedaría una recurrencia del tipo $T(n) = T(n-1)+\Theta(1)$. Habrá 3 métodos para resolver recurrencias, esto es para obtener las cotas asintóticas "Θ o "O" de la solución:

- En el método de substitución, conjeturaremos una cota y luego la probaremos utilizando inducción matemática.
- En el método del árbol de recursión, convertiremos la recurrencia en un árbol cuyos nodos representaran los costos de varios niveles de la recursión. Usaremos técnicas para acotar sumatorias para resolver la recurrencia.
- El método maestro proveerá cotas para recurrencias de la forma $T(n) = aT(n/b) + f(n)$, en donde $a \geq 1$, $b > 1$ y $f(n)$ es una función dada, las recurrencias de este tipo se presentan muy frecuentemente. Para utilizar el método maestro será necesario memorizar tres casos, pero una vez hecho, podremos determinar cotas asintóticas muy fácilmente para recurrencias simples.

Ocasionalmente podremos ver recurrencias que no son igualdades sino que serán desigualdades, como lo es $T(n) \leq 2T(n/2) + \Theta(n)$. Dada que dicha recurrencia solo nos da noción de una cota superior en $T(n)$, daremos la solución con notación O en vez de Θ. De igual forma, con la desigualdad invertida $T(n) \geq 2T(n/2) + \Theta(n)$, la recurrencia solo nos dará la idea de una cota inferior de $T(n)$, por lo tanto usaremos Ω para dar su solución.

Método de substitución

El método de substitución para resolver recurrencias se compone de dos pasos. El primero de ellos es conjeturar una formula general de la solución y el segundo es usar inducción matemática para encontrar las constantes y mostrar que la solución funciona. Veamos este método mediante un ejemplo. Tomemos la recurrencia perteneciente a Merge Sort.

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

El caso base hace referencia a cuando nuestro arreglo tiene tamaño 1, en cuyo caso devolveremos el arreglo sin ejecutar ninguna operación sobre el, y el caso recursivo hace referencia a la recursión del arreglo sobre cada una de sus mitades (siendo así divide) y luego uniéndolo en tiempo n (siendo el conquer). En primer lugar vamos a acotar a n por el múltiplo de 2 mayor o igual mas cercano, quedando así $n \leq 2^k$ para algún $k \in N$. Luego, reescribiremos a $T(n)$ haciendo el reemplazo por k.

$$T(2^k) = \begin{cases} 1 & \text{si } k = 0 \\ 2T(2^{k-1}) + 2^k & \text{caso contrario} \end{cases}$$

Probemos alguno de sus valores y analicemos los resultados correspondientes para encontrar algún patrón que nos ayude a visualizar la formula general.

$$
\begin{aligned}
&T(2^0) = 1 \\
&T(2^1) = 2 \cdot T(2^0) + 2^1 = 2 + 2 && = 2^1 + 1 \cdot 2^1 \\
&T(2^2) = 2 \cdot T(2^1) + 2^2 && = 2^2 + 2 \cdot 2^2 \\
&T(2^3) = 2 \cdot T(2^2) + 2^3 && = 2^3 + 3 \cdot 2^3 \\
&T(2^4) = 2 \cdot T(2^3) + 2^4 && = 2^4 + 4 \cdot 2^4
\end{aligned}
$$

$$T(2^k) = 2^k + k \cdot 2^k$$

Probaremos que esta conjetura es valida utilizando inducción en N. El caso base es trivial, ya que $T(2^0) = 2^0 + 0 \cdot 2^0 = 0$, por lo que coincide con la definición. Luego para el paso inductivo, consideraremos que la formula vale hasta k (nuestra hipótesis inductiva) y queremos ver que vale para $k+1$ (nuestra tesis inductiva).

$$T(2^{k+1}) = 2T(2^k) + 2^{k+1} \overset{HI}{=} 2(2^k + k \cdot 2^k) + 2^{k+1} = 2^{k+1} + k \cdot 2^{k+1} + 2^{k+1} = 2^{k+1} + (k+1) \cdot 2^{k+1}$$

Por lo que nuestra tesis inductiva inductiva sera válida y la formula general que encontramos valdrá para todo $n \in N$. Como $n \leq 2^k \iff log(n) \leq k$ para algún k, tendremos que $T(n) \leq n + n \cdot log(n)$ y por lo tanto $T(n) \subseteq O(n \cdot log(n))$. De la misma forma, si utilizamos el múltiplo de 2 menor o igual mas cercano a n, obtendremos la misma cota pero perteneciente a $\Omega(n \cdot log(n))$, por lo que nos implicara que $T(n) \subseteq \Theta(n \cdot log(n))$.

