ALGORITMOS Y ESTRUCTURAS DE DATOS

Tomas Lisazo

2023

Apuntes y resumen de la materia

Indice

1 Lógica

1.1 Lógica proposicional

La lógica proposicional es un sistema de lógica basado en fórmulas estructuradas con ciertos simbolos, siguiendo una semantica y una sintactica clara, y que indica el valor de verdad de la misma. El objeto mas basico de la lógica proposicional es una fórmula atomica, que solemos notar con letras.

Por ejemplo, p puede ser un predicado, y puede significar *esta lloviendo*. En particular, p puede ser cierto o puede no serlo, propiedad que describimos como *valor de verdad*. Este valor de verdad puede ser o bien $True$ (verdadero, en ingles), o bien $False$ (falso, en ingles). Continuando con el ejemplo, si ahora mismo esta lloviendo, entonces p es verdad, tiene un valor de verdad de $True$. Por otro lado, si no esta lloviendo cuando usted está leyendo esto, entonces p sera falso, y tendria un valor de verdad de $False$.

Nosotros vamos a trabajar con una serie de simbolos que nos van a permitir unir fórmulas atomicas para armar estructuras mas complejas. Especificamente, veremos:

- o lógico
- y lógico
- Implicacion
- Doble implicacion
- Negación lógica

A su vez, estos operadores tienen una logica muy basica, basada en las tablas de verdad, que nos indican como funcionan y alteran los valores de verdad en cada caso. Las tablas son, dados p y q fórmulas cualesquiera:

Negación

p	$\neg p$
T	F
F	T

o lógico

p	q	$p \vee q$
T	T	T
F	T	T
T	F	T
F	F	F

y lógico

p	q	$p \vee q$
T	T	T
F	T	F
T	F	F
F	F	F

Implicacion

p	q	$p \longrightarrow q$
T	T	T
F	T	T
T	F	F
F	F	T

Doble implicacion

p	q	$p \longleftrightarrow q$
T	T	T
F	T	F
T	F	F
F	F	T

Con estas simples reglas, podemos formar fórmulas como $(p \wedge q) \longrightarrow r$ (leasé: *los valores de verdad de p y q en conjunto, implican la veracidad de r*).

Nos interesan mucho, en particular, lo que son las equivalencias. En particular, ser capaces de determinar si dos o mas fórmulas son equivalentes entre si. Decimos que esto sucede, cuando tienen exactamente los mismos valores de verdad. La equivalencia se suele notar con la doble implicacion. De este modo, podemos decir cosas como:

- $(p \vee p) \longleftrightarrow p$
- $(p \wedge p) \longleftrightarrow p$
- $(p \vee q) \vee r \longleftrightarrow p \vee (q \vee r)$
- $(p \wedge q) \wedge r \longleftrightarrow p \wedge (q \wedge r)$
- $(p \vee q) \longleftrightarrow (q \vee p)$
- $(p \wedge q) \longleftrightarrow (q \wedge p)$
- $p \wedge (q \vee r) \longleftrightarrow (p \wedge q) \vee (p \wedge r)$
- $p \vee (q \wedge r) \longleftrightarrow (p \vee q) \wedge (p \vee r)$
- $\neg(p \wedge q) \longleftrightarrow \neg p \vee \neg q$
- $\neg(p \vee q) \longleftrightarrow \neg p \wedge \neg q$
- $(p \longrightarrow q) \longleftrightarrow (\neg p \vee q)$

Todas estas son propiedades super utiles y practicas, asi que recomiendo tenerlas a mano.

Existe otra cuestion mas, que son las valuaciones. Una valuacion es una funcion $v : \nu \longrightarrow \{T, F\}$, es decir, toma como entrada una fórmula, y devuelve su valor de verdad. Lo notamos $v \models p$, que se leeria como $v(p) = T$. En caso contrario, diriamos $v \not\models p$, que significa $v(p) = F$.

De aca, las reglas antes mencionadas son deducibles, pero aca estan explicadas para mas accesibilidad:

- $v \models \neg p$ si y solo si $v \not\models p$
- $v \models (p \vee q)$ si y solo si $v \models p$ o $v \models q$
- $v \models (p \wedge q)$ si y solo si $v \models p$ y $v \models q$
- $v \models (p \longrightarrow q)$ si y solo si $v \not\models p$ o $v \models q$
- $v \models (p \longleftrightarrow q)$ si y solo si ($v \models p$ si y solo si $v \models q$)

Llamamos *tautologías* a aquellas fórmulas lógicas p que para toda valuacion v, se cumple $v \models p$. A su vez, existen las *contradicciones*, las cuales son todas las fórmulas lógicas p para las que todas las valuaciones v se cumple que $v \not\models p$.

Analogamente, decimos que una fórmula lógica es *satisfacible* si existe por lo menos una valuacion v tal que $v \models p$. Finalmente, decimos que una formula es *insatisfacible*, cuando no es satisfacible.

1.2 Lógica trivaluada secuencial

Existen ciertas cuestiones que no podemos representar en un modelo de $verdadero$ y $falso$. Por ejemplo, si tenemos un predicado $P(n)$, que dice que dado un numero n, p resulta cierta si y solo si $n = 2$.

Existe un problema, y es que no siempre podemos caer en numeros que podamos comparar. Si nos ingresan numeros naturales, o incluso reales, o yendo al caso, hasta complejos, no tendremos ningun problema. También podrian ingresar fórmulas matematicas, y no deberia haber ningun contratiempo a primera vista, pero si que lo hay. Vean en la tabla lo que sucede en el ultimo caso: estamos dividiendo por 0, lo cual esta prohibido en las matematicas. O bueno, mas que prohibido, no definido, porque lleva a contradicciones.

n	$P(n)$
1	F
2	T
3	F
4 / 2	T
1 / 0	???

Es para este tipo de casos que necesitamos ampliar nuestro sistema lógico. Evidentemente, no podemos abarcar todas las fórmulas simplemente con $verdadero$ y $falso$, asi que se creo un sistema nuevo: la lógica trivaluada, que como dice su nombre, tiene tres valores de verdad. Ahora vamos a contar con $verdadero$ y $falso$, como siempre, pero también con $indefinido$, notado a veces por el simbolo $\perp$.

Ahora, podemos definir la operacion de antes, resultando en que $P(1/0)$ es $indefinido$. A causa del nuevo valor, tenemos que actualizar las tablas de verdad, y revisitar las reglas de los simbolos para que todo tenga coherencia con lo antes dicho. Para esto mismo, vamos a introducir el concepto de *lógica trivaluada secuencial*.

La primera parte ya la conocemos, es la lógica trivaluada de la que venimos hablando. Pero la ultima parte, la de *secuencial*, es la que mas interesante vuelve todo esto. Ahora ya no evaluaremos las fórmulas de manera conmutativa, de ahora en adelante, el orden de las operaciones nos sera sumamente

relevante. Vamos a tener nuevos simbolos que se tengan que leer estrictamente de izquierda a derecha, que notaremos con una L pequeña junto a los simbolos clasicos, y llamaremos *luego*. Por ejemplo, al *o lógico* nuevo, lo llamaremos *o luego*, y lo notaremos $\vee_L$.

Las tablas revisitadas quedarian:

o lógico

p	q	$p \vee_L q$
T	T	T
F	T	T
T	F	T
F	F	F
T	$\bot$	T
F	$\bot$	$\bot$
$\bot$	T	$\bot$
$\bot$	F	$\bot$
$\bot$	$\bot$	$\bot$

y lógico

p	q	$p \wedge_L q$
T	T	T
F	T	T
T	F	T
F	F	F
T	$\bot$	$\bot$
F	$\bot$	F
$\bot$	T	$\bot$
$\bot$	F	$\bot$
$\bot$	$\bot$	$\bot$

1.3 Sistema deductivo

Ahora bien, todo excelente con este sistema de lógica y demas, pero no nos sirve de nada si no podemos relacionar fórmulas diferentes. Es decir, si siempre hablamos de cosas independientes, no vamos a llegar muy lejos. Es por eso que existen los *sistemas deductivos*, en los que buscamos *probar* una fórmula en particular, a la que llamaremos *conclusión*, en base a unas otras, a las que llamaremos *premisas*.

Una forma de notar todo esto es la siguiente:

$$\frac{p_1 p_2 \dots p_n}{q} \textit{ Regla deductiva}$$

Donde el conjunto $p_1 p_2 \dots p_n$ son las n premisas, y q es nuestra conclusión. A su vez, en lugar de *regla deductiva*, nosotros notariamos el nombre de la regla utilizada.

Llamaremos prueba a una secuencia de reglas lógicas aplicadas sucesivamente para llegar a alguna conclusión. En base a esto, deducimos el *secuente*, que notamos:

$$p_1, p_2, \dots, p_n \vdash q$$

Leasé el conjunto de premisas $\{p_1, p_2, \dots, p_n\}$ con el que podemos obtener una prueba de q. Consideramos valido al secuente si podemos construir de manera satisfactoria esa prueba.

Ahora que aclaramos todo esto, podemos pasar a lo pesado. Resulta que la eleccion de estas reglas lógicas es muy importante, ya que solo deberiamos ser capaces de construir pruebas validas. Es decir, no podemos permitirnos la posibilidad de satisfacer pruebas a conclusiones erroneas.

En particular, y de forma muy resumida, tenemos las siguientes reglas:

$$\frac{}{p}\ Hyp$$

$$\frac{p \quad q}{p \wedge q}\ \wedge i$$

$$\frac{p \wedge q}{p}\ \wedge e$$

$$\frac{p}{\neg\neg p}\ \neg\neg i$$

$$\frac{\neg\neg p}{p}\ \neg\neg e$$

$$\frac{p}{p \vee q}\ \vee e$$

$$\frac{p \quad p \longrightarrow q}{q}\ \longrightarrow e$$

$$\frac{\begin{array}{|c|}\hline p_n \\ \vdots \\ q \\ \hline\end{array}}{p \longrightarrow q}\ \longrightarrow i, n$$

Donde i significa *introduccion* y e *eliminacion*. Estos son solo algunas de las reglas que existen en casi cualquier sistema deductivo. Hay muchas mas que se pueden deducir de estas o incluso crear algunas mas especificas, como el *Modus tollens*.

Dandole un cierre a todo esto, vamos a definir unos terminos más. Llamaremos *teorema* a toda fórmula lógica p tal que el secuente $\vdash p$ es válido.

Nota del autor: Son todas muy intuitivas y, en general, cosas que ya se saben, simplemente se les pusieron nombres rimbombantes. Recordar no entrar en panico a la hora de intentar decifrar los simbolos raros de todo este capitulo.

1.4 Lógica de primer orden

Finalmente, llegamos a la parte final: la *lógica de primer orden*, o la lógica de predicados. Empecemos por un ejemplo, que va a ser mas fácil. Consideremos la frase:

Todo estudiante es más joven que algún profesor

Si siguieramos con la lógica proposicional seguro llamariamos a la frase p, o cualquier otra letra de preferencia, y le asignariamos algun valor de verdad. Pero quizas con eso perdemos informacion, podriamos armar expresiones mas pequeñas, que engloben un concepto muy especifico, y reconstruir la fórmula original mas compleja. Por mencionar una forma de hacerlo, ya que hay infinitas, aca mostramos:

Ser estudiante

Ser profesor

Ser más joven que

Podemos plantearlo como funciones. Vamos a llamar a nuestras variables *individuos*, es decir, entidades indivisibles y distintiva. En nuestro ejemplo, los *individuos* serian los estudiantes y los profesores, o las personas yendo al caso.

A su vez, vamos a tener *predicados*, que son funciones que toman por entrada individuos, hacen alguna operacion y devuelven el estado de verdad de la operacion hecha. Continuando con la frase, podemos tener un *predicado* que sea $E(x)$, que toma individuos x y nos informa si ese individuo es un estudiante o no. Por otro lado, podemos tener un $P(x)$, que de manera semejante nos informa si el individuo x es un profesor. También podemos tener predicados mas complejos, que tomen dos o mas entradas, por ejemplo podriamos definir $J(x, y)$, que dado un individuo x y un individuo y, nos indica si x es mas joven que y.

Por ultimo, vamos a introducir el concepto de los cuantificadores. Estos son simbolos que nos van a hacer mas facil la escritura de las fórmulas. En particular, tenemos dos: el *para todo*, que notamos $\forall$; y el *Existe* (también llamado *para algún*), y que notamos $\exists$.

Con todo esto en la mesa, vamos a poder constuir estructuras complejas como la frase de antes, sin perder ninguna informacion primordial. Utilizando nuestros ejemplos de antes, una fórmula equivalente a la frase podria ser:

$$\forall x(E(x) \longrightarrow (\exists y(P(y) \wedge J(x, y))))$$

Esta no es la unica fórmula que se puede constuir y tener el mismo significado. Podriamos haber usado otros prodicados, o ordenarla de manera diferente, el punto es que ahora la frase es inequivoca y no hay informacion que se pueda perder, que es lo que mas nos interesa.

En general, a la hora de usar cuantificadores, vamos a escribirlos *tipados*. Esto quiere decir, que vamos a requerir que la variable sea de algun tipo especifico. Lo notamos:

$$\forall x : T \qquad\qquad \exists x : T$$

Que se leerian como *para todo x del tipo T*, y *existe algún x del tipo T*, respectivamente. Si llamamos H al tipo *humano*, la fórmula original podriamos corregirla:

$$(\forall x : H)(E(x) \longrightarrow ((\exists y : H)(P(y) \wedge J(x, y))))$$

Existen un par de reglas utiles a la hora de trabajar con los cuantificadores. Primero es lo que sucede al usar la negacion en los mismos. Resulta que, si negamos un cuantificador universal, obtenemos un cuantificador existencial pero con el predicado negado, y viceversa. Todo esto quiere decir que:

$$\neg(\forall n)(P(n)) \longleftrightarrow (\exists n)(\neg P(n))$$
$$\neg(\exists n)(P(n)) \longleftrightarrow (\forall n)(\neg P(n))$$

Lo cual es una regla muy util y practica, pero no la unica. También hay formas de generalizar fórmulas: en particular, decimos que un cuantificador universal generaliza la conjuncion, y un cuantificador existencial generaliza la disyuncion. Es decir:

$$(\forall n : \mathbb{Z})(P(n)) \longleftrightarrow P(1) \wedge P(2) \wedge P(3) \dots$$
$$(\exists n : \mathbb{Z})(P(n)) \longleftrightarrow P(1) \vee P(2) \vee P(3) \dots$$

En el mundo de la lógica de primer orden, existen unas claras reglas sintacticas que todo LPO (lenguaje de primer orden) debe seguir. En particular, requiere:

- Un conjunto F de *símbolos de función* cada uno con aridad (cantidad de argumentos) $n > 0$: f_1, f_2, f_3, ..., f_n
- Un conjunto numberable C de *constantes*: c_0, c_1, ...
- Un conjunto P de *símbolos de predicado* cada uno con aridad $n \geq 0$: p_1, p_2, p_3, ..., p_m, $\doteq$

El simbolo $\doteq$ denotará igualdad.

A su vez, llamamos *terminos* al conjunto de todas las constantes y variables dentro del LPO. Las fórmulas atomicas son todo símbolo de predicado con aridad 0. Finalmente, les decimos fórmulas a todas las fórmulas atomicas dentro del LPO, y todas las que puedas formar con las operaciones lógicas antes vistas.

Combinando esto con los cuantificadores, tenemos los conceptos de *variable libre* y *variable ligada*, que a primera vista no parecen ser de mucho interes pero resultan ser mas que utiles. En particular, llamamos *variable ligada* a una ocurrencia de x si forma parte de una fórmula del tipo $\forall x$ o $\exists x$. Contrariamente, decimos que x es una *variable libre* si no es ligada.