La ultima formula puede ser generalizada cambiando sus constantes por variables.

$$T(2^k) = \begin{cases} c & \text{si } k = 0 \\ 2T(2^{k-1}) + f(2^k) & \text{caso contrario} \end{cases}$$

Luego, la conjetura quedaría de la forma $T(2^k) = 2^k \cdot c + \sum_{i=1}^{k} 2^{k-i} f(2^i) \leq 2^k \cdot c + k \cdot 2^k f(2^k)$. Si definimos a $f(n) = n$ y a $c = 1$, podremos observar que nos quedará $T(2^k) \leq 2^k + k \cdot 2^{k+1}$ la cual solo difiere en una constante ya que utilizamos una cota mas bruta, pero pertenecerá a la misma clase que la función anterior.

Método del árbol de recursión

Si bien podremos usar el método de substitución para proveer una prueba suficiente de que una solución a una recurrencia es correcta, tendremos problemas en encontrar una buena conjetura, algo en lo que nos ayudara el árbol de recursión.

En un árbol de recursion, cada nodo representa el costo de un solo subproblema en algún momento del conjunto de invocaciones recursivas, el nodo principal representará el costo inmediato inicial de la función (sin contabilizar la recursión) y los nodos adyacentes a el, es decir el primer nivel, representarán el costo inmediato de cada paso. Como hojas del árbol tendremos los casos base de la recursión. Para obtener nuestra conjetura, sumaremos los costos de cada uno de los niveles del árbol para obtener un conjunto de costos por nivel, y luego sumaremos todos los conjuntos por nivel para obtener el costo total de todos los niveles de la recursión.

Un árbol de recursión es el mejor método para generar una buena conjetura, que luego podremos verificar utilizando el método de substitución. Cuando utilizamos el método del árbol del recursión para generar una conjetura buena, por lo general es tolerable un leve grado de "descuido", dado que luego estaremos verificando la conjetura. Si somos suficientemente cuidadosos al dibujar el árbol de recursión y sumar los costos, podremos usar el árbol de recursión como una prueba directa para la solución de una recurrencia.

Método maestro

El método maestro provee una receta para resolver recurrencias que tengan la forma:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ a \cdot T(n/b) + f(n) & n > 1 \end{cases}$$

En donde $a \geq 1$ y $b > 1$ son constantes y $f(n)$ es una función asintóticamente positiva. Para usar el método maestro, habrá que memorizar tres casos, pero una vez hecho se podrán resolver recurrencias muy fácilmente, muchas veces sin la necesidad de usar lápiz y papel.

El método maestro tiene su base en el teorema maestro, el cual dice que siendo $a \geq 1$ y $b > 1$ constantes, $f(n)$ una función y $T(n)$ esta definido en los enteros no-negativos por la recurrencia anterior, entonces $T(n)$ tiene las siguientes cotas asintóticas:

- Si $f(n) \subseteq O(n^{log_b(a)-\epsilon})$ para alguna constante $\epsilon > 0$, entonces $T(n) \subseteq \Theta(n^{log_b(a)})$.
- Si $f(n) \subseteq \Theta(n^{log_b(a)})$, entonces $T(n) \subseteq \Theta(n^{log_b(a)} \cdot lg(n))$.
- Si $f(n) \subseteq \Omega(n^{log_b(a)+\epsilon})$ para alguna constante $\epsilon > 0$, y si existe alguna constante $c < 1$ y se cumple que $af(n/b) \leq cf(n)$ a partir de un n suficientemente grande, entones $T(n) \subseteq \Theta(f(n))$.

En cada uno de los tres casos, comparamos la función $f(n)$ con la función $n^{log_b(a)}$. Intuitivamente, la polinomialmente más grande de las dos funciones determina la solución a la recurrencia. En el primer caso, la función $n^{log_b(a)}$ es polinomialmente más grande, entonces la solución será $\Theta(n^{log_b(a)})$. En el tercer caso si $f(n)$ resulta ser la función polinomialmente más grande, entonces la solución será $T(n) \subseteq \Theta(f(n))$, la condición de regularidad en este caso esta presente para asegurarse de que $f(n)$ sea polinomialmente mas grande. Finalmente, en el segundo caso, las dos funciones son "del mismo tamaño", esto quiere decir que cualquiera puede ser utilizada como cota asintótica de la otra, se multiplicarán por un factor logarítmico y la solución será $T(n) \subseteq \Theta(n^{log_b(a)} \cdot lg(n)) \subseteq \Theta(f(n) \cdot lg(n))$. Es importante remarcar que el teorema maestro no cubre todos los casos de $f(n)$, por lo cual habrá algunos casos que caerán en las brechas entre alguno de los casos, puntualmente cuando $f(n)$ sea asintóticamente mas grande o mas chica que $n^{log_b(a)}$ pero no lo sea polinomialmente.

Para utilizar el teorema maestro simplemente debemos determinar a que caso pertenece la recurrencia (si es que lo hace) y escribir la respuesta. Algunos ejemplos de uso del teorema maestro son:

- En la recurrencia $T(n) = 9T(n/3) + n$ tendremos que $a = 9$, $b = 3$ y $f(n) = n$, por lo que tendremos que $n^{log_b(a)} = n^{log_3(9)} = n^2 \subseteq \Theta(n^2)$. Como $f(n) \subseteq O(n^{2-\epsilon})$ con $\epsilon = 1$, entonces podremos aplicar el primer caso del teorema maestro por lo que la solución a esta recurrencia será $T(n) \subseteq \Theta(n^2)$.
- En la recurrencia $T(n) = T(2n/3) + 1$ tendremos que $a = 1$, $b = 3/2$ y $f(n) = 1$, por lo que tendremos que $n^{log_b(a)} = n^{log_{3/2}(1)} = n^0 \subseteq \Theta(1)$. Como $f(n) \subseteq O(1)$ con $\epsilon = 0$, como $\epsilon > 0$ para que pueda pertenecer al primer caso, la solución a esta recurrencia se ajusta al segundo caso del teorema maestro, por lo que $T(n) \subseteq \Theta(lg(n))$.
- En la recurrencia $T(n) = 3T(n/4) + n \cdot lg(n)$ tendremos que $a = 3$, $b = 4$ y $f(n) = n \cdot lg(n)$, por lo que tendremos que $n^{log_b(a)} = n^{log_4(3)} = 0{,}79$. Con $epsilon \approx 0{,}2$ tendremos que $f(n) \subseteq \Omega(n)$, lo que se acerca al tercer caso del teorema maestro si podemos cumplir la condición de regularidad. Sea $af(n/b) \leq cf(n) \implies 3f(n/4) \leq cf(n) \implies 3nlg(n/4)/4 \leq cnlg(n)$ lo cual se cumplirá para todo $c \geq 3/4$ y para un n suficientemente grande. Esto nos dice que la solución de la recurrencia es entonces $T(n) \subseteq \Theta(nlg(n))$.
- En la recurrencia $T(n) = 2T(n/2) + n \cdot lg(n)$ tendremos que $a = 2$, $b = 2$ y $f(n) = n \cdot lg(n)$, por lo que tendremos que $n^{log_b(a)} = n^{log_2(2)} = n$. El tercer caso del teorema maestro parece ajustarse ya que $f(n) \subseteq \Omega(n)$ con $\epsilon = 0$. El problema es que ϵ debe ser mayor a 1, y en el segundo caso del teorema la cota superior esta por debajo de $f(n)$. El problema en este caso es que $f(n)$ es asintóticamente mas grande que n pero no lo es polinomialmente, por lo que la recurrencia cae en la brecha entre el segundo y tercer caso. En este caso el teorema maestro no decide nada respecto a la recurrencia.
- En la recurrencia $T(n) = 2T(n/2) + n$, que corresponde al merge sort, tendremos que $a = 2$, $b = 2$ y $f(n) = n$, por lo que tendremos que $n^{log_b(a)} = n^{log_2(2)} = n$. Luego, esto cae en el segundo caso del teorema ya que $f(n) \subseteq \Theta(n)$, por lo que $T(n) \subseteq \Theta(nlg(n))$.

4.2.2. En resumen

La metodología consiste en dividir un problema en problemas similares pero mas chicos, en resolver estos problemas menores hasta que podamos resolverlos de forma ADHOC y luego combinar las soluciones de los problemas menores para obtener la solución del problema original. Este método tiene sentido siempre y cuando la división y la combinación no sean excesivamente caras.

El esquema general de dividir y conquistar consiste en:

- Si X es suficientemente chico, entonces resolveremos el problema de forma ADHOC.

- En caso contrario, haremos una descomposición en sub-instancias $X_1, X_2, ..., X_k$, en donde para cada una tendremos $Y_i \leftarrow DC(X_i)$ sub-soluciones, que luego combinaremos las Y_i soluciones para construir la solución general para X.

4.3. Ordenamiento

4.3.1. Cota inferior para algoritmos basados en comparaciones

Una lista de n elementos distintos puede tener $n!$ permutaciones o formas de orden distintas. Luego, una cota arbitrariamente mayor para $n! \leq n \cdot n \cdot ... \cdot n = n^n$ y una cota arbitrariamente menor podrá ser $n! \geq n/2 \cdot n/2 \cdot ... \cdot n/2 = n/2^{n/2}$. Luego, ambas cotas perteneceran a $\Theta(nlog(n))$ si les aplicamos un logaritmo, lo que nos implicara que $log(n!) \in \Theta(nlg(n))$. Todas las decisiones estarán basadas en comparar claves, típicamente mediante ifs, es decir que están basadas en resultados booleanos. Un algoritmo de ordenamiento correcto debe generar una secuencia distinta de comparaciones booleanas para cada permutación distinta de $1..n$. Como habrá $n!$ diferentes permutaciones en una secuencia de n elementos, deberemos tener $n!$ secuencias distintas de comparaciones. Si un algoritmo hace $\leq d$ preguntas booleanas, generara $\leq 2^d$ secuencias distintas de respuestas booleanas, y por lo tanto:

$$n! \leq 2^d \implies lg_2(n!) \leq d \implies d \in \Omega(n \cdot lg(n))$$

No podremos preguntar d preguntas en menos de d tiempo, si suponemos un tiempo acotado para cada preguntar, por lo que el algoritmo gasta $\Theta(d)$ tiempo preguntando d preguntas. Por lo que todo algoritmo de ordenamiento basado en comparaciones toma en el peor caso $\Omega(n \cdot lg(n))$ tiempo. Algoritmos mas rápidos que estos deberán hacer decisiones de q-caminos para un q grande.