Existe una operacion mas que todavia no mencionamos: la sustitución. Dada una variable x, un término t, y una fórmula p, notamos $p\{t/x\}$ a la fórmula que se obtiene de reemplazar cada ocurrencia libre de x en p por t (evitando capturas). Por ejemplo, algunas aplicaciones serian:

- $P(x, y)\{x/w\} = P(w, y)$
- $(\exists z(P(x, y)))\{x/w\} = \exists z(P(w, y)))$
- $(\exists x(P(x, y)))\{x/w\} = \exists x(P(x, y)))$

Ahora bien, dado cualquier lenguaje de primer orden L, su estructura, llamemosla $\mathtt{M}$, es un par

$$\mathtt{M} = (M, I)$$

donde M es un conjunto no vacío al que llamamos *universo*; e I es la *función de interpretación*, que es la encargada de asignar funciones y predicados sobre M a símbolos de L, siguiendo las siguientes reglas:

- Para toda constante c, se tiene que $I(c) \in M$
- Para toda f de aridad $n > 0$, se tiene que $I(f) : M^n \longrightarrow M$
- Para todo predicado P de aridad $n \geq 0$, se tiene que $I(P) \subseteq M$
- $I(\doteq)$ es la relación de identidad sobre M

Con esto de la estructura podemos terminar de definir conceptos importantes para mas adelante. En particular, dado un lenguaje L y su estructura $\mathtt{M}$, decimos que s es una *asignación* si s es una función $s : V \longrightarrow M$. A su vez, notamos como $s \models \mathtt{M}\ p$ al hecho de que la asignación s satisface la fórmula p bajo la estructura $\mathtt{M}$.

Notemos que p podria ser cualquier predicado, es decir, que podemos cambiar en la fórmula anterior cualquier estructura lógica. Podriamos decir $s \models \mathtt{M}\ p \wedge q$, que significa *$s$ satisface tanto p como q, bajo la estructura de primer orden* $\mathtt{M}$.

Finalmente, decimos que una fórmula p es válida o verdadera en $\mathtt{M}$ si y solo si se verifica que $s \models \mathtt{M}\ p$ para toda asignación s.

Para terminar este capitulo, mostraremos unas reglas algo complejas pero muy utiles. Las reglas del sistema deductivo para los cuantificadores universales y existenciales. En particular son:

$$\frac{p(x)}{\forall x(p(x))}\ \forall i \qquad\qquad \frac{p(t)}{\exists x(p(x))}\ \exists i$$
$$\frac{\forall x(p(x))}{p(t)}\ \forall e$$

$$\dfrac{\exists x(p(x)) \qquad \boxed{\begin{array}{c} p_n \\ \vdots \\ q \end{array}}}{q}\ \exists e$$

Teniendo en cuenta que ni en $\forall i$ ni en $\exists e$ la variable x debe ocurrir libre en ninguna hipótesis previa.

2 Especificación

2.1 El contrato

Dentro del mundo de la programacion, siempre trabajamos bajo la idea de *contrato*, en la que alguien nos solicita que diseñemos un programa que lo ayude a realizar una operacion concreta. Bajo estos terminos, necesitamos aclarar que tipo de problema queremos solucionar y como vamos a poder hacerlo.

Es en este contexto donde nace la idea de *especificación*, con la cual queremos dejar bien en claro, de manera formal y sin lugar a dudas o interpretaciones, que es lo que hay que resolver. No nos va a decir como hacerlo, pero si nos indica que tipo de parámetros de entrada vamos a utilizar, que salida esperamos, y, sin mas vueltas, que proceso queremos hacer. Esta escritura formal nos sera util a la hora de testear el programa y verificar su correctitud.

Cada parámetro que tenga una especificación va a tener un tipo de dato, lo cual es un conjunto de valores y operaciones que se pueden aplicar sobre esos mismos valores.

Retomando lo del contrato, siempre vamos a tener esta perspectiva. Vamos a trabajar en terminos de un *usuario* que nos solicita a nosotros, los *programadores*, que diseñemos un *programa* para solucionar un *problema*.

2.2 Sintaxis de la especificación

Una especificación consta de varias partes, en particular, se ven mas o menos asi:

```
proc nombre(parámetros){
    requiere { precondiciones }
    asegura { postcondiciones }
}
```

Donde `proc` es una palabra reservada para refererirnos a *procedimiento*, es para iniciar la definicion de la especificación. El *nombre* es el nombre que le damos a nuestra especificación, el cual podemos elegir nosotros. Recomendamos usar nombres claros y que indiquen cual es la operacion a resolver. Luego, los *parámetros* son los valores de entrada o salida que nuestra especificación va a necesitar para realizar su evaluacion, suelen escribirse tipados.

Luego, el renglon de `requiere`, la cual es una palabra reservada para iniciar las *precondiciones*. Estas van a ser nuestras condiciones que el usuario debera cumplir con sus valores de entrada. Es decir, nosotros (los *programadores*) vamos a dar por hecho que estas condiciones se van a cumplir. Es obligacion del usuario cumplir siempre con los `requiere` de nuestro programa.

Finalmente, llegamos a la parte del `asegura`, que también es una palabra reservada, en este caso inicia las *postcondiciones*. A diferencia del anterior punto, estas seran las condiciones que nuestro programa deberá cumplir para obtener la correcta salida o solucion. Es decir, es obligacion del *programador* asegurar que las *postcondiciones* se cumplan, y que el programa concluya lo que es correcto.

Una cosa a tener en cuenta es acerca de los parámetros, los cuales tienen tres formas de entrar: *in*, *out* e *inout*. Esto se suele especificar en la propia sintaxis antes de los parámetros, por ejemplo *in* $x : \mathbb{Z}$, o lo que es lo mismo *x va a ser una variable de entrada de tipo entero*. Primero, las variables que se inicializan con el prefijo *in* son aquellas que solo se puede acceder dentro de la funcion en la que se creo. El pasaje *out* hace referencia a que los valores son exclusivamente salida, es decir, parte del resultado o el resultado en cuestion. En definitiva, son variables que se ven afectadas al concluir el programa. Por otro lado, una variable que se inicializa con *inout*, es una variable de entrada pero que su valor se ve alterado o modificado al concluir el programa. No es un nuevo resultado, modificamos el mismo valor de entrada.

Algo util a tener en cuenta a la hora de trabajar con funciones que tengan parámetros *inout*, es como acceder al valor "viejo" o al "modificado" indistintamente. Para eso usamos la función *old(x)*, que toma una variable x y devuelve el valor inicial de la misma. Por ejemplo:

```
proc incrementar(inout x : ℤ){
    requiere { True }
    asegura { x = old(x) + 1 }
}
```

Otro ejemplo de especificación, una vez visto todo esto, seria:

```
proc sumar(in x : ℤ, in y : ℤ, out res : ℤ){
    requiere { True }
    asegura { res = x + y }
}
```

En el que vemos una función que devuelve la suma de los valores ingresados. Algo interesante es que en este caso el `requiere` no necesitaba nada, lo cual notamos con el valor *True*.

Vamos a contar con muchos tipos de datos, algunos conocidos y otros nuevos. En particular, tenemos:

- $\mathbb{Z}$: Numeros enteros
- $\mathbb{R}$: Numeros reales
- *Bool*: Tipo booleano, *verdadero* y *falso*
- *Char*: Caracter, denota una letra del alfabeto
- *enum nombre* {*constantes*} : Tipo enumerado
- $T_0 \times T_1 \times \ldots \times T_n$ o $(x_0, x_1, \ldots, x_n)$: n-uplas
- $seq\langle T \rangle$: Secuencia (o lista) de tipo T

Cada uno de estos tipos van a tener sus operaciones para poder trabajar con ellos, y la mayoria ya los conocemos, de hecho, pero eso lo veremos en profundidad mas adelante.

Una herramienta importante a la hora de escribir especificaciones son los *comentarios*, que son maneras de explicar en lenguaje humano que hace lo que estamos escribiendo en lenguaje lógico. Son partes de la especificación que no se van a tener en cuenta a la hora de resolver el problema, se van a ignorar, pero nos ayudan en la lectura. Se denotan como /∗ *tu comentario* ∗/.

```
proc raizCuadrada(in x : ℤ, out res : ℤ){
    /* La raiz cuadrada de reales solo existe para positivos */
    requiere { x ≤ 0 }
    asegura { res ≤ 0 ∧ (res * res = x) }
}
```

Para terminar, vamos a analizar unos conceptos importantes que son consecuencia de la estructura de la especificación. Existe el termino *sub-especificación*, lo que ocurre cuando la precondicion es muy restrictiva y la postcondicion mas laxa. Esto nos provoca una reduccion en los casos posibles de entrada muy notable, o ignora condiciones necesarias para la salida, lo que lleva a resultados no deseados. Por otro lado, existe la *sobre-especificación*, que es el caso inverso, es decir, tener una precondicion mas laxa y una postcondicion mas restringida. En este caso, permitimos mas entradas de las que quizas queremos, y limita los posibles algoritmos que puedan resolver luego el problema.

2.3 Funciones auxiliares, predicados y condicionales

Existen ocasiones en las que hay una o mas operaciones que utilizamos reiteradas veces, y puede ser molesto repetirlas, aparte de hacer dificil la futura lectura de la especificación. Para estos casos, tenemos las *funciones auxiliares*, que son especificaciones mucho mas basicas y sencillas, las cuales hacen una operacion exclusiva. La sintaxis de las funciones auxiliares se ve asi:

`aux` *nombre* (*parámetros*) : *resultado* = *procedimiento*

Se asemeja bastante a lo visto antes, lo unico nuevo es aquella parte de *procedimiento*. Aqui es donde escribiremos la operacion a realizar o el resultado a dar. Fijense que no tenemos ni `requiere` ni `asegura`, lo cual hace mucho mas basica esta estructura. Un ejemplo habitual puede ser obtener un valor aproximado de alguna constante matematica, como:

`aux` *pi* () : $\mathbb{R} = 3.14159$

Noten que aca hicimos una función que no tiene parámetros de entrada, lo cual también podria aplicarse a las especificaciones habituales.

También, podemos definir *predicados*, que son operaciones basicas como las funciones auxiliares de antes, pero que exclusivamente evaluan una fórmula lógica, es decir, su resultado es unicamente un valor de verdad. Un ejemplo comun de esto puede ser:

```
pred esPar ( n :ℤ ) {
    n mod 2 = 0
}
```

Finalmente, tenemos las *expresiones condicionales*, las cuales son funciones que elige entre dos elementos del mismo tipo, según una fórmula lógica. En particular, si la fórmula es verdadera, elije la primera opcion, y en caso contrario, la segunda. La estructura de una expresion condicional es del tipo:

`if` *condicion* `then` *primer resultado* `else` *segundo resultado* `fi`

Estas estructuras se pueden usar en cualquier especificacion, tanto en los procedimientos como en las funciones auxiliares. Por ejemplo, si quisieramos una función que devuelve el máximo entre dos numeros dados, podriamos escribir:

`aux` *máx* $(n : \mathbb{Z}, m : \mathbb{Z}) : \mathbb{Z} =$ `if` $n > m$ `then` n `else` m `fi`

Algo importante a tener en cuenta, es que a la hora de especificar problemas no podemos utilizar otros procedimientos para la solucion del mismo. Solo podemos utilizar predicados y funciones auxiliares, obviamente definiendolos previamente. A su vez, dentro de las funciones auxiliares no podemos definir instrucciones recursivas, ya que solo se tratan de un reemplazo sintactico.

2.4 Viejos y nuevos tipos de datos

Como mencionamos antes, tenemos muchos tipos de datos a la hora de especificar, lo cual esta bueno porque nos da mucha versatibilidad a la hora de estucturar problemas mas complejos, y no hay necesidad de reinventar siempre cada tipo de dato. Pero si es importante repasar las operaciones y fórmulas que podemos construir con cada uno.

Los tipos "numericos", es decir, los *enteros* y los *reales*, ya los conocemos bien. Tenemos todas las operaciones habituales: suma, resta, multiplicacion, y en los reales, la division (pero no por cero). También tenemos los operandos de orden (menor, mayor, menor o igual, etc), y la igualdad. La potenciacion, y, en los reales, la raiz (con indice par, no para negativos), entre otras.

Luego tenemos los tipos *bool*, o *booleanos*, que contienen valores de verdad. De esto hablamos mas a fondo en todo el capitulo 1 (el de lógica), pero en particular podemos tratarlos con la lógica de la semántica bi-valuada estándar, es decir, tenemos la *negación* (notada $\neg$ o a veces con *!*), el *y lógico* ($\wedge$ o &&) y el *o lógico* ($\vee$ o ||).

El tipo *char*, o caracteres en español, es el conjunto de letras, dígitos y símbolos del alfabeto español. En particular. viene en el orden estándar ASCII, que seria $'a', 'b', \ldots, 'z', 'A', 'B', \ldots, 'Z', '1', '2', \ldots, '9'$. A los caracteres podemos consultar su numeracion en este orden, u obtener un caracter dado un orden puntual, por ejemplo:

$ord('a') = 1$ $\qquad$ $char(ord('b')) =' b'$

$ord('a') + 1 = ord('b')$ $\qquad$ $char(ord('c') - ord('b')) =' a'$

$char(3) =' c'$ $\qquad$ $ord('a') < ord('c')$

Los tipos *enumerados*, por otro lado, son un conjunto de constantes que construimos nosotros. Como tal no hay tipos enumerados preexistentes, los vamos inventando a medida que los necesitemos. Existe, para tener en cuenta, la convención de escribir sus valores en mayúsculas, ya que son datos inmutables y que no van a cambiar, asi las diferenciamos de otras variables. Para definir el tipo *Día*, por ejemplo, escribiriamos:

`enum` *Día*$\{LUN, MAR, MIER, JUE, VIE, SAB, DOM\}$

Equivalentemente a los *char*, los tipo *enumerados* poseen funciones de orden y comparacion. Por ejemplo, podemos consultar que dia es el que está en la posicion 2, o en que posicion está determinado día. Noten que el indice de posiciones empieza en 0.

$ord(LUN) = 0$ $\qquad$ $Día(2) = MIE$ $\qquad$ $MAR < JUE$

Continuando con los tipos de datos, tenemos la *upla*, o $n - upla$, o *tupla*. Le digan como le digan, en definitiva, se trata de un "vector" de datos, que pueden ser distintos o no. Algo importante, es que las *tuplas*

tienen un tamaño fijo: no podemos agregar ni quitar elementos. Se definen $T_0 \times T_1 \times \ldots \times T_n$, donde T_i es el tipo de en la posicion i. Por ejemplo, $\mathbb{Z} \times \mathbb{Z}$ define el conjunto de vectore de dos dimensiones de los enteros, es decir los (x, y), donde tanto x e y son enteros. También podemos combinar tipos, como $\mathbb{Z} \times Char$, es decir, vectores de la forma (n, a), donde n es un entero y a algun caracter de los que vimos antes.

Podemos acceder a cualquier posicion del vector utilizando la notacion $(a_0, a_1, \ldots, a_n)_m$, donde $0 \leq m \leq k$, en caso contrario se indefine.

$(12, True,' a')_0 = 12$ $\quad$ $(12, True,' a')_2 =' a'$

$(12, True,' a')_1 = True$ $\quad$ $(12, True,' a')_3 = \perp$

2.5 Secuencias, matrices y conjuntos

Luego, en la lista de tipos, tenemos a las *secuencias*, que tienen tantas operaciones y particularidades que tienen su propio capitulo. Las *secuencias* son una coleccion de valores de un mismo tipo, que pueden estar repetidos o no, y que tienen un cierto orden. Se definen $seq\langle \mathrm{T} \rangle$, donde T es el tipo de los datos. El tamaño de las secuencias, a diferencia de las tuplas, puede variar, o incluso pueden estar vacias. A su vez, como $seq\langle \mathrm{T} \rangle$ es un tipo de dato en si mismo, podriamos definir una secuencia de secuencias de tipo T, es decir $seq\langle seq\langle \mathrm{T} \rangle \rangle$.

Por ejemplo, veamos una $seq\langle \mathbb{Z} \rangle$:

- $\langle$1,2,3,4,5$\rangle$ es una secuencia de enteros
- $\langle$5,4,3,2,1$\rangle$ es *otra* secuencia de enteros, ya que tiene otro orden
- $\langle$1,2,3,4,True$\rangle$ es una secuencia de enteros mal definida, ya que uno de sus elementos es de otro tipo
- $\langle\rangle$ es una secuencia de enteros que está vacia

Tenemos un monton de funciones muy interesantes con las secuencias, vamos a verlas una a una.

Para empezar, tenemos una operacion que nos indica la *longitud* de la secuencia. Lo notamos $length(a : seq\langle T \rangle) : \mathbb{Z}$, donde a es la secuencia a evaluar. También podemos escribir $|a|$, que es considerablemente mas corto. Unos ejemplos de esto:

$|\langle$1,2,3$\rangle| = 3$ $\quad$ $|\langle\rangle| = 0$ $\quad$ $|\langle$'M', 'a', 's'$\rangle| = 4$

Por otro lado, tenemos la *indexacion*, que dado un indice de la secuencia nos devuelve el valor en esa posicion. Requerimos que el indice dado i para la lista l cumpla $0 \leq i < |l|$, ya que en otros casos sera indefinido. Es semejante a lo que haciamos con los vectores, solo cambia la notacion, pero es algo a tener en cuenta.

$\langle$1, 2, 3$\rangle[0] = 1$ $\quad$ $\langle\rangle[0] = \perp$ $\quad$ $\langle$'M', 'a', 's'$\rangle[1] =' a'$

También podemos evaluar *igualdad* entre secuencias, la cual se cumple si y solo si tienen la misma cantidad de elementos, y para cualquier indice dado, el i-esimo elemento en la primera lista es igual al de la segunda. Por ejemplo:

$\langle$1, 2, 3$\rangle = \langle$1, 2, 3$\rangle$ $\quad$ $\langle$1, 2, 3$\rangle \neq \langle$1, 3, 2$\rangle$

$\langle\rangle = \langle\rangle$ $\quad$ $\langle$2, 2, 2$\rangle = \langle$2, 2, 2$\rangle$

Luego tenemos las operaciones de *head* (o cabeza) y *tail* (o). La operacion *head*, dada una lista s nos devuelve el primer elemento de la misma, pero es requisito que la lista tenga por lo menos un elemento. Por otro lado, la función *tail*, dada una lista s nos devuelve la lista sin el primer elemento, es decir, habiendole sacado la cabeza.

$head(\langle$1,2,3$\rangle) = 1$ $\quad$ $tail(\langle$1,3,2$\rangle) = \langle$3,2$\rangle$

$head(\langle\rangle) = \perp$ $\quad$ $tail(\langle$2$\rangle) = \langle\rangle$

Para agregar elementos a una lista tenemos la operación de *AddFirst* (o añadir al principio), que dado un elemento t y una lista l, crea una secuencia igual a l pero "moviendo" todos sus elementos una posicion y dejando en la primera a t.

$addFirst$('H', ⟨'o','l','a'⟩) = ⟨'H','o','l','a'⟩

$addFirst$(1, ⟨2,2,2⟩) = ⟨1,2,2,2⟩

$addFirst$(1, ⟨⟩) = ⟨1⟩

Quizas queremos combinar dos secuencias en una sola, y para eso tenemos la operación de la *concatenación*. La notamos $concat(a, b)$ o a ++ b, donde a y b son dos secuencias cualesquiera. Lo que hace esta operación es devolvernos una lista con los elementos de a, seguidos de los de b. Las dos listas **deben** ser del mismo tipo.

⟨'H'⟩ ++ ⟨'o','l','a'⟩ = ⟨'H','o','l','a'⟩

⟨1,2,3⟩ ++ ⟨3,4,5⟩ = ⟨1,2,3,3,4,5⟩

⟨⟩ ++ ⟨⟩ = ⟨⟩

⟨1,2⟩ ++ ⟨⟩ = ⟨1,2⟩

⟨⟩ ++ ⟨1,2⟩ = ⟨1,2⟩

Tambien podemos "cortar" secuencias, con la operacion de *subseq* (o subsecuencia). Esta operacion toma una lista l, y dos numeros enteros d y h, donde se cumple que $0 \leq d \leq h \leq |l|$, y cuando no se indefine. Esta operación devuelve una sublista de l en las posiciones entre d (inclusive) y h (exclusive). Notemos que cuando $d = h$, el resultado es la secuencia vacía.

$subseq$(⟨'o','l','a'⟩, 0, 1) = ⟨'o'⟩

$subseq$(⟨'o','l','a'⟩, 0, 3) = ⟨'o','l','a'⟩

$subseq$(⟨'o','l','a'⟩, 1, 1) = ⟨⟩

$subseq$(⟨'o','l','a'⟩, -1, 3) =⊥

$subseq$(⟨'o','l','a'⟩, 0, 5) =⊥

Para terminar tenemos la operación de reemplazar un valor en alguna posicion de una secuencia. Es decir, la operación *setAt* (o modifica en). Dada una lista l, una posicion i, y un valor n del tipo de los elementos de la secuencia, podemos construir la operacion $setAt(l, i, n)$, donde requerimos que $0 \leq i < |l|$, ya que en caso contrario se indefiniria.

$setAt$(⟨'o','l','a'⟩, 0, 'p') = ⟨'p','l','a'⟩

$setAt$(⟨'o','l','a'⟩, 0, 'O') = ⟨'O','l','a'⟩

$setAt$(⟨'o','l','a'⟩, 0, 3) =⊥

$setAt$(⟨⟩, 0, 5) =⊥

Aparte de las secuencias, tenemos otro tipo igual de interesante: los *conjuntos*. Estos son muy parecidos a las secuencias, pero con claras diferencias. Para empezar, en los conjuntos no puede haber elementos repetidos, y no importa el orden. Al listar sus elementos los encerramos entre llaves como $\{\ldots\}$, pero tambien podemos notarlos $conj\langle T\rangle$, donde T es el tipo de los elementos que contiene, ya que deben ser todos del mismo tipo, como en las secuencias.

Al igual que en las secuencias, es como $conj\langle T\rangle$ es un tipo en si mismo, es posible crear conjuntos de conjuntos, o incluso conjuntos de secuencias, o secuencias de conjuntos. Algunos ejemplos para empezar pueden ser:

- {1,2,3} es un conjunto de $\mathbb{Z}$
- {3,2,1,1} es un conjunto de $\mathbb{Z}$ igual al anterior, porque no impota el orden o los repetidos
- {} es un conjunto vacio de cualquier tipo.

Tenemos unas cuantas operaciones para trabajar sobre conjuntos. Empecemos con las mas básicas: el *cardinal*. Representa el tamaño del conjunto, es decir, la cantidad de elementos que tiene. Podemos escribirlo como $cardinal(c)$ o bien $|c|$, donde c es el conjunto a evaluar.

$|\{1,2\}| = 2$ $\qquad$ $|\{'o','l','a'\}| = 3$ $\qquad$ $|\{\}| = 0$

Tambien tenemos la operación de *pertenece*, que nos indica si un elemento e pertenece al conjunto c. Es importante notar que e debe ser del mismo tipo que los elementos que contiene c. Lo escribimos como $in(e, c)$, o mas coloquialmente como $e \in c$.

$1 \in \{1,2\} = True$ 'b' $\in \{'a'\} = False$ $2 \in \{\} = False$

Por otro lado, podemos combinar conjuntos con la operación *unión*, que notamos $c_o \cup c_1$, que significa *todos los elementos de* c_0 *y los de* c_1. Obviamente, los elementos de c_0 y los de c_1 deben ser del mismo tipo entre si.

$\{1,2\} \cup \{3,4\} = \{1,2,3,4\}$ $\{\} \cup \{\} = \{\}$
$\{1,2\} \cup \{\} = \{1,2\}$

Aparte de esto, tenemos la intersección de conjuntos, que notamos como $intersection(c_0, c_1)$, o mas comodamente $c_0 \cap c_1$. Es decir, los elementos que compartan los conjuntos.

$\{1,2\} \cap \{2,3\} = \{2\}$ $\{1,2\} \cap \{\} = \{\}$

Tambien existe la *diferencia* de conjuntos, que notamos $diff(c_0, c_1)$ o bien $c_0 - c_1$. Consiste en "sacar" de c_0 todos los elementos que comparta con c_1.

$\{1,2\} - \{2,3\} = \{1\}$ $\{1,2\} - \{\} = \{1,2\}$

Finalmente, tenemos la *igualdad*, conocida como $c_0 = c_1$. Determina si dos conjuntos son o no iguales, lo que sucede si y solo si los dos conjuntos tienen la misma cantidad de elementos, y tienen exactamente los mismos elementos. Recordemos que en conjuntos el orden no es relevante.

$\{1,2\} = \{1,2\}$ $\{1,2,1,2\} = \{2,1\}$
$\{1,2\} = \{2,1\}$ $\{1,2,3\} \neq \{1,2\}$

Por ultimo, las *matrices*. Consisten en secuencias de secuencias, donde todas las secuencias internas tienen exactamente el mismo tamaño, y no deben ser vacias. Es decir, una matriz de numeros enteros se define como $seq\langle seq\langle \mathbb{Z} \rangle\rangle$, aunque tambien aceptamos el reemplazo sintactico de $Mat\langle \mathbb{Z} \rangle$.

En si mismo, las matrices no tienen operaciones basicas, pero, al tratarse finalmente de secuencias, podemos construir las nuestras. Por ejemplo, podemos construir un predicado que nos indique si una secuencia de secuencias es efectivamente una matriz:

```
pred esMatriz ( m : seq⟨seq⟨ℤ⟩⟩ ) {
    (∀i)(0 ≤ i < |m| ⟶_L |m[i]| > 0 ∧ (∀j)(0 ≤ j < |m| ⟶_L |m[j]| = |m[i]|))
}
```

Tambien, para ser un poco mas explicitos a la hora de medir las filas o columnas, podemos construir funciones auxiliares.

`aux` *filas* $(m : seq\langle seq\langle \mathbb{Z} \rangle\rangle) : \mathbb{Z} = |m|$

`aux` *columnas* $(m : seq\langle seq\langle \mathbb{Z} \rangle\rangle) : \mathbb{Z} =$ `if` $filas(m) > 0$ `then` $|m[0]|$ `else` 0 `fi`

De aca, podemos definir estructuras mas complejas, como determinar si una matriz es la matriz identidad, o si es una matriz cuadrada, por ejemplo.

2.6 Sumatorias y productorias

Para los tipos numéricos $\mathbb{Z}$ y $\mathbb{R}$ contamos con estructuras matematicas para acumular resultados de operaciones repetidas. Una de estas estructuras es la *sumatoria*, que notamos:

$$\sum_{i=d}^{h} p(i)$$

Y que retorna la suma total de todas las expresiones $p(i)$, con $d \leq i \leq h$, es decir:

$$p(d) + p(d+1) + \ldots + p(h-1) + p(h)$$

Hay que tener en cuenta que si no se cumple que $d < h$ la sumatoria va a retornar 0, ya que d y h deben definir un rango finito de valores enteros. Si existe un i tal que $d \leq i \leq h$ y $p(i) = \perp$, entonces toda la expresión se indefine. Noten que todas las expresiones $p(i)$ deben ser un valor numérico, es decir pertenecer a $\mathbb{Z}$ o a $\mathbb{R}$.

Podemos construir estructuras muy interesantes con esta notacion, por ejemplo, si quisieramos sumar todos los elementos de una secuencia podriamos escribir un procedimiento que sea:

```
proc sumaTodos(in l : seq⟨ℤ⟩, out res : ℤ){
    requiere { True }
    asegura { res = Σ_{i=0}^{|l|-1} l[i] }
}
```

De manera analoga, tenemos la *productoria*: la cual consiste en mas o menos lo mismo, pero en lugar de sumar todas las expresiones bajo un rango, vamos a multiplicarlas. Las escribimos:

$$\prod_{i=d}^{h} p(i)$$

Donde tambien necesitamos que $d < h$, ya que si no devolveria 1. Identicamente a lo anterior, si existe un i tal que $d \leq i \leq h$ y $p(i) = \perp$, entonces toda la expresión se indefine. Noten que todas las expresiones $p(i)$ deben ser un valor numérico, es decir pertenecer a $\mathbb{Z}$ o a $\mathbb{R}$.

3 Correctitud

3.1 Estados y ejecución

A la hora de programar para resolver un problema, no solo queremos escribir codigo "a lo bobo", es necesario verificar que ese codigo, ese *programa* resuelve satisfactoriamente el problema, la especificación. Para verificar lo que llamamos *correctitud* vamos a trabajar con el concepto de *estado*, que consiste en la asignacion y valor de las variables en determinados momentos de ejecucion del codigo mismo.

Decimos que un programa, con una precondición P y una postcondición Q, es correcto si y solo si, iniciando en un estado P **finaliza su ejecucion** y alcanza un estado final que **satisface** Q. Si llamamos S al programa, lo notamos {P} `S` {Q}.

Por ejemplo, si analizamos el siguiente programa:

```
int x := 0;
x := x + 3;
x := x * 2;
```

Si bien no es un programa muy interesante, podemos poner a prueba el metodo que vamos a utilizar en este capitulo, que es el de analizar la sucesion de los distintos estados a lo largo del programa. Continuando con el mismo ejemplo, podriamos afirmar que:

```
int x := 0;
```
$\{\ x = 0\ \}$
```
x := x + 3;
```
$\{\ x = 3\ \}$
```
x := x * 2;
```
$\{\ x = 6\ \}$

Entonces podemos determinar cual sera la conclusion del programa, y mas importante aun, que tendra una. A esto nos refereriamos con *finalizar la ejecucion*, el programa no se queda atorado en un bucle infinito o analiza una situacion indefinida, llega a una conclusion.

Ahora veamos un programa algo mas complejo. Supongamos que tenemos un programa que, segun lo que nos indican, dados dos valores a y b, el programa intercambia los valores. Es decir, le asigna a a lo que valiera b, y viceversa. El programa es el siguiente:

```
a := a + b;
b := a - b;
a := a - b;
```

A primera vista no es facil determinar si es o no correcto, por lo que iremos recorriendo las sentencias una a una. La idea es que si partimos de que $a = A_o$ y $b = B_0$, queremos llegar a que $a = B_o$ y $b = A_0$.

$$\{\ a = A_0 \land b = B_0\ \}$$
```
a := a + b;
```
$$\{\ a = A_0 + B_0 \land b = B_0\ \}$$
```
b := a - b;
```
$$\{\ a = A_0 + B_0 \land b = (A_0 + B_0) - B_0 = A_0\ \}$$
```
a := a - b;
```
$$\{\ a = (A_0 + B_0) - B_0 = B_0 \land b = A_0\ \}$$

Y llegamos a la postcondicion que queriamos. Efectivamente, este programa cumple con lo pedido, decimos que es *correcto*.