4.3.2. Insertion sort

Es un algoritmo de ordenamiento in-place (es decir que no consume demasiada memoria adicional para ordenar la lista) y corre en tiempo n^2 en el peor caso. El invariante de este algoritmo es que la lista S este ordenada. El algoritmo funcionara teniendo una lista S que comenzara vacía y una lista sin ordenar I de n elementos, en cada paso obtendremos un elemento de n y lo **insertaremos** ordenadamente en S, S e I pueden ser sub-arreglos del arreglo original. En el caso que tengamos n claves fuera de lugar, el tiempo es de $O(w)$ en donde w es el numero de "intercambios" a realizar. Si utilizamos un árbol balanceado de búsqueda como la estructura en S el algoritmo se volverá $O(n \cdot lg(n))$ aunque sera mas lento que otros algoritmos de búsqueda por las constantes ocultas del árbol balanceado.

4.3.3. Selection sort y Heap sort

Es un algoritmo de ordenamiento in-place (es decir que no consume demasiada memoria adicional para ordenar la lista) y corre en tiempo n^2 en el peor caso. El invariante de este algoritmo es que la lista S este ordenada. El algoritmo funcionara teniendo una lista S que comenzara vacía y una lista sin ordenar I de n elementos, en cada paso **seleccionaremos** el mínimo elemento de I y lo insertaremos al final en S, S e I pueden ser sub-arreglos del arreglo original.

Si cambiamos I por un min-heap, podremos realizar la operación de encontrar un elemento mínimo en tiempo $O(lg(n))$, dejando al algoritmo con tiempo total en $O(n \cdot lg(n) + n) \subseteq P(n \cdot lg(n))$, en donde la suma en el primer caso será causada por convertir el arreglo en un heap. Esta ultima modificación nos dará el algoritmo denominado como Heap sort, el cual también será un ordenamiento in-place ya que el árbol del heap podrá ser guardado en un arreglo. Una optimización importante para evitar algunas constantes a la hora de implementar el Heap sort es representar el mismo a la inversa de lo que normalmente lo haríamos en el arreglo, dejando la raíz del árbol en el final del mismo. Esto provocara que a medida que saquemos el mínimo elemento del arreglo, el mismo vaya a parar (por el comportamiento mismo de la eliminación del heap) al comienzo del arreglo, que por definición del tamaño del heap, estará fuera del mismo. Si queremos implementar esto mismo utilizando la representación clásica del Heap sort, deberemos invertir el orden del arreglo una vez finalizadas las n iteraciones.

4.3.4. Mergesort

Mergesort es el algoritmo de ordenamiento Divide and Conquer por excelencia. El mismo comenzara con una lista de I elementos, la cual **dividirá** a la mitad por un total de $\lceil lg(n) \rceil$ veces. Una vez llegado al caso base, el cual tendrá un arreglo trivialmente ordenado de 1 elemento, utilizara merge para combinar los resultados parciales de cada división y de esta forma obtener el resultado al problema original. La función de recursión para este algoritmo sera de $T(n) = 2 \cdot T(n/2) + \Theta(n)$, que por el teorema maestro siendo $a = 2$, $b = n/2$ y $f(n) = n$, nos quedara que $n^{lg_b(a)} = n^{lg_2(2)} = n^1$, por lo que se ajustara al segundo caso del teorema ya que $f(n) \subseteq O(n)$ lo que implicara que $T(n) \subseteq \Theta(n \cdot lg(n))$. Si bien Mergesort funcionara bien para listas y arreglos en el tiempo dado anteriormente, no es un algoritmo de ordenamiento in-sort, por lo que utilizara memoria adicional, lo que es para considerar ya que para arreglos habrá otras soluciones mejores en lo que respecta a espacio.

4.3.5. Quick sort

El Quick sort es un algoritmo muy estudiado, analizado y utilizado en la práctica, basado en Divide and Conquer. Éste algoritmo eligirá un elemento del arreglo (idealmente el mediano), llamado *pivot*, para luego separar el arreglo en dos mitades: los elementos menores al pivot por un lado y los mayores por el otro. Una vez *particionado* el arreglo, se hará una llamada recursiva de Quick sort sobre cada mitad.

Los costos de éste algoritmo son $O(n^2)$ en el caso peor, y $O(n * log(n))$ en el caso mejor y promedio. En la práctica el algoritmo es eficiente, pero la elección del pivot es fundamental.

Esto se puede ver ya que el caso peor (para obtener complejidad cuadrática) se da cuando en cada llamada recursiva elejimos como pivot al mínimo (o máximo) elemento del arreglo. Como contraparte, el caso mejor se da cuando para cada llamada recursiva elegimos como pivot al mediano del arreglo. Para el caso promedio, podríamos pensar en un input proveniente de una distribución uniforme. O sea, que la probabilidad de que el elemento i-ésimo sea el pivot es 1/n, en cuyo caso la complejidad del algoritmo nos quedaría $O(n * log(n))$. Pero no siempre podemos suponer que el input proviene de una distribución uniforme. Sí ese no es el caso, podemos forzarlo de alguna forma. Por ejemplo: permutando el input, o eligiendo el pivot al azar.

4.3.6. Bubble sort

4.3.7. Bucket sort

Funcionara cuando tengamos claves pequeñas en un rango, por ejemplo desde 0 hasta $q-1$, en donde $q \in O(n)$. Tendremos un arreglo de q colas, numeradas desde 0 a $q-1$, y encolaremos cada elemento de forma tal que la clave i vaya a la cola i. Una vez hecho esto concatenaremos las colas en orden, de forma tal que el fin de la cola i concatenado con el siguiente de la cola $i+1$. El tiempo del algoritmo sera de $\Theta(q+n)$, tomara $\Theta(q)$ en inicializar y concatenar cada una de las colas y $\Theta(n)$ en poner cada uno de los elementos en las colas. Si $q \in O(n)$, el tiempo total sera de $\Theta(n)$, si esta condición no se cumple deberemos utilizar algoritmos mas complicados para resolver esto.

```
1: function BUCKETSORT(A, q)
2:     L = array of q empty lists                          ▷ O(q)
3:     for all i ∈ [0, |A| − 1] do                         ▷ O(n)
4:         L[A[i].key].append(A[i])
5:     end for
6:     output = []
7:     for all i ∈ [0, q − 1] do                           ▷ O(q + n)
8:         for all j ∈ [0, |L[i]|] do                      ▷ O(|L[i]|)
9:             output[i + j] = L[i][j]
10:        end for
11:    end for
12: end function
```

Bucket sort es estable, esto quiere decir que elementos con las mismas claves se obtendrán en el mismo orden en el que ingresaron luego de que el arreglo sea ordenado por sus claves. Insertion, selection, mergesort pueden ser hechos fácilmente estables, al igual que quicksort con una lista enlazada (la versión con arreglo no lo es). Heapsort, por otro lado nunca es estable.

4.3.8. Counting sort

Si los elementos son claves y no hay información asociada, contaremos las copias de cada uno de los elementos utilizando por ejemplo un arreglo S. El arreglo comenzara lleno de ceros y a medida que contaremos iremos incrementando para la clave i la posición i del arreglo. Luego construiremos el arreglo final en tiempo $O(n)$. Al igual que bucket sort el algoritmo toma $O(q+n)$ tiempo, y si $q \in O(n)$ el tiempo total sera de $\Theta(n)$.

```
1: function COUNTINGSORT(A, q)
2:     count = empty array of q elements                   ▷ O(q)
3:     for all i ∈ [0, |A| − 1] do                         ▷ Calculate the frequencies of each key: O(n)
4:         count[A[i].key] += 1
5:     end for
6:     total = 0
```

```
    for all i ∈ [0, q − 1] do                ▷ Calculate the starting index of each key: O(q)
        oldCount = count[i]
        count[i] = total
        total+ = oldCount
    end for
    output = empty array of n elements                                          ▷ O(n)
    for all i ∈ [0, |A| − 1] do      ▷ Copy to output array, preserving order of inputs with equal keys: O(n)
        output[count[A[i].key]] = A[i]
        count[A[i].key]+ = 1
    end for
end function
```

4.3.9. Radix sort

Supongamos que queremos ordenar mil elementos que estarán en un rango de $[0, 10^8]$. En vez de proveer 10^8 buckets, proveamos 10 buckets, luego ordenaremos los elementos por el primer dígito solamente. De ésta manera, los buckets tendrán rangos: el primer bucket contendrá elementos desde 0 hasta 9999, el segundo desde 10000 hasta 19999, y así. Luego al obtener cada cola podremos ordenar cada una recursivamente por el segundo dígito, etc. De esta forma dividiremos en subconjuntos cada vez mas pequeños el problema original. El problema con esto es que los buckets mas chicos serán ordenados de forma ineficiente.

Una idea interesante es mantener los números en una pila mientras se ordena, y ordenar primero utilizando el dígito menos significativo y continuando hasta el dígito mas significativo. Esto funcionara ya que bucket sort es estable, y por lo tanto el orden relativo que le demos en una primera iteración se mantendrá en las iteraciones que le continúen. Podremos incrementar la velocidad utilizando mas cantidad de buckets, es decir ordenando de a 2 o 3 dígitos ($q = 10$ o $q = 100$). q sera la cantidad de buckets y además el "radix" de dígitos que usaremos como una clave de ordenamiento.