En particular, no vamos a corregir programas escritos en un lenguaje de programación especifico, como Java o Python, sino que vamos a inventar nuestro propio lenguaje de instrucciones basicas, con el que podremos diseñar los programas, antes de implementarlos, y hacer este tipo de demostraciones y verificaciones.

3.2 SmallLang y predicados de analisis

A este "lenguaje de programación" simplificado lo llamaremos *smallLang*, del ingles *small language*, o *pequeño lenguaje* en español. Va a consistir de dos instrucciones basicas:

Hacer nada: Se indica con la instruccion *skip*.

Asignación: Indicada por la instruccion $x := E$ (asignemos E a la variable x).

A su vez, tenemos estructuras de control mas complejas que nos van a permitir hacer evaluaciones o formulas mas complejas:

Secuencia: $S1; S2$ si tanto $S1$ como $S2$ son programas.

Condicional: *if* B *then* $S1$ *else* $S2$ *endif* es un programa, si B es una expresion lógica, y si tanto $S1$ como $S2$ son programas.

Ciclo: *while* B *do* S *endwhile* es un programa, si B es una expresion lógica, y si S es un programa.

Con este lenguaje, por mas basico que parezca, podemos representar algoritmos complejos y, lo que nos interesa a nosotros en particular, demostrar que son correctos.

Aparte de esto, para evaluar las instrucciones del smallLang, vamos a tener dos predicados nuevos: $def(E)$ y Q^x_E.

Decimos que, dando un predicado E, $def(E)$ son las condiciones necesarias para que E este definida. Por ejemplo:

$def(x + y) \equiv def(x) \wedge def(y)$

$def(x/y) \equiv def(x) \wedge (def(y) \wedge y \neq 0)$

$def(\sqrt{x}) \equiv def(x) \wedge x \geq 0$

$def(a[i]) \equiv (def(a) \wedge def(i)) \wedge_L 0 \leq i < |a|$

En particular, nosotros damos por hecho que las variables ya estan definidas. Es decir, que $def(x)$, con x una variable cualquiera, es $True$. Teniendo esto en cuenta, podemos reformular lo antes dicho a:

$def(x + y) \equiv True$

$def(x/y) \equiv y \neq 0)$

$def(\sqrt{x}) \equiv x \geq 0$

$def(a[i]) \equiv 0 \leq i < |a|$

Finalmente, el predicado Q^x_E se lee como *reemplaza en el predicado* Q*, todas las apariciones* ***libres*** *de la variable* x *por la expresion* E. Un ejemplo de esto seria:

$Q \equiv 0 \leq i < n \wedge_L (\forall j : \mathbb{Z})(a[j] = x)$

$Q^i_k \equiv 0 \leq k < n \wedge_L (\forall j : \mathbb{Z})(a[j] = x)$

$Q^j_k \equiv 0 \leq i < n \wedge_L (\forall j : \mathbb{Z})(a[j] = x)$

Con todo esto, podemos construir todas las demostraciones de todos los algoritmos que nos encontremos, y para hacer esto, tenemos un metodo: el de la *precondicion mas debil*.

3.3 Precondición mas debil

Los problemas pueden tener muchos `requiere` y `asegura`, como vimos en el capitulo anterior. En particular, nos interesa buscar unos `requiere` P que, al ejecutar el programa S, podamos concluir el `asegura` Q.

Llamamos *precondición mas debil* a la precondición menos restrictiva, es decir, lo minimo que podemos pedir para que el programa concluya con la postcondición Q. Lo notamos como $wp(S, P)$, del ingles *weakest condition*. Un ejemplo para ilustrar esto puede ser:

```
x := x + 1;
```

Si suponemos una postcondición $Q \equiv x \leq 6$, podriamos deducir la precondición, y escribir:

$$\{\ x \leq 5\ \}$$

```
x := x + 1;
```
$\{ x \leq 6 \}$

Pero tambien podriamos escribir:

$\{ x \leq 6 \}$
```
x := x + 1;
```
$\{ x \leq 6 \}$

Lo cual seguiria siendo correcto, pero nos estariamos "salteando" posibles casos de entrada, posibles valores que x puede tomar. En este caso, la precondicion mas debil seria $\{x \leq 5\}$, ya que es la que nos permite la mayor cantidad de casos de entrada, como dijimos antes, la menos restrictiva.

Para encontrar estas precondiciones mas debiles, lo que haremos sera recorrer la ejecución del programa "para atras", empezando desde la ultima linea y tomando como referencia la conclusion, la postcondición. Es decir, intentaremos deducir que tuvo que pasar antes para llegar a la postcondición a la que se llego.

Para esto vamos a utilizar una serie de axiomas para transformar las instrucciones de smallLang a predicados logicos, que finalmente es lo que son las precondiciones. Los axiomas son los siguientes:

1. $wp(x := E, Q) \equiv def(E) \wedge_L Q^x_E$
2. $wp(skip, Q) \equiv Q$
3. $wp(S1; S2, Q) \equiv wp(S1, wp(S2, Q))$
4. Si $S \equiv$ *if* B *then* $S1$ *else* $S2$ *endif*, entonces,
$$wp(S, Q) \equiv def(B) \wedge_L ((B \wedge wp(S1, Q)) \vee (\neg B \wedge wp(S2, Q)))$$

Utilizando estas reglas podemos ir reconstruyendo las precondiciones apartir de las postcondiciones, y de hecho, vamos a llegar a la mas debil, siempre. Una vez llegados a la precondición mas debil, lo que nos interesa chequear es si nuestra precondición inicial era en efecto buena. En particular, nos interesa ver que:

$$P \longrightarrow wp(S, Q)$$

Veamos un ejemplo mas complejo para que quede todo claro. Supongamos el siguiente programa S:

```
    if (a > b) then
S1:     res := a - b;
    else
S2:     res := b - a;
    endif;
```

Con la postcondicion de $Q \equiv \{res = |a - b|\}$, y ninguna precondición. No conocemos que tienen que cumplir las entradas para que el programa pueda concluir certeramente la postcondición, por lo que vamos a averiguarlo. Primero reescribamos lo que ya sabemos:

```
    if (a > b) then
S1:     res := a - b;
    else
S2:     res := b - a;
    endif;
```

$\{ res = |a - b| \}$

Utilizando el axioma 4, podemos deducir que:

$wp(S, Q) \equiv def(a > b) \wedge_L ((a > b \wedge wp(S1, Q) \vee (\neg(a > b) \wedge wp(S2, Q)))$

$wp(S, Q) \equiv def(a) \wedge def(b) \wedge_L ((a > b \wedge wp(S1, Q) \vee (\neg(a > b) \wedge wp(S2, Q)))$

$wp(S, Q) \equiv True \wedge_L ((a > b \wedge wp(S1, Q)) \vee (\neg(a > b) \wedge wp(S2, Q)))$

$wp(S, Q) \equiv (a > b \wedge wp(S1, Q)) \vee (\neg(a > b) \wedge wp(S2, Q)$

Veamos cada precondición mas debil por separado para mas orden.

$wp(S1, Q) \equiv (def(a - b) \wedge_L a - b = |a - b|)$

$wp(S1, Q) \equiv (def(a) \wedge def(b)) \wedge_L (a - b = a - b \vee a - b = b - a)$

$wp(S1, Q) \equiv True \wedge_L (True \vee a - b = b - a)$

$wp(S1, Q) \equiv True$

Analogamente, con la precondición de $S2$, llegamos a:

$wp(S1, Q) \equiv (def(b - a) \wedge_L b - a = |a - b|)$
$wp(S2, Q) \equiv True$
Retomando lo que dejamos a medias, quedaria:
$wp(S, Q) \equiv (a > b \wedge wp(S1, Q)) \vee (\neg(a > b) \wedge wp(S2, Q))$
$wp(S, Q) \equiv (a > b \wedge True) \vee (\neg(a > b) \wedge True)$
$wp(S, Q) \equiv a > b \vee \neg(a > b)$
$wp(S, Q) \equiv True$

Y finalmente, nuestra precondición mas debil para este problema, con ese programa, es $\{True\}$, es decir, que no necesitamos pedirle nada a los parametros de entrada, podemos ingresar cualquier numero.

Antes de terminar, vamos a hablar sobre las *secuencias*, ya que son un caso bastante particular a la hora de buscar su precondición mas debil. Notemos que si tuvieramos la instruccion:

```
a[i] := 3;
```

Es equivalente a $setAt$(a, i, 3) que vimos antes. Lo bueno de esto, que a primera vista parece ser solo un cambio de nombre y ya, es que podemos transcribir una instruccion del smallLang al lenguaje de predicados y lógica de primer orden que vimos antes. Es decir, podemos traducir a precondiciones o postcondiciones los algoritmos que diseñemos.

Supongamos el codigo y la postcondicion:

```
A[i+2] := 0;
```

$$\{\ (\forall j : \mathbb{Z})(0 \leq j < |A| \wedge_L A[j] \geq 0)\ \}$$

Para empezar a buscar la precondición mas debil en esta situacion, hacemos como hariamos habitualmente. Llamemos S al programa y Q a la postcondición, momentaneamente.

$$wp(S, Q) \equiv def(S) \wedge_L Q_S^A$$

Pero la asignación S no tiene significado directo en nuestro lenguaje de primer orden, por lo que debemos traducirlo a la función $setAt(,,)$, como dijimos antes. Entonces:

$$S \equiv setAt(\text{A, i+2, 0})$$
$$wp(S, Q) \equiv def(S) \wedge_L Q_S^S$$

Luego, podemos reemplazar, y seguir las reglas conocidas.
$wp(S, Q) \equiv def(S) \wedge_L Q_S^A$
$def(setAt(\text{A, i+2, 0})) \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |setAt(\text{A, i+2, 0})| \wedge_L setAt(\text{A, i+2, 0})[j] \geq 0)$
$0 \leq i + 2 < |A| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |A| \wedge_L setAt(\text{A, i+2, 0})[j] \geq 0)$
$0 \leq i + 2 < |A| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |A| \wedge_L ((j = i + 2 \longrightarrow A[i + 2] \geq 0) \wedge (j \neq i + 2 \longrightarrow A[j] \geq 0)))$

Ahora, se que quedo como un choclo largo todo esto, pero se va a ir acortando. Primero notemos como separamos los casos donde $j = i + 2$ para poder desarmar el $setAt(,,)$, ese truco es muy util. Por otro lado, podemos darnos cuenta que el valor de $A[i + 2]$ es conocido, ya que es donde estamos haciendo la asignacion, asi que lo conocemos; de ese modo, podemos continuar:
$0 \leq i + 2 < |A| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |A| \wedge_L ((j = i + 2 \longrightarrow 0 \geq 0) \wedge (j \neq i + 2 \longrightarrow A[j] \geq 0)))$
$0 \leq i + 2 < |A| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |A| \wedge_L ((j = i + 2 \longrightarrow True) \wedge (j \neq i + 2 \longrightarrow A[j] \geq 0)))$
$0 \leq i + 2 < |A| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |A| \wedge_L ((True) \wedge (j \neq i + 2 \longrightarrow A[j] \geq 0)))$
$0 \leq i + 2 < |A| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |A| \wedge_L (j \neq i + 2 \longrightarrow A[j] \geq 0))$

Y asi llegamos a la precondición mas debil de este ejemplo. Noten que tiene mucho sentido, ya que la postcondición estaba diciendo que al finalizar el programa todos los valores en la secuencia A seran positivos, pero al ejecutar el programa sabiamos que en la posición $i+2$ iba a haber un 0. Entonces, la precondición mas debil, el conjunto de secuencias mas amplio que servirian de entrada a este programa, son aquellas donde todos sus valores son positivos, salvo por la posicion $i + 2$, que es que el programa va a cambiar de todos modos.

3.4 El quinto axioma

Antes nosotros mostramos en particular unicamente cuatro axiomas, pero en realidad existen cinco. No mostramos que hacer con los ciclos, es decir, como encontrar su precondición mas debil.

Primero veamos una operacion de predicados, parecida a las que vimos antes. Dado un programa de la forma `while B do S endwhile` vamos a definir la operacion $H_k(Q)$ como el conjunto de precondiciones tal que al ejercutar el programa durante k iteraciones, concluya en el estado Q.

Se define inductivamente como:
$H_0(Q) \equiv def(B) \wedge \neg B \wedge Q$
$H_{k+1}(Q) \equiv def(B) \wedge B \wedge wp(S, H_k(Q)) \forall k$

Lo cual tiene lógica, ya que si el programa se ejecuta 0 veces, entonces es identico a la regla `skip`: Q tendria que haber valido de antes. Por otro lado, si sabemos que se ejecuto $k+1$ veces el programa, podemos asegurar que necesitamos la precondición mas debil tal que al ejecutarlo una vez, tenia que cumplir antes que al ejecutarlo k veces concluyamos en aquella precondición antes mencionada.

Finalmente, el quinto axioma de la precondicion mas debil quedaria:

$$wp(\texttt{while B do S endwhile}, Q) \equiv (\exists i \geq 0)(H_i(Q))$$

Otra cosa importante para tener en cuenta, es como definimos la correctitud en un ciclo. Para que un ciclo sea correcto decimos que se tienen que cumplir dos cosas: primero, que dada la precondición P, si corremos el programa con algun ciclo S, llegamos a satisfacer la postcondición Q; y segundo, que el ciclo termine. Es decir, que el programa no se *cuelgue* en un ciclo infinito.

Nota aparte, esto es uno de los problemas mas importantes de la computación en general: *the halting problem*, o *problema de la parada*, en español. La pregunta que se intenta averiguar es, si dado un programa cualquiera, podemos determinar o no si este se detendra. La respuesta es no, basicamente. Esta demostrado que existen programas a los que no podremos determinar si se deben detener en algun momento o tienen una ejecucion infinita.

Existen metodos, pero no son tan comodos o deterministas de desarrollar. En particular, nosotros veremos dos: el *teorema del invariante*, para analizar si la ejecucion del programa es correcta, y el *teorema de terminacion*, con el que diremos si el programa finaliza o no.

3.5 Invariante y terminacion de ciclos

Estos son los dos metodos que vamos a utilizar para demostrar la correctitud de programas que contengan ciclos. Vamos por partes, que siempre es mas comodo. Primero vamos a ver cada teorema por separado, y luego mostraremos un ejemplo desarrollado donde apliquemos ambos.

En primer lugar, tenemos el *teorema del invariante*, que consiste en demostrar que el programa, dada la precondición, va a cumplir la postcondición. Como tal, el procedimiento es bastante directo y mecanizable, el problema radica en encontrar al *invariante*. El invariante es un predicado que, como dice su nombre, no varia. Especificamente, no varia durante la ejecución del programa.

Es decir, si llamamos I al invariante de un programa S. El invariante seria un predicado tal que I vale antes de comenzar el ciclo, y que, si vale I y B en cualquier iteración del ciclo, entonces I seguira valiendo al finalizar esa iteración.

En particular, una vez que tenemos al invariante I, el teorema del invariante dice que el programa es *parcialmente* correcto, si se cumple que:

1. $P \longrightarrow I$
2. $\{I \wedge B\}$ `S` $\{I\}$
3. $(I \wedge \neg B) \longrightarrow Q$

Con *parcialmente correcto* nos referimos a que se verifica que el programa, dada la precondición, cumple la postcondición, pero no sabemos si el programa termina o no.