Cada iteración en un radix sort inspecciona $lg_2(q)$ bits de cada clave. Si utilizamos $q = 256$, estaremos explorando 8 bits de cada clave en cada iteración. Si utilizamos ints de 32 bits, necesitaremos realizar 4 pasadas para ordenar cualquier cantidad de números (si bien a mayor cantidad usaremos mucha mas memoria para realizarlo). Si todas las claves están representadas en b bits, la cantidad de iteraciones que necesitaremos sera de $\lceil \frac{b}{lg_2(q)} \rceil$. El tiempo de corrida del radix sort sera de $O((n+q) \cdot \lceil \frac{b}{lg_2(q)} \rceil)$.

El criterio para elegir q sera de forma tal que $q \in O(n)$, de forma tal que cada iteración tome tiempo lineal. Si hacemos q suficientemente grande, la cantidad de iteraciones sera chica, por lo que deberemos elegir q proporcional a n. Si hacemos esta elección la complejidad total de radix sort nos quedara de la forma $O(n + \lceil \frac{b \cdot n}{lg(n)} \rceil)$, si por otro lado podemos considerar a b como una constante, el ordenamiento tomara tiempo lineal. Por otro lado si queremos usar menos memoria podríamos usar q proporcional a $\sqrt{n}$, redondeada a la potencia de 2 mas cercana lo que incrementara la cantidad de iteraciones necesarias al doble pero decrementara la memoria que utilizamos de forma potencial.

4.4. Algoritmos para memoria secundaria

En ésta sección vamos a considerar los casos en los cuales trabajamos con memoria secundaria. Sí bien los problemas abstractos siguen siendo los mismos, el volumen de los datos hace que se encuentren necesariamente en memoria secundaria.

Las principales características de éstos dispositivos son:

1. El tiempo de acceso a los elementos es mucho mayor que en memoria principal (típicamente, 10^6 veces más lento). Por lo tanto, queremos minimizar la cantidad de veces que un elemento hace el viaje desde la memoria secundaria a la principal.

2. A su vez, el tiempo de acceso está dominado por el posicionamiento (más que la transferencia en sí misma), es decir, conviene acceder a bloques de datos simultáneamente.

3. Muchas veces, el manejo de memoria por parte del SO (caching) puede ser de buena ayuda sí se usa un método con buena "localidad de referencia".

4.4.1. Ordenamiento para memoria secundaria

Introducimos el concepto de **Ordenación-Fusión**:

Consiste en hacer varias pasadas sobre el archivo, dividiéndolo en bloques que entren en memoria interna, ordenando esos bloques y luego fusionando. Además, el acceso secuencial resulta apropiado para ese tipo de dispositivos. Se busca minimizar el número de pasadas sobre el archivo. Algunos métodos:

Fusión Múltiple Equilibrada: Supongamos que tenemos que ordenar los registros con valores: "EJEMPLO-DEORDENACIONFUSION". Los registros (o sus claves) están en una cinta, y solo pueden ser accedidos secuencialmente. En memoria hay sólo espacio para un número constante pequeño de registros, pero disponemos de todas las cintas que necesitemos (vamos a suponer capacidad en memoria principal para 3 registros). El algoritmo se desarrolla en varias pasadas:

- Mientras haya elementos en la cinta de entrada. Para i desde 1 hasta 3, hacer: leer 3 registros a la vez y escribirlos, ordenados, en la cinta i.

- Mientras haya bloques de 3 en las 3 primeras cintas, hacer: Fusionar los primeros bloques de cada cinta, armando un bloque de 9 (cada bloque en las cintas 1 a 3 tenía longitud 3) en las cintas 4, 5 y 6 alternadamente.

- Etcetera.

Complejidad (N registros, memoria interna de tamaño M, fusiones a P vías): La primera pasada produce aprox. N/M bloques ordenados. Se hacen fusiones a P vías en cada paso posterior, lo que requiere aprox. $log_P(N/M)$ pasadas. Por ejemplo: para un archivo de 200 millones de palabras, con capacidad en memoria para 1 millon de palabras, y 4 vías, no se requiere más de 5 pasadas.

Selección por sustitución: La idea en éste método es usar una Cola de Prioridad de tamaño M para armar bloques, "pasar" los elementos a través de una CP escribiendo siempre el más pequeño y sustituyéndolo por el siguiente. Pero, si el nuevo es menor que el que acaba de salir, lo marcamos como si fuera mayor, hasta que alguno de esos nuevos llegue al primer lugar de la CP.

Fusión Polifásica

4.4.2. Búsqueda para memoria secundaria

Acceso Secuencial Indexado (ISAM)
Árboles B
Hashing Extensible

4.5. Compresión

4.5.1. Codificación por longitud de series

La codificación por longitud de series esta basada en la redundancia de los caracteres en una serie. Podremos reemplazar una secuencia de repeticiones de un carácter por un solo ejemplar del mismo acompañado de la cantidad de repeticiones. Por ejemplo la secuencia $AAABBAABCDCCCD$ puede ser codificada como $4A3B2A1B1C1D3C1D$. A esta ultima, podremos mejorarla obviando los 1 y no cambiar las secuencias pares (ya que ocupan lo mismo) dejando así $4A3BAABCDCD$.

Para los archivos binarios se puede presentar una variante mucho mas eficiente, la misma se trata de almacenar solo la longitud de la secuencia, sabiendo que los caracteres están alternados. En archivos genéricos, necesitamos representaciones separadas para los elementos del archivo y los de su versión codificada. Por ejemplo, no habría forma de distinguir 47 64 de 4 764 si no tuviésemos el espacio. Si tenemos un alfabeto fijo (por ejemplo solo letras y el espacio en blanco), para lograr una diferenciación entre cadenas podremos usar un carácter que no aparece dentro de la misma como carácter de escape. Cada aparición indica que los siguientes caracteres son un par de $\langle longitud, caracter \rangle$, en donde la longitud estará representada por un carácter también en donde se traducirá según su posición en el alfabeto. Por ejemplo si tenemos la cadena QEA, siendo Q el carácter de escape, significara que la A se repetirá 5 veces, quedando la cadena decodificada como $AAAAA$. Si tenemos una serie mas larga que el alfabeto mismo podremos usar varas secuencias de escape, por ejemplo si tenemos 51 repeticiones del carácter A podremos codificarlo de la forma $QZAQYA$ $(26 + 25)$.

4.5.2. Códigos de longitud fija

Hay varias formas de representar la información de un archivo. Una de ellas es diseñar un código binario de caracteres en el cual cada carácter sera representado por un único string binario, el cual llamaremos simplemente código. Si usamos códigos de longitud fija en un alfabeto de 6 caracteres, necesitaremos 3 bits para representar esos 6 caracteres ($a = 000$, $b = 001$, ...). Suponiendo que tenemos un archivo con 100mil caracteres necesitaríamos 300mil bits para codificarlo.

4.5.3. Códigos de longitud variable

Un código de longitud variable reduce el espacio utilizado por los códigos de longitud fija asignándole códigos mas cortos a los caracteres mas usados y códigos mas largos a caracteres menos usados. De esta forma, utilizando el mismo texto en el ejemplo anterior con una codificación apropiada, podremos reducir de 300mil bits a 224bits, ganando así un 25 %. Sin embargo esta codificación lleva consigo un problema.

La codificación es simple para cualquier codificación binaria de caracteres (sea o no variable), dado que lo único que tenemos que hacer es traducir cada uno de los caracteres a su correspondiente código y concatenarlo. El problema se presenta a la hora de decodificar un archivo codificado. En la decodificación de un archivo codificado con códigos de longitud fija la tarea es trivial, ya que estaremos seleccionando una cantidad fija de bits para cada letra distinta. En cambio, si la codificación fue realizada con longitud variable, podríamos tener problemas al reconocer el código de un carácter si no tomamos recaudos al respecto. Una forma de lograr esto es agregar un carácter que indique una separación entre dos códigos, otra forma para lograr esto sin el uso de un separador son los códigos prefijos.

Los códigos prefijos simplifican la decodificación ya que ningún código es prefijo de ningún otro, de esta forma el código con el que comienza un archivo codificado no es ambiguo y por lo tanto nos desambigua el resto del archivo. Podremos simplemente identificar el código inicial, traducirlo al carácter correspondiente, y repetir el proceso de decodificación del resto del archivo codificado. El proceso de decodificación necesita una representación conveniente para el código prefijo de forma tal que fácilmente pueda obtener el carácter inicial. Un árbol binario cuyas hojas sean los caracteres dados nos dará dicha representación. Interpretaremos el código binario como un camino desde la raíz hasta el carácter, en donde 0 significara ir al hijo derecho y 1 significara ir al hijo izquierdo. Notar que estos no son árboles de búsqueda binarios, sino que serán mas bien equivalentes a tries binarios.

4.5.4. Códigos prefijos óptimos

Un código óptimo para un archivo es siempre representado como un árbol binario completo, en donde cada nodo que no sea una hoja tiene dos hijos. Por ejemplo, si una codificación contiene códigos comenzando en 10 pero ningún comenzando en 11, entonces no será óptimo. Si C es el alfabeto de donde todos los caracteres son obtenidos y todas

las frecuencias de aparición de los caracteres son positivas, entonces el árbol de prefijo óptimo tiene exactamente $|C|$ hojas, una para cada letra del alfabeto, y exactamente $|C| - 1$ nodos internos.