Veamos punto por punto. Primero queremos que nuestra precondición implique nuestro invariante. Es decir, que si se cumplen las condiciones del programa, tambien se cumpla el invariante. Por otro lado, queremos que si se cumple el invariante y la condición del ciclo, es decir, va a suceder alguna iteración del mismo, queremos que al finalizar la ejecucion de cuerpo del ciclo, el invariante siga valiendo. O sea, que no cambie. Finalmente, lo que queremos es que, si el invariante se cumple, y no se cumple la guarda, es decir, no vamos a iterar dentro del ciclo, se va a satisfacer la postcondición del programa.

Si lo seguimos punto a punto es bastante lógico, queremos un punto medio, un *puente*, entre la precondición, y la postcondición, y que no cambie durante las iteraciones del ciclo. El problema radica a la hora de elegir un buen invariante.

En segundo lugar, tenemos el *teorema de terminación*, que nos ayuda a determinar si un programa finaliza su ejecucion o no. Analogo al invariante, en este caso vamos a necesitar una f_V, tambien llamada *función variante*. Como indica el nombre, será una función que varia a lo largo de las iteraciones del ciclo del programa.

En particular, el hecho de que sea una función, es porque esta formada por las variables del programa. Es como un indicador de *cuanto falta* para que el programa termine su ejecución.

Formalmente, lo definimos como: sea $\mathbb{V}$ el producto cartesiano de los dominios de las variables del programa, llamamos *función variable* o f_V, a la funcion $f_V : \mathbb{V} \longrightarrow \mathbb{Z}$.

El teorema de terminación dice que, si existe una función f_V, un invariante I y v_0 un valor de f_V, tales que:

1. $\{I \wedge B \wedge f_V = v_o\}$ `S` $\{f_V < v_o\}$

2. $(I \wedge f_V \leq 0) \longrightarrow \neg B$

Entonces el programa en algun momento termina.

Ahora veamos todo esto en un ejemplo, supongamos el siguiente programa:

```
while (i <= n) do
  s := s + i;
  i := i + 1;
endwhile;
```

Y supongamos tambien, que tenemos de precondición $\{n \geq 0 \wedge i = 1 \wedge s = 0\}$ y de postcondición $\{s = \sum_{k=1}^{n} k\}$. Ahora necesitamos verificar que este programa es correcto, y en particular, que el ciclo del programa finaliza y cumple con la postcondición.

Empecemos demostrando la correctitud del ciclo con el teorema del invariante. Primero, definimos nuestro invariante como $\{1 \leq i \leq n+1 \wedge s = \sum_{k=1}^{i-1} k\}$. Suena como que me lo saque de la galera, lo se, pero si analizamos el ciclo del programa, podemos darnos cuenta que en particular estas son las cosas que efectivamente no cambian a lo largo de cada iteracion. Un consejo para buscar invariantes, es siempre tener en cuenta el indice y sus limites, y predicados que se encuentren en la postcondición, pero ajustados de modo que no cambien.

Vamos parte por parte. Primero hay que demostrar que $P \longrightarrow I$.

Como $i = 1$ y $n \geq 0$, en particular se cumple que $0 \leq i \leq n + 1$

Sabemos que $s = 0$, pero como $i = 1$, se tiene que $s = \sum_{k=1}^{i-1} k = \sum_{k=1}^{0} k = 0$

Y listo, utilizando solo los predicados de la precondición, deducimos el invariante. Ahora hagamos lo mismo con los demas casos. Tenemos que verificar $\{I \wedge B\}$ `S` $\{I\}$.

$\{i = I_0 \wedge s = S_0 \wedge 1 \leq I_0 \leq n + 1 \wedge S_0 = \sum_{k=1}^{I_0-1} k \wedge I_0 \leq n\}$

```
s := s + i;
```

$\{i = I_0 \wedge s = S_0 + I_0 \wedge 1 \leq I_0 \leq n + 1 \wedge S_0 = \sum_{k=1}^{I_0-1} k \wedge I_0 \leq n\}$

En particular, como $s = S_0 + I_0$ y $S_0 = \sum_{k=1}^{I_0-1} k$, tenemos que $s = (\sum_{k=1}^{I_0-1} k) + I_0 = \sum_{k=1}^{I_0} k$

$\{i = I_0 \wedge s = \sum_{k=1}^{I_0} k \wedge 1 \leq I_0 \leq n + 1 \wedge S_0 = \sum_{k=1}^{I_0-1} k \wedge I_0 \leq n\}$

```
i := i + 1;
```

$\{i = I_0 + 1 \wedge s = \sum_{k=1}^{I_0} k \wedge 1 \leq I_0 \leq n + 1 \wedge S_0 = \sum_{k=1}^{I_0-1} k \wedge I_0 \leq n\}$

En particular, si $i = I_0 + 1$, y $I_0 \leq n$, tenemos que $i \leq n + 1$

A su vez, si $i = I_0 + 1$ entonces $I_0 = i - 1$, por lo que $s = \sum_{k=1}^{I_0} k = \sum_{k=1}^{i-1} k$

Y finalmente, concluimos $\{1 \leq i + 1 \leq n + 1 \wedge s + 1 = \sum_{k=1}^{i-1} k\}$

Por ultimo, pasemos a demostrar $I \wedge \neg B \longrightarrow Q$.

$\{1 \leq i \leq n + 1 \wedge s = \sum_{k=1}^{I_0-1} k \wedge i > n\}$

Como $1 \leq i \leq n + 1$ y tambien $i > n$ tenemos que $i = n + 1$

Entonces, $s = \sum_{k=1}^{I_0-1} k = \sum_{k=1}^{n} k$

Que es lo que queriamos demostrar. Entonces terminamos la primera parte. El programa es *parcialmente* correcto, es decir, si el ciclo no se queda infinitamente iterando y en algun momento termina, el programa cumplira con la postcondición. Ahora analicemos si efectivamente termina.

Como función variante vamos a elegir $n - i + 1$, que tambien, va a aparecer sacado de la galera, pero tiene mucho sentido si miramos detalladamente la funcion. Consta del limite de iteracion (n), y una variable que sabemos va cambiando a lo largo del programa (i). Probemos punto a punto si funciona.

Primero hay que comprobar que $\{I \wedge B \wedge f_V = v_o\}$ `S` $\{f_V < v_o\}$.

En particular, del invariante nos interesa que $\{1 \leq i \leq n + 1 \wedge i \leq n \wedge f_V = n - i + 1\}$

```
s := s + i;
```

Queda igual, $\{1 \leq i \leq n + 1 \wedge i \leq n \wedge f_V = n - i + 1\}$, ya que ahora s no nos interesa

```
i := i + 1;
```

$\{1 \leq i + 1 \leq n + 1 \wedge i + 1 \leq n \wedge f_V = n - (i + 1) + 1\}$

En particular, $f_V = n-(i+1)+1$, por lo que $f_V = n-(i+1)+1 = n-i$, pero recordemos que $n-i+1 = v_0$, y entonces $f_V = n - i < n - i + 1 = v_0$.

Y asi, como vimos, demostramos que la función efectivamente decrece con cada iteración del ciclo. Para finalizar, vamos a verificar que sucede $I \wedge f_V \leq 0 \longrightarrow \neg B$.

$\{1 \leq i \leq n + 1 \wedge n - i + 1 \leq 0\}$

$\{1 \leq i \leq n + 1 \wedge n + 1 \leq i\}$

Veamos que $i \leq n + 1$ y al mismo tiempo $i \geq n + 1$, por lo que podemos deducir que $i = n + 1$. Y esto implica totalmente a $\neg B$, ya que $B \equiv i \leq n \longrightarrow \neg B \equiv i > n$, y si $i = n + 1$, en particular se cumple que $i > n$, como nosotros queriamos.

Entonces ya esta. Encontramos una funcion variante que satisface las dos condiciones del teorema de terminacion, por lo que podemos afirmar que, en algun momento, el ciclo va a terminar; el programa no se va a *colgar*.

En particular, como demostramos tanto la correctitud como la terminación del programa, podemos finalizar diciendo que el programa es correcto completamente. Satisface la postcondicón y finaliza su ejecución.

4 Tipos Abstractos de Datos

4.1 La abstracción de la realidad

Muchas veces, como programadores, no vamos a tener un objetivo tan claro como *contar todos los elementos de una lista* o *definir un conjunto con ciertas propiedades*. De hecho, generalmente el objetivo del programa va a ser algo como *hacer un buscaminas* o *modelar un restaurante*, y la cosa se nos complica.

Para estos casos aparecen los llamados *tipos abstractos de datos* (de ahora en adelante, TADs). Estas son estructuras construidas en el lenguaje de especificación que representan ideas u objetos mas complejos que los tipos de datos que poseemos, pero que intentan modelar sus propiedades y comportamientos.

Quedense con esta idea clave: *modelar*. El objetivo de un TAD es representar de la manera mas fiel posible al concepto original. Esto nos va a servir para, a la hora de posteriormente implementar dicho concepto en un lenguaje de programacion, poder demostrar su correctitud mucho mas facil.

La estructura de un TAD consiste en dos partes clave: los observadores y las funciones (o procedimientos, seria mas adecuado).

4.2 Observadores y propiedades

Los observadores son estructuras que nos permiten visualizar algun dato del TAD en cuestion. Es decir, nos permiten acceder a alguno de los datos (o un conjunto de los mismos) que componen al TAD. El objetivo de los observadores es ayudarnos a discernir una instancia del objeto de otra.

Es decir, que dadas dos instancias del tipo que queremos representar, solo en base a los observadores deberiamos ser capaces de diferenciar uno de otro, o, lo que es lo mismo, afirmar si son o no iguales.

Por ejemplo, si quisieramos representar puntos del plano cartesiano, y crearnos un TAD *punto*, para ya tener operaciones y propiedades definidas, podriamos hacer lo siguiente:

```
TAD Punto {
    obs x : ℝ
    obs y : ℝ
}
```

Pero tambien, podriamos representar un punto por las coordenadas polares: con ρ y θ. En ese caso, nuestro TAD quedaria:

```
TAD Punto {
    obs ρ : ℝ
    obs θ : ℝ
}
```

Podrian estar pensando ahora *hey, pero si con ambas nos es suficiente, ¿por que no utilizar ambas a la vez?*, a lo que yo contestaria *porque es una mala practica*. Basicamente, la idea de los observadores es utilizar la menor cantidad posible, es decir, poder deducir informacion con la menor cantidad de pistas posible. No es que nos haga daño usar informacion "redudante", pero se la considera mala practica por eso mismo. Por ejemplo, podriamos deducir ρ conociendo los valores de x y de y, y es identico con θ. Analogamente, podriamos hacer el camino inverso. Por esta misma razon, con utilizar una de las dos representaciones nos alcanza.

Teniendo esto en cuenta, podemos afirmar que dos instancias del tipo *punto* son iguales si y solo si sus observaciones son identicas. A este hecho lo llamamos *igualdad observacional*.

Y asi, dado un punto, podemos observar su coordenada x y su coordenada y. No es muy util esto, es decir, ahora mismo el TAD no hace nada, pero podemos agregarle propiedades a los puntos. Para definir propiedades, podemos formular procedimientos como lo venimos haciendo, con el lenguaje de especificacion, utilizando funciones auxiliares y demas. A su vez, teniendo en cuenta todas las precauciones que tambien vimos.

Continuando con el ejemplo, agreguemos unas funciones basicas.

```
TAD Punto {
    obs x : ℝ
    obs y : ℝ
```

```
    proc crearPunto(in x1 : ℝ, in y1 : ℝ, out res : Punto){
        asegura { res.x = x1 ∧ res.y = y1 }
    }
}
```

Aca se puede ver como, dados dos numeros reales, podemos generarnos una *cosa* de tipo *punto* (¡Hemos creado nuestro propio tipo de dato!). Ahora mismo solo podemos crear puntos, pero podemos agregar funciones mas utiles, como calcular el modulo de un punto. Eso se veria asi:

```
TAD Punto {
    obs x : ℝ
    obs y : ℝ
    proc crearPunto(in x1 : ℝ, in y1 : ℝ, out res : Punto){
        asegura { res.x = x1 ∧ res.y = y1 }
    }

    proc modulo(in p : Punto, out res : ℝ){
        asegura { res = √((p.x)² + (p.y)²) }
    }
}
```

Nos estamos tomando la libertad de no escribir los *requiere* ya que en todos estos casos son igual a $True$, en caso de necesitarlos, tambien habria que escribirlos.

Bien, ya estamos escribiendo nuestros propios TADs, y somos capaces de agregarles propiedades y comportamientos. Ahora levantemos un poco la dificultad, para molestar un poco, supongamos que queremos crear dos TADs: uno de *estudiante* y otro de *materia*.

Vamos a basar un estudiante en tres cosas: su nombre, la edad y la carrera que esta estudiando. Por otro lado, la materia va a ser alguna materia a la que vamos a poder pedirle de que carrera forma parte, y que estudiantes la estan cursando.

A su vez, vamos a querer definir una lista de funciones: queremos verificar que el estudiante sea mayor de edad, y dada una materia, ver si esta cursando esa materia.

```
TAD Estudiante {
    obs nombre : String
    obs edad : ℤ
    obs carrera : String
    proc esMayor(in e : Estudiante, out res : Bool){
        asegura { res =True⟷ e.edad ≥ 18 }
    }

    proc estaCursando(in e : Estudiante, in m : Materia, out res : Bool){
        asegura { res =True⟷ ((∃i : ℤ)(0 ≤ i < |mostrarCursantes(m)| ⟶
                    mostrarCursantes(m)[i].nombre = e.nombre)) }
    }
}

TAD Materia {
    obs cursantes : seq⟨Estudiante⟩
    obs carrera : String
    aux mostrarCursantes (in m : Materia) : seq⟨Estudiante⟩ = m.cursantes
}
```

Parecia complicado, pero no nos quedo algo tan enrevesado. De aca, uno puede especificar lo que quiera. Podemos utilizar la propiedad de carrera para clasificar los alumnos segun su carrera, o comparar la cantidad de estudiantes de cada materia. Esto es solo un ejemplo basico.

Notemos como la función *mostrarCursantes* es una función auxiliar. Esto es asi, ya que si lo hubieramos hecho un procedimiento, no podriamos luego utilizarlo en el TAD de *estudiante*. Recordemos, un procedimiento no puede utilizarse dentro de otro, y aca se conserva esa regla. Si podemos, por otro lado, usar funciones auxiliares o predicados, pero no procedimientos. Los observadores son exclusivos de la definición del TAD, asi que tampoco podemos usarlos en otros TADs.

5 Diseño de algoritmos

5.1 ¿Donde está el diseño?

Nosotros vimos dos etapas enormes a la hora de trabajar en el mundo de la programación: la *especificación* y la *implementación*. Esto se debe a la simpleza de los problemas que queriamos resolver, pero en el mundo real existen problemas mas complejos o, sin ir mas lejos, TADs que vamos a querer representar. Esta clase de problemas va a requerir un paso intermedio, algo que nos ayude a resolver el problema antes de implementarlo, una forma de unir el mundo de la especificación y el de la implementación. Es en este contexto, donde entra la etapa de *diseño*.

Esta etapa de diseño consiste en una union entre la especificación y la implementación, como dijimos antes. Va a haber funciones que tomen por parametros tipos de la especificación, y devuelvan tipos de implementación, o viceversa. A su vez, la sintaxis a veces puede resonar como smallLang, pero otras puede parecerse a la lógica matematica que venimos viendo. Va a ser un poco abstracto, porque nos estamos tomando muchas libertades para que sea lo mas ameno posible.

A su vez, lo que vamos a intentar dar a entender en este paso es el *como* tenemos planeado resolver los problemas planteados. Es decir, vamos a dar con nuestro algoritmo, nuestro plan de ataque. Mas adelante veremos si ese plan de ataque es bueno o malo, segun los criterios de la complejidad.