Dado un árbol T correspondiente a un código de prefijos, fácilmente podremos computar el numero de bits requeridos para codificar un archivo para cada carácter c en el alfabeto C. Sea el atributo $c.frec$ el que denote la frecuencia de c en el archivo y sea $d_T(c)$ una función que denota la profundidad de c en el árbol (notar que además es la longitud de bits utilizados para codificar c), el numero de bits requeridos para codificar un archivo sera:

$$B(T) = \sum_{c \in C} c.frec \cdot d_T(c)$$

Que definiremos como el costo de un árbol T.

4.5.5. Códigos de Huffman

Huffman invento un algoritmo goloso que construye un código prefijo óptimo o árbol prefijo óptimo llamado ćodigo de Huffman o árbol de Huffman. En el pseudocódigo a continuación, asumiremos que C es un conjunto de n caracteres y que cada carácter $c \in C$ es un objeto con el atributo $c.frec$ que nos informara de su frecuencia. El algoritmo construye el árbol de Huffman T correspondiente al código de Huffman de forma bottom-up. Comienza con un conjunto de $|C|$ hojas y realiza una secuencia de $|C| - 1$ operaciones de "merge" para crear el árbol final. El algoritmo utiliza una cola de prioridad Q ordenada por el atributo $frec$ en donde el elemento que tendrá mas prioridad será aquel con el valor mínimo. Q sera utilizada para identificar los 2 objetos menos frecuentes de forma tal de mergearlos. Cuando mergeamos dos objetos, el resultado es un nuevo objeto cuya frecuencia es la suma de las frecuencias de los dos objetos que fueron mergeados. En esta implementación los costos estarán calculados como si hubiésemos implementado la cola de prioridad con un heap.

```
1:  function CREARHUFFMAN(C)
2:      n = |C|
3:      Q = C                                                    ▷ O(n · lg(n))
4:      for all i ∈ [1, n − 1] do
5:          Nodo z
6:          z.izq = izq = popRemoveMin(Q)                        ▷ O(lg(n))
7:          z.der = der = popRemoveMin(Q)                        ▷ O(lg(n))
8:          z.frec = izq.frec + der.frec
9:          push(Q, z)                                           ▷ O(lg(n))
10:     end for
11:     return popMin(Q)                                         ▷ O(1)
12: end function                                                 ▷ O(2n · lg(n)) ⊆ O(n · lg(n))
```

La complejidad del algoritmo esta calculada groseramente ya que el único momento en el que el heap tendrá n elementos será cuando se agregue el ultimo elemento al mismo en la linea 3 o en el comienzo de la iteración en la linea 6.

Es importante remarcar la relación existente entre las repeticiones de los caracteres y la altura del árbol de Huffman. Un texto que contiene $|C|$ caracteres distintos será representado por un árbol de altura $\lceil log_2(|C|) \rceil$ como mínimo. Cuando decimos como mínimo hacemos referencia al caso cuando todos los caracteres tienen la misma frecuencia. Es interesante remarcar que el la longitud del árbol en el caso mínimo coincide con la cantidad de bits necesarios para una codificación de longitud fija de $|C|$ elementos, lo cual tiene sentido ya que al ser todas las frecuencias iguales los caracteres terminarán con la misma longitud en su codificación.

A medida que haya caracteres que aparezcan con distinta frecuencia que otros, el árbol se "degenerará" y por lo consecuente su altura se extenderá, llegando esta pudiendo ser $|C| - 1$ en el caso de que el árbol este totalmente degenerado hacia un lado. Esto ocasionara que los códigos de los caracteres sean de la forma $c_1 = 0, c_2 = 10, c_3 = 110, ..., z_n = 111..,0$, es decir que si tenemos la lista de caracteres ordenados de forma descendente por la frecuencia, el carácter i tendrá un código de longitud i (y se encontrará a distancia i de la raíz en el árbol), tendrá $i - 1$ unos precedentes y un 0 al final de su código. Esto último tiene un caso análogo en donde intercambiamos los ceros por unos.

La degeneración total hacia un solo lado de un árbol de Huffman será provocada si se encuentra cierta relación entre las frecuencias de los caracteres. Sea C la lista de caracteres ordenada por sus frecuencias de forma ascendente en donde c_i es el carácter en la posición i y sea $f : N \to N$ una función definida de la forma $f(i) = c_i.frec$, para

que el árbol sea totalmente degenerado se debe cumplir que $f(i-1)+f(i-2)<f(i)$ para todo $i \in [3,n]$ en donde $n=|C| \geq 3$. Las frecuencias de los caracteres al menos valen 1, ya que de lo contrario el carácter no esta presente en el texto a codificar. Es interesante recalcar que si tomamos un árbol totalmente degenerado con los mínimos valores de frecuencias posibles la sucesión conformada por estos deberá coincidir con la sucesión de Fibonacci, esto es la sucesión $f(1), f(2), ..., f(n)$ deberá coincidir $f(i)$ con $fib(i)$ para cada $i \in [1,n]$.

Métodos ZL