5.2 De regreso a los TADs

Los TADs son una estructura sumamente compleja. Son un conjunto de operaciones y estructuras de datos, que representan ideas mas abstractas y no tan tangibles, como un numero, una lista o una palabra.

Supongamos que queremos diseñar el TAD mostrado antes, el de *estudiantes*. Lo primero que hariamos seria utlizar las palabras reservadas *modulo* y *implementa*, en la que indicamos el nombre de nuestro modulo a diseñar y de que TAD proviene. En nuestro caso:

```
Modulo estudianteImpl implementa Estudiante {
    nombre : String
    edad : int
    materiasCursando : array[String]
}
```

Noten que escribimos tambien las variables que creemos vamos a necesitar. Estos van a ser un analogo a los observadores de los TADs, pero van a estar en el tipo de la implementacion. Es decir, ya no tenemos secuencias o conjuntos, sino *arrays*, que tienen un tamaño fijo y no dinamico. Si quisieramos que no haya repetidos, por ejemplo, tendriamos que asegurarnos de eso en como diseñemos nuestro modulo.

Podemos hacer aclaraciones afuera del modulo, como cambios de nombres que nos sean mas comodo. Por ejemplo, vamos a reemplazar uno de los *String*, por *Materias*. Es un reemplazo sintactico nada mas, pero va a facilitar la lectura. De ese modo, quedaria:

```
Materia es String

Modulo estudianteImpl implementa Estudiante {
    nombre : String
    edad : int
    materiasCursando : array[Materia]
}
```

Las operaciones, que seguiremos llamando procedimientos, podemos escribirlas ahora en *SmallLang*, o cualquier otra estructura de pseudocodigo. Lo importante es que se entienda nuestro enfoque para resolver estos problemas. Si quisieramos escribir las operaciones de *esMayor* y *estaCursando*, podriamos escribir lo siguiente:

```
Materia es String

Modulo estudianteImpl implementa Estudiante {
    nombre : String
    edad : int
```

```
    materiasCursando : array[Materia]
      proc esMayor(in e : estudianteImpl, out res : Bool){
          res = e.edad >= 18
      }

      proc estaCursando(in e : estudianteImpl, in m : Materia out res : Bool){
          res = False
          i := 0
          while i < e.materiasCursando:
            if e.materiasCursando[i] == m:
              res = True
      }
}
```

Noten que nos tomamos muchas libertades con la escritura, por ejemplo, no escribimos los *endif* y *endwhile*, ni los punto y coma. Reiteramos en que, esto no es una implementación, es un diseño. Nuestro objetivo es analizar el comportamiento del algoritmo. Aun asi, si quisieramos, podriamos demostrar correctitud, o buscar los invariantes de ciclos o lo que quisieramos realmente.

Ahora bien, a nosotros nos gustaria verificar que nuestro diseño es correcto. Es decir, efectivamente identifica y representa al TAD que estamos analizando. Para eso, existen dos funciones importantes a tener en cuenta.

La primera es el *invariante de representación*. Es parecido al invariante del ciclo que vimos antes, pero no igual. Basicamente, un *invariante de representación* es un predicado o conjunto de predicados que, al ejecutar el programa, no cambia, no se ve afectado. A su vez, nos tiene que estar diciendo *algo* sobre el TAD que queremos evaluar.

Es decir, el *invariante de representación* es un conjunto de predicados que hablan sobre el TAD y que nunca deberian romperse, nunca deberian ser falsos.

Por ejemplo, supongamos que tenemos un TAD para definir un circulo, y lo tenemos escrito asi:

```
TAD Circulo {
    obs centro : Punto
    obs radio : ℝ
}
```

Ahora bien, sin importar el programa que estemos evaluando (recordemos, no estamos implementado todavia), nos interesa que la propiedad *radio* nunca sea negativa, ya que entonces nuestro objeto circulo no tendria sentido. Ese seria un buen invariante de representacion, entonces, tenemos que:

$$I_{rep} \equiv radio > 0$$

Ahora bien, un invariante de representación esta definido sobre el diseño de un TAD. Es decir, es un predicado que, entre sus parametros, esta la instancia de un TAD implementado. Supongamos que, estamos pensando en la implementación del circulo, y se nos ocurre que podriamos hacerlo con una variable de tipo *punto* que sea un centro, y otra de tipo $\mathbb{R}$ que sea el radio. Entonces, el diseño del TAD circulo se veria:

```
Modulo CircImpl implementa Circulo {
    centro : Punto
    radio : ℝ
}
```

Entonces, su invariante de representación podemos escribirlo como:

```
pred InvRep ( in objeto : TADImpl ) {
    objeto.radio ≥ 0 ⟷ res = True
}
```

Y eso deberia cumplirse antes de la ejecucion del codigo, y despues, es decir, queremos que se cumpla:

$$\{P \wedge I_{rep}\} \quad \texttt{S} \quad \{Q \wedge I_{rep}\}$$

Nunca vamos a querer evaluar casos que no cumplan con la precondición o que no lleguen a la postcondición, asi que los incluimos en la tripla de Hoare.

Otra propiedad util es la conocida como *función de abstracción*, que consiste en una funcion $T_{impl} \longrightarrow T$, donde T_{impl} es el tipo del TAD implementado, es decir, su modelo hecho programa; y T es el tipo del TAD a representar.

Retomando el ejemplo del circulo, deberia tomar por parametro de entrada un circulo implementado y devolvernos un circulo abstracto del TAD, es decir:

`aux` $FuncAbs\ (in\ c' : CircImpl) : Circulo = (c'.centro = res.centro \wedge c'.radio = res.radio)$

Quizas no parezca muy interesante en principio, pero lo notable aca es que recibimos un parametro de tipo *circulo implementado* y recibimos su *representación original* del TAD. Para que la función de abstracción tenga sentido, requerimos que los observadores sean suficientes para distinguir una instancia de otra (lo que dijimos al principio).

5.3 Complejidad

A la hora de diseñar algoritmos y estructuras de datos, tenemos que procurar dos cosas clave: que el algoritmo en cuestion efectivamente resuelva el problema (lo que venimos viendo hasta ahora); y que sea eficiente. Ahora bien, ¿que significa la *eficiencia*?

Si tuvieramos dos algoritmos, y uno tarda 2 horas, y el otro tardase 1 hora, es facil advertir cual de los dos es mejor: el segundo. Tarda menos, evidentemente lo que sea que haga lo hace *mejor*.

Supongamos que tenemos un programa, y nos interesa determinar su duración en base al tamaño del input. Es decir, queremos ver la relación entre el tamaño de la entrada (por ejemplo, que tan larga es una lista de entrada) y el tiempo que consume resolver el problema (por ejemplo, cuanto tarda en encontrar un elemento en esa lista). Este concepto de relación entre el tamaño de la entrada y el tiempo que consume, es al que llamamos *eficiencia*. A mayor eficiencia, menor tiempo consumira con inputs grandes.

Nota: En el ejemplo hablamos de la duración temporal del programa, pero podriamos estar buscando reducir otra cosa (tamaño en memoria, consumo de energia, entre otras particularidades).

Esta medida de la eficiencia es la que nos va a permitir elegir entre dos o mas algoritmos, poder categorizarlos y evaluarlos, determinar cual es mejor o peor que otro segun la circunstancia que queramos medir.

Ahora bien, medir el tiempo de un programa no es tarea facil. Uno puede hacerlo de dos maneras: midiendolo *a mano*, es decir, cronometrando el programa al correrlo; o medirlo de manera teorica en base a su comportamiento. La primera medida tiene varios posibles contratiempos. A saber, quizas cambie la medicion segun la computadora que corre el programa, siendo una mas potente que la otra, o quizas depende del lenguaje en el que este escrito pero no del algoritmo en si. La segunda medida, por otro lado, es independiente a la computadora o al lenguaje, y vale para cualquier instancia del programa.

Vamos a basarnos en un concepto importante para medir la complejidad: las operaciones elementales (OE). Estas van a ser instrucciones que vamos a considerar con tiempo 1 (no tiene unidad definida). El tiempo de un programa, entonces, será la suma de tiempos de cada instruccion.

Un poco de notación: vamos a llamar $T(n)$ a la complejidad temporal para un tamaño de entrada n, y por otro lado, $S(n)$ denotará la complejidad espacial para un tamaño de entrada n. Finalmente, llamaremos $t(S)$ al tiempo que tarde en ejecutar el programa S.

De lo anterior se deducen las siguientes reglas:

- $t(\texttt{If C Then S1 Else S2 Endif;}) = t(C) + max\{t(S1), t(S2)\}$
- $t(\texttt{Case C Of v1:S1 | v2:S2 | …| vn:Sn End;}) = t(C) + max\{t(S1), t(S2), \ldots, t(Sn)\}$
- $t(\texttt{While C Do S End;}) = t(C) + (ndeiteraciones) \times (t(C) + t(S))$
- $t(\texttt{MiFuncion(P1, P2, …, Pn)}) = 1 + t(P1) + t(P2) + \ldots + t(Pn) + t(F)$

En el ultimo caso tenemos una llamada a función, donde toma n parametros, y hace la operación F.

Hay que tener una cosa mas en cuenta: entradas de mismo tamaño pueden tener medidas de tiempo diferentes. Por ejemplo, supongamos que queremos evaluar la complejidad de un algoritmo que busca un elemento en una lista, y lo que hace es chequear uno a uno entre los elementos y compararlos con el elemento en cuestión. Si la lista es mas larga, el algoritmo llevara mas tiempo en resolverse, salvo que el elemento este

al principio de la lista. Es decir, si preguntamos por el primer elemento de la lista, por mas larga que esta sea, vamos a medir siempre el mismo tiempo (el minimo, en este caso).

Es por esta situación que nos interesan tres casos particulares de la medición del tiempo: el caso *peor*, el caso *mejor* y el caso *medio*. Podemos construirlos de la siguiente manera:

- $T_{peor}(n) = max_{instanciasI,|I|=n}\{t(I)\}$
- $T_{mejor}(n) = min_{instanciasI,|I|=n}\{t(I)\}$
- $T_{prom}(n) = \sum_{instanciasI,|I|=n}\{P(I) \times t(I)\}$

P(I) en este contexto representa la probabilidad de que un input sea la instancia I.

5.4 Análisis asintótico de la complejidad

En general, no nos interesa saber el tiempo *exacto* de un algoritmo, sino que nos interesa ver su crecimiento. Es decir, nos interesa su *orden de magnitud*. De esa forma, vamos a poder evaluar mas facilmente entre dos algoritmos que, a priori, tienen medidas distintas, pero quizas mismo orden de magnitud, o viceversa.

Entonces, decimos que el orden (logarítmico, lineal, cuadrático, exponencial, etc) de la función $T(n)$ es el que expresa el comportamiento dominante cuando el tamaño de la entrada es grande. En otras palabras, es lo que nos interesa realmente.

Como tal, lo que nos interesa es el *análisis asintótico* de la complejidad, que es el comportamiento de la misma para valores de entrada suficientemente grandes. Nosotros vamos a aproximar estos valores con cotas (de ahi lo asintótico), las cuales denotamos:

O (O grande) es la corta superior

Ω (omega) es la corta inferior

θ (theta) es el orden *exacto* de la función

Estas cotas van a determinar los limites de la complejidad del algoritmo que estemos analizando. La notación $f \in O(g)$ indica que la función f no crece más rápido que alguna función proporcional a g. Estamos diciendo que $O(g)$ es un conjunto de funciones, y lo definimos como:

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

Analogamente, definimos a *omega* como:

$$O(g) = \{f | \exists n_0, k > 0 \text{ tal que } n \geq n_0 \longrightarrow f(n) \geq k \cdot g(n)\}$$

Finalmente, a *theta* lo expresamos asi:

$$O(g) = \{f | \exists n_0, k_1, k_2 > 0 \text{ tal que } n \geq n_0 \longrightarrow k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)\}$$

Una vez teniendo estas definiciones podemos empezar a deducir muchas de las propiedades que vamos a utilizar a la hora de resolver ejercicios de complejidad. Aca listamos unas cuantas:

- Si existe $\lim_{n \longrightarrow \infty} \frac{f(x)}{g(x)} = k$, y segun los valores de k:

  Si $k \neq 0$ y $k < \infty$, entonces $\Omega(f) = \Omega(g)$, $\theta(f) = \theta(g)$ y $O(f) = O(g)$

  Si $k = 0$, entonces $\Omega(f) \neq \Omega(g)$, $\theta(f) \neq \theta(g)$ y $O(f) \neq O(g)$
- Si $f_1 \in O(g)$ y $f_2 \in O(h) \Longrightarrow f_1 + f_2 \in O(max\{g, h\})$
- Si $f_1 \in O(g)$ y $f_2 \in O(h) \Longrightarrow f_1 \cdot f_2 \in O(g \cdot h)$
- Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Longrightarrow f_1 + f_2 \in \Omega(g + h)$
- Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Longrightarrow f_1 \cdot f_2 \in \Omega(g \cdot h)$
- Si $f_1 \in \theta(g)$ y $f_2 \in \theta(h) \Longrightarrow f_1 + f_2 \in \theta(g + h)$
- Si $f_1 \in \theta(g)$ y $f_2 \in \theta(h) \Longrightarrow f_1 \cdot f_2 \in \theta(g \cdot h)$

Existen muchas propiedades mas, pero estas suelen ser las mas utiles.

Ahora, ¿como se relaciona todo esto con la eficiencia de programas? Bueno, ahi es donde entra lo que definimos antes de la función $T(n)$. Vamos a analizar estas funciones temporales en base a estas cotas, utilizando la notación antes mostrada. A saber, si tenemos que:

$$T_{peor} \in O(g)$$

Decimos que el peor de los casos (la peor instancia de un tamaño n), la función temporal $T(n)$ no va a superar a $g(n)$. Es decir, es su cota superior. Podriamos hacer analisis similares con las distintas cotas, son bastante analogas. Tambien, podemos utilizar lo que vimos antes de los casos peor, mejor y promedio.

Vamos a mencionar casos conocidos o que ya vimos y mostrar su correspondiente complejidad.

Para empezar, tenemos el famoso caso de *buscar un elemento en una lista*. Si la lista no tiene una forma de orden, nos va a ser muy dificil encontrar una forma *eficiente* de encontrar un elemento en ella. Lo unico que podemos hacer es ir chequeando uno a uno, y compararlo con el elemento en cuestion. En el peor de los casos, terminamos chequeando todos los elementos (o bien el elemento esta en la ultima posicion de la lista, o bien ni siquiera pertenece). Eso quiere decir que, si la lista mide n, podemos decir que el algoritmo de busqueda tiene una complejidad $O(n)$, ya que es el peor de los casos. A su vez, el mejor de los casos es que el elemento si pertenezca, y este en la primera posicion. Por lo que podemos añadir que el algoritmo tiene $\Omega(1)$.

Ahora, supongamos que la lista esta ordenada. Por ejemplo, es una lista de numeros enteros y se encuentra ordenada de menor a mayor. En este caso podriamos llegar a provecharnos de esto y usarlo a nuestro favor. No hace falta chequear uno a uno cada elemento, podriamos dividir la lista a la mitad, y ver si el elemento que estamos buscando pertenece a una o a la otra. Para hacer esto, podriamos preguntar si el elemento es mayor que el valor del medio. Al estar la lista ordenada, podemos afirmar que esto nos separa la lista en dos, y que con ese dato ya sabemos a que mitad pertenece. Noten que esto se puede hacer repetidas veces hasta llegar al elemento en cuestion (si pertenece, en caso contrario, llegariamos a un numero al que no podriamos llegar, o lista vacia). Si hacen las cuentas que ahora no vamos a detallar, podemos llegar a que, en el peor de los casos, la complejidad de este algoritmo es de $O(log_2(n))$, lo cual es mucho mejor que $O(n)$. A este algoritmo se le conoce como *busqueda binaria*.

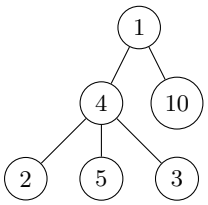
Como veran, el simple dato de que la lista esta ordenada, ya nos ayuda a agilizar el procedimiento. El problema ahora seria procurar que la lista siempre se encuentre ordenada.

6 Arboles

6.1 Una estructura nueva

Existe una estructura de datos que es muy versatil y util, y es a la que le vamos a dedicar todo este capitulo. Volvamos al ejemplo de antes, el de buscar numeros en una lista que esta ordenada. Existen otras formas de representar (u ordenar, mejor dicho) estructuras que funcionen semejante a esta lista ordenada. Los que vamos a ver aca, como dice el titulo del capitulo, son los *arboles*.

Estos son estructuras que consisten de nodos (los numeros) y conexiones (flechitas que los unen). Visualmente los representamos asi:



A cada "burbuja" la llamaremos *nodo*, y las flechitas nos van a indicar las conexiones entre cada nodo. Decimos que un nodo es *padre* de otro, cuando tiene una flecha que apunta a ese otro nodo. De igual manera, al segundo nodo lo llamaremos *hijo* del primero. En el ejemplo de antes, 1 es padre de 4 y 10, y 4 tiene 3 hijos: el 2, el 5 y el 3.

A su vez, continuando la analogia con los *arboles*, decimos que una *rama* es cualquier nodo con algun padre que tenga por lo menos un hijo. A su vez, llamamos *hoja* a los nodos que no poseen hijos. En el dibujo, diriamos que 2, 5 y 3 son hojas, y que 4 y 10 son ramas. Pero nos falta el 1, donde nace el arbol. A este nodo particular lo indicaremos como la *raiz* del arbol. Entonces, 1 es nuestra raiz.

Finalmente, otro dato que mas adelante nos va a resultar interesante es la *profundidad* del arbol. Consiste basicamente en el largo de la rama mas larga, es decir, la cantidad de nodos que hay entre la raiz y la hoja mas lejana.

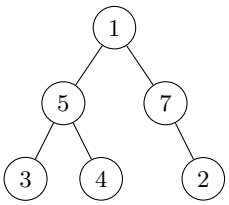
A nosotros, con todo esto, nos interesa representar conjuntos. Por lo tanto, no nos interesan los arboles con valores repetidos, asi que vamos a evitarlos a toda costa. Vamos a centrarnos en los arboles con *valores unicos*.

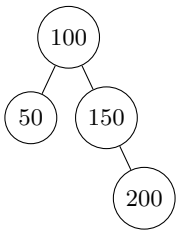
A su vez, otra cosa que nos interesa es categorizar los arboles en tipos. Una forma de hacer esto es contando la cantidad *maxima* de hijos. Es decir, cual es el nodo (o los nodos) con mas hijos. De esta forma, podriamos tener arboles *binarios*, donde los nodos solo pueden tener 0, 1 o 2 hijos; o arboles *ternarios*, donde llegan hasta 3; y asi podriamos seguir hasta lo que nos interese. En el ejemplo de antes, el arbol que dibujamos seria un arbol ternario, ya que el 4 tiene 3 hijos.

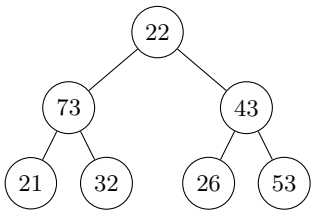
En particular, vamos a estudiar el caso de los *arboles binarios*, ya que tienen muchas propiedades que nos van a ser de utilidad.

6.2 Arboles Binarios de Busqueda

Existen infinitos arboles binarios. Por ejemplo, podriamos dibujar los arboles:





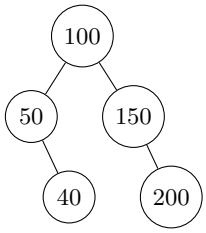


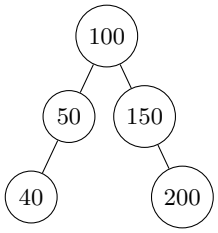
Ahora, como notaran, nos tenemos mucho control sobre el orden o la cantidad de hijos de cada nodo. Podriamos armarlos de manera super arbitraria, y de antemano no podriamos afirmar con certeza nada de ningun arbol.

Es por eso que nos vamos a interesar en un tipo concreto de arbol binario: los *arboles binarios de busqueda* (a veces escritos como ABB, por sus siglas), que son arboles que nos van a facilitar la busqueda de elementos, como su nombre bien indica.

La logica detras de un arbol binario de busqueda es la siguiente: todo nodo va a tener a su izquierda numeros *menores* a ese nodo, y a su derecha numeros *mayores*. Viendo los arboles de antes podemos afirmar, entonces, que el arbol de en medio es un arbol binario de busqueda, y los otros dos no, ya que hay nodos mayores o menores por todos lados.

Notemos que esta propiedad de busqueda se tiene que cumplir para *todos* los nodos del arbol, no solamente la raiz. Por ejemplo, si tuvieramos el arbol:





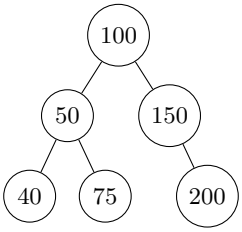
El de la izquierda no seria un arbol binario de busqueda, ya que a la derecha del 50 hay un 40, por mas que el 40 se encuentre a la izquierda del 100. Para solucionar esto, tendriamos que ubicar el 40 a la izquierda del 50, y asi lograr acordar la definicion de los ABB, como es el caso del arbol de la derecha.

¡Bien! Ya aprendimos lo que es un arbol binario de busqueda, ahora ¿que provecho podriamos sacarle? Bueno, esta es una respuesta muy interesante.

Resulta que buscar un elemento en una estructura asi, es super facil. Lo primero que haremos es preguntarnos si el elemento que queremos averiguar es efectivamente la raiz. En un caso menos interesante, esto seria afirmativo, pero en el mas interesante pasariamos al paso dos. Nos preguntariamos si el elemento es mayor o menor que la raiz. Si es menor, bajaremos por la rama izquierda, y si es mayor, bajamos por la rama derecha. Y ahora volvemos a empezar, ¿es el elemento que estamos buscando el que ahora estamos viendo? Caso afirmativo, lo encontramos. Caso contrario, chequeamos si es mayor o menor, y bajamos a la rama adecuada. Si en algun momento llegamos a alguien que no tiene un hijo correspondiente (por ejemplo, el elemento es mas chico pero el nodo no tiene hijo izquierdo, o viceversa), quiere decir que el elemento no pertenece al arbol.

De manera semejante, agregar un elemento es bastante sencillo. Solo hacemos el mismo recorrido que al buscar un elemento pero, cuando encontremos un *espacio disponible*, metemos al elemento ahi. Es decir, le asignamos de padre al ultimo nodo que hayamos encontrado mayor o menor que el elemento nuevo. Obviamente, el nodo nuevo tiene que no pertenecer actualmente, ya que como dijimos antes, no queremos repetidos.

Por ejemplo, si en el arbol de antes quisieramos ingresar el 75, el arbol nuevo seria:

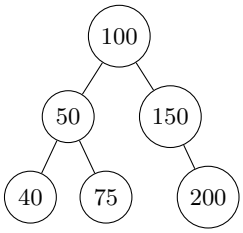


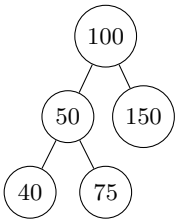
El problema mas grande con estas estructuras parte de eliminar elementos. Una vez aclarado que el nodo pertenece y lo tengamos ubicado, vamos a tener tres posibles casos:

- Que el nodo eliminado sea una hoja (no tenga hijos)
- Que el nodo eliminado tenga un solo hijo
- Que el nodo eliminado tenga dos hijos

El primero de los casos es el mas facil. Si el nodo que queremos eliminar se trata de una hoja, entonces solo necesitamos decirle a su padre que ya no lo apunte. Es decir, indicarle al nodo padre que, en esa direccion, ya no tiene un hijo.

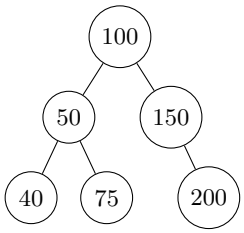
En el ejemplo de antes, si quisieramos borrar una hoja (por nombrar alguna, el 200), hariamos lo siguiente:

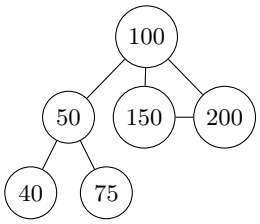


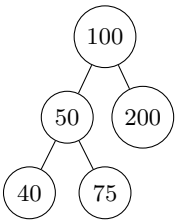


En el caso donde el nodo tiene un unico hijo, tambien es relativamente sencillo. Lo que tenemos que hacer es *saltearlo*, desde su padre hasta su propio hijo. Tambien podemos pensarlo como que, la rama que quedo "colgando" la *subimos* para continuar el papel de su padre liminado.

Continuando con el ejemplo, si quisieramos borrar el 150, que tiene de hijo al 200, hariamos:

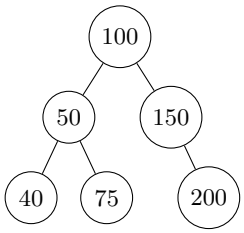


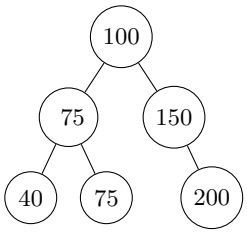


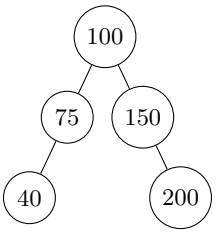


Y finalmente, el caso mas dificil, o realmente el mas largo: cuando el nodo eliminado tiene dos hijos. Escencialmente, lo que haremos sera reemplazar al nodo eliminado por su sucesor. Es decir, vamos a ver a los nodos en orden numerico y, para no romper la estructura interna del arbol, vamos a usar el nodo inmediatamente siguiente del que queremos eliminar.

Para verlo con un ejemplo, en nuestro arbol de antes, queremos eliminar el 50. Su siguiente en este arbol es el 75, asi que escribimos 75 en el lugar del 50, y borramos el viejo 75.







Es probable que ahora piensen "¿pero que pasa cuando el sucesor de un numero no es tan obvio? ¿cuando el arbol es mas grande o engorroso?". A lo que yo diria "que bueno que exista tambien un algoritmo para eso". Lo primero que hay que hacer, para buscar al sucesor, es preguntarnos si el nodo en cuestion tiene una rama derecha. Es decir, si hay numeros mas grandes por debajo de el. Si eso es afirmativo, entonces muy facil: simplemente buscamos el minimo de ese *subarbol*. El sucesor de un numero es el minimo de los numeros mas grandes que el.

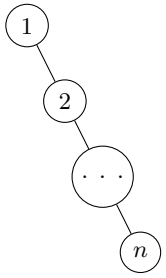
Vamos a tener un problema cuando el nodo que queremos borrar no tiene rama derecha, aunque no es tan complicado. Basicamente, lo que haremos sera utilizar el mismo procedimiento de busqueda, el de pertenencia. Pero cada vez que estemos "parados" en un nodo, nos guardaremos ese nodo si es mas grande que el nodo que queremos eliminar. Si lo piensan, es la misma logica, buscar al minimo de los mas grandes que el propio nodo, solo que empezando desde la raiz del arbol.

Como ven, la eliminación de elementos es un poco engorrosa, pero es un sacrificio que estamos dispuestos a pagar para que la busqueda o la inserción sean elegantemente eficaces.

Ahora bien, si se dan cuenta, dado cualquier arbol de n nodos, podemos empezar a calcular la complejidad de las distintas operaciones que venimos planteando. Por ejemplo, determinar si un elemento pertenece o no, en base al algoritmo antes mencionado, nos costaria en el peor de los casos $O(n)$. ¿Cual es el peor de los casos? Un arbol donde solo se crece en una direccion, es decir, un arbol que su rama mas larga (su *profundidad*) mide n, como se ve en la imagen siguiente.

En este arbol se puede ver que, si continuara indeterminadamente, nos costaria n pasos, suponiendo que preguntamos por el nodo mas grande, o alguien que no pertence al arbol. Pasaria lo mismo si la rama creciera indefinidamente hacia la izquierda, es decir, si fueran elementos cada vez menores. El peor caso seguiria siendo el mismo.

Notemos que a la hora de insertar un elemento, tenemos el mismo problema: nuestro peor caso seria querer ingresar un elemento al final.



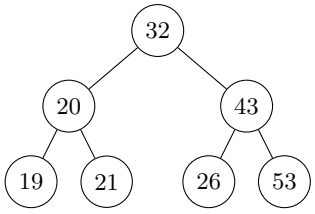
Lo que si nos complica mucho es la eliminación, donde claramente el peor de los casos es la situación del nodo eliminado teniendo dos hijos. Este calculo es algo mas dificil, pero si revisamos los pasos anteriores nos dara la misma situacion: $O(n)$.

Ahora bien, si recuerdan el capitulo anterior donde hablamos de las listas ordenadas, no parece que mejoramos mucho la situación. Es mas, parece que complicamos mas la estructura para obtener el mismo resultado. Pero resulta que esto es solamente la primera parte. Vamos a explorar un tipo concreto de arbol binario de busqueda, pero antes, tenemos que hablar del *balanceo* de los arboles binarios.

6.3 Balanceo de los arboles binarios

Como vieron antes, necesitamos una forma de asegurar una estructura al arbol, ya que sino nos pueden molestar con arboles feos o que no sirven. Nuestro primer acercamiento a resolver este problema podria ser el de plantear un *arbol completo*.

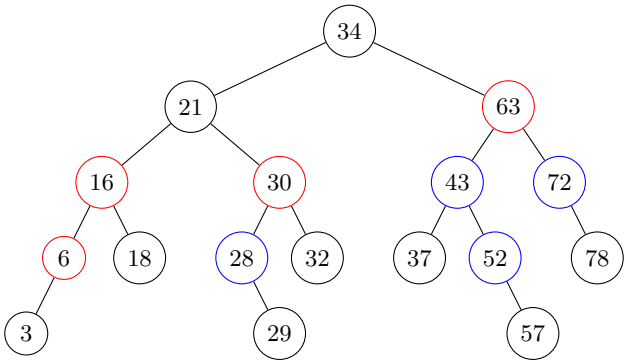
Llamamos arbol completo a los arboles que todos sus nodos tienen 2 hijos, a exepcion de los del ultimo nivel, que tienen 0. Por ejemplo:



Si sabemos que el arbol esta completo, podemos ver que se nos solucionan los problemas de complejidad de antes. Podemos afirmar que, a lo mucho, buscar un nodo tiene complejidad $O(\log n)$. Pero tenemos un nuevo problema ahora: el de mantener completo al arbol. Imaginemos que queremos ingresar el elemento 17 a este arbol, ¡vamos a tener que ingresar otros 7 nodos nuevos para que siga completo!

Por ende, decimos que mantener un arbol completo es *caro*, en terminos de espacio y tiempo. Nota de vital importancia: vean que los nodos del "piso del fondo" son mas del 50% de todo el arbol.

Ahora bien, quizas no necesitamos que el arbol este totalmente completo, pero si lo *suficientemente* completo. Basicmanete lo que queremos es que todas las ramas de un arbol tengan *casi* la misma longitud. Pero, ¿como definimos este *casi*?



Llamamos a un arbol *balanceado*, cuando para cualquier nodo en el, la diferencia de longitud entre sus ramas izquierda y derecha difiere, a lo sumo, en una unidad. Es decir, calculamos su *factor de balanceo*, que

consiste en $A_D - A_I$ (A_x es la altura del subarbol x), y vemos si nos da 1, 0 o -1.

En el ejemplo de arriba, podemos ver los factores de balanceo de cada nodo en base a sus subarboles. Por un lado tenemos los nodos de color negro, que tienen factores de balanceo igual a 0, es decir, sus ramas mas largas izquierda y derecha miden lo mismo. Los de color rojo son los que tienen un factor de balanceo de -1, que significa que la rama izquierda es un nodo mas larga que la derecha. Finalmente, los azules tienen un factor de 1, lo que nos dice que la rama derecha es un nodo mas larga que la izquierda.

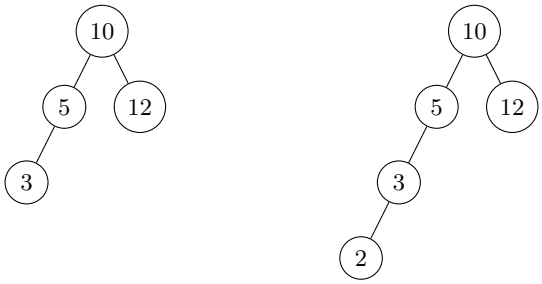
Un arbol que cumple esta condición lo llamaremos *árbol balanceado en altura*, o tambien conocido como AVL, por sus creadores (dos rusos a los que no voy a intentar escribir su nombre, por respeto y porque lo haria mal). El arbol antes visto, seria un AVL.

Notemos que, volviendo a lo que nos interesa, si sabemos que tenemos un AVL, tenemos en claro que la altura del arbol siempre sera $\log n$, ya que si fuera mas que eso el árbol dejaria de estar balanceado. Con este dato, ya podemos tener unas mejores complejidades a la hora de ejecutar nuestras operaciones, aunque tenemos una nueva dificultad: tenemos que mantener el arbol balanceado.

Verificar si un nodo pertence no cambia, ya que no afectamos el contenido del arbol, solo lo vemos. El problema surje a la hora de querer agregar o eliminar un nodo.

6.4 Rotaciones

Si tenemos un arbol balanceado la primera parte de eliminar o insertar es exactamente identica a lo que vimos antes. El verdadero asunto arranca cuando tenemos que chequear que el balanceo se siga cumpliendo. Por ejemplo, supongamos que le queremos agregar el numero 2 al arbol de la izquierda, generando el de la derecha.



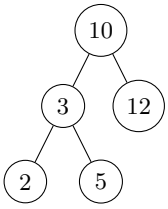
El primero es facil notar que se encuentra balanceado, por mas que no esten pintados los factores de balanceo. Pero lo importante aca es ver que, al agregar el elemento nuevo, el balanceo se pierde, y por varias ramas.

La rama izquierda del 10 es 2 nodos mas larga que la derecha. Si lo piensan, al 5 le sucede lo mismo.

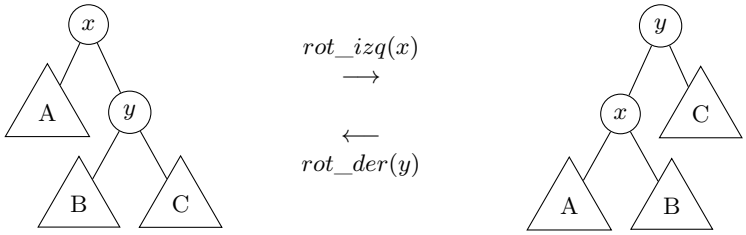
Y lo unico que hicimos fue agregar un unico elemento. Imaginense, si ya era complicado eliminar un nodo antes, como es ahora. Una manera de resolver esto podria ser estructurar al arbol de la siguiente manera:

El arbol sigue siendo el mismo, tiene los mismos nodos, simplemente los distribuimos de otra manera. Este "arreglo" que le hicimos al arbol, se lo llama *rotacion*, y es lo que vamos a ver en este capitulo.

Como se imaginaran, existen muchas rotaciones, y hay que saber cuales usar dependiendo la estructura del arbol que tengamos que rebalancear.



La idea de una rotación es reestructurar las ramas y los nodos, pero no perder el orden de quien es mayor que quien. Supongamos los siguientes arboles genericos, donde los nodos con letras simbolizan otros subarboles, y x e y valores cualesquiera:



Notemos como siempre se verifica que $A < x < B < y < C$, por lo que ambas estructuras representan exactamente al mismo arbol.

Como se ve en la imagen, dado un nodo cualquiera (en la foto, el nodo x), rotarlo hacia la izquierda seria tomar su hijo derecho, y hacerlo su padre izquierdo. Luego, reacomodar los subarboles de modo que se siga cumpliendo el orden de las desigualdades. Una rotacion a derecha, en este caso del nodo y, consiste en la operacion inversa: tomar a su hijo izquierdo y convertirlo en su padre, de modo que su hijo derecho sea el nodo inicial. Luego reacomodar los subarboles.

Nota del autor: Los nombres de izquierda y derecha tienen una logica, pero a mucha gente le hara sentido si tuvieran los nombres invertidos. Como tal, los nombres son arbitrarios, seria un debate filosofico si es mejor una notacion o la otra, el punto es que se entienda el concepto de rotacion.

Volviendo al asunto relevante, la idea seria insertar el elemento nuevo, y verificar en la rama del nuevo elemento (ya que las demas no se vieron afectadas) si se rompio o no el balanceo del arbol. Si no se rompio, es nuestro dia de suerte, no hay que hacer nada. Si, por otro lado, encontramos al nodo donde el balanceo ya no es valido, necesitamos aplicar alguna rotacion o combinacion de rotaciones para que el arbol sea el mismo pero cambiando el balanceo.

Para que analicen ustedes mismos, armamos la tabla de complejidades de distintas operaciones y estructuras.

	Pertenece	Insertar	Borrar
ABB	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$

Se nota la mejora.

7 Estructuras mas complejas

7.1 Heaps y colas de prioridad

Aqui no vimos lo que son las *colas* y las *pilas*. Basicamente son secuencias en las que, a la hora de sacar elementos, los sacamos siguiendo reglas especificas. No podemos preguntar por un indice o elemento particular, sino que obtendremos, en el caso de las colas, el primero en haber ingresado, y en el caso de las pilas, el último. Como la cola de un supermercado, el primero que llega, es el primero que sale. Y lo mismo al apilar platos, el primero de todos, va a ser el ultimo que saques.

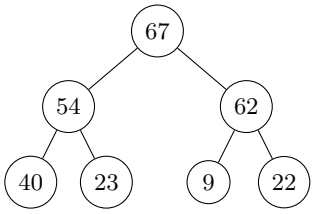
Ahora bien, estas son estructuras muy utiles, pero no las vamos a ver a fondo. Lo que nos interesa es un caso particular de las *colas*, que son las *colas de prioridad*. En lugar de devolvernos el elemento que se ingreso al principio, nos va a devolver el mayor de los elementos. O el menor, segun como lo configuremos. El punto es el mismo: independientemente del orden de ingreso de los elementos, las colas de prioridad siempre nos devuelven los elementos en orden ascendente (o al revez).

Estas estructuras son muy utiles, pero es necesario tener acceso al maximo (o minimo) de una secuencia. Claramente podriamos hacerlo "a mano", teniendo una secuencia y siempre buscando el maximo, pero la complejidad nos quedaria siempre en $O(n)$.

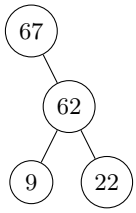
Como seguro se estan imaginando, existe una estructura para bajar esa complejidad, y aunque parezca sorprendente, la reduce hasta $O(1)$, y esa estructura, es el *heap*.

Un *heap* se representa como un arbol binario, pero no es de busqueda. En este caso, las reglas para deisgnar un *heap* son: 1) ser perfectamente balanceados; y 2) la clave de cada nodo debe ser mayor *o igual* a la de sus hijos, si los tiene. Noten que esto deja de ser de busqueda porque, primero, permitimos igualdad, y segundo, no tenemos distincion entre hijo izquierdo y derecho, solo nos interesa que los hijos sean menores que el padre, pero no cual de los dos es mayor al otro.

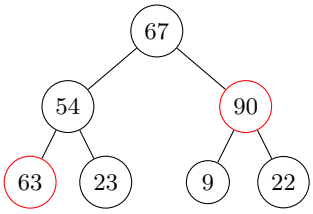
Con eso en cuenta, es facil ver que:



Esto es un heap



Esto NO es un heap



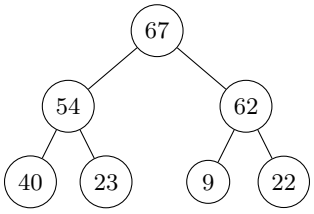
Esto NO es un heap

Nota aparte: a estos heaps en particular los llamamos *max-heaps*, ya que tienen en la raiz al nodo mayor. Tambien existen los *min-heaps*, que funcionan exactamente igual salvo porque la raiz es el nodo menor, y la regla pasa a ser que cada padre sea menor o igual a sus hijos.

Las operaciones que queremos obtener con esto son: maximo (ver el maximo sin sacarlo), desencolar (obtener el maximo, sacandolo de la lista) y encolar (insertar un elemento).

Empecemos por lo facil, la de *maximo*. Claramente tenemos un acceso directo al maximo del heap, sea por un puntero o como sea la forma que lo hayamos implementado, pero lo debemos tener a mano siempre. Claramente esto termina siendo $O(1)$.

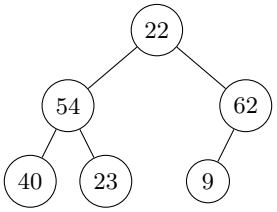
Ahora viene lo picante: *desencolar*. Supongamos el heap que vimos antes:



Si desencolamos, lo primero que haremos sera retirar el 67, pero no podemos simplemente eliminarlo, asi que lo que haremos sera reemplazarlo con el nodo que este "de ultimo". En este caso, el ultimo es el 22, porque este arbol es *izquierdista*. Llamamos *izquierdista* a los arboles que vamos llenando de izquierda a derecha, y de

igual manera, existen los *derechistas*, que son iguales pero con la logica inversa. Por favor, se le pide al lector eliminar toda carga politica de estos nombres.

Continuando, en nuestro caso el ultimo nodo es el 22, por lo que lo eliminamos (al no tener hijos, simplemente lo borramos), y lo ponemos de raiz, quedandonos:

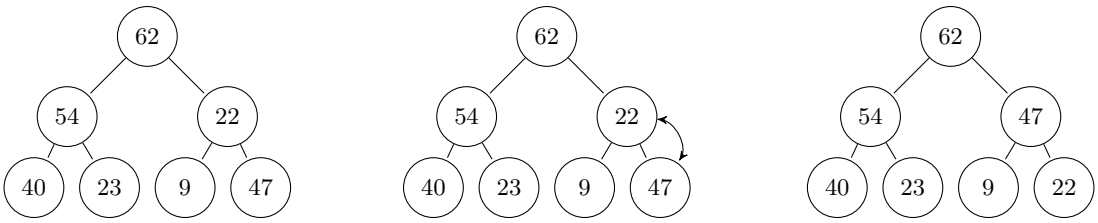


Bien, seguimos teniendo un arbol balanceado, pero ahora tenemos un problema: tenemos que reacomodar el arbol para que se cumpla la regla de los heaps. Lo bueno es que este algoritmo es bastante sencillo.

Primero vemos los dos hijos del 22, en este caso son 54 y 62, y nos preguntamos cual es mas grande de los dos. En particular, es el 62. Ahora verificamos si es mas grande que el 22, lo que es afirmativo, y los cambiamos de lugar. Luego repetimos el chequeo con los nuevos hijos del 22, asi hasta que tenga hijos que sean menores a el, o no los tenga. Graficamente, podemos ver que:

En este caso, el 22 quedaria ahi, ya que no es mas chico que el 9, y no tiene otro hijo con el que comparar. Por lo tanto, el heap nuevo tras haber eliminado el maximo quedaria asi. En esta operacion a lo mucho hacemos $\log n$ operaciones, por lo que concluimos con una complejidad de $O(\log n)$

Finalmente, veamos el caso de *insertar*. Para este caso, supongamos que ahora, al arbol al que recien le eliminamos un nodo, le queremos meter el numero 47. Analogo a lo anterior, lo primero que hacemos es agregarlo al fondo, y luego lo comparamos con el padre para "corregir" su posicion. En nuestro ejemplo:



Si quisieramos insertar un elemento nuevo a un arbol que ya se encuentra lleno, simplemente añadimos un nodo como hijo izquierdo del nodo sin hijos de mas a la izquierda.

Notemos que, en el peor de los casos, estas correcciones de posicion, las haremos hasta la raiz, si es que justo ingresamos un nuevo elemento mas grande que todos. En cuyo caso, hariamos una cantidad de correcciones igual a la altura del arbol, que es $\log n$. Entonces, tenemos que insertar nos ocupa $O(\log n)$

Comparando las complejidades con nuestras anteriores estructuras obtenemos que:

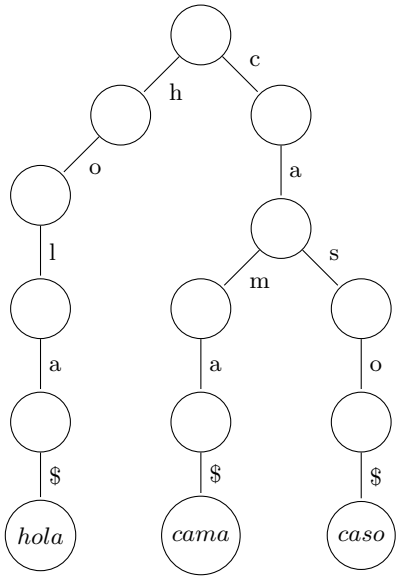
	Pertenece	Insertar	Borrar	Buscar minimo	Borrar minimo
Lista Ordenada	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$	$O(1)$
ABB	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$	$O(\log n)$

7.2 Tries

Todo bien con estas estructuras medio raras y demas, pero tenemos imaginemos que queremos representar una lista de palabras. O una lista de numeros tan largos que, antes que ver el valor del numero, conviene ver las cifras del mismo. El concepto se va a entender mejor con el ejemplo de las palabras, asi que nos vamos a quedar con eso, pero se podria aplicar la misma idea a numeros y chequeos de cifra a cifra.

Supongamos que tenemos la secuencia $[hola, casa, caso]$. Podriamos represntarla con alguna de las estructuras anteriores, si, lo cual seria mas o menos dificil, pero, primero, tendriamos que inventarnos algun tipo de ordenamiento para palabras, y, segundo, seguiriamos con las mismas complejidades de antes. Pero existe una estructura que, para estos casos, la complejidad de busqueda es lineal, es decir $O(1)$. Bueno, en realidad, bajo ciertos supuestos, pero lo vamos a ir viendo despacio. Primero presentemos la idea de los *tries*.

Continando con nuestro ejemplo, representar esa secuencia en un trie consistiria en escribir:



Si se fijan, al ordenar la lista de esta forma para buscar una palabra en la lista dependemos exclusivamente del largo de la palabra, no del la cantidad de elementos en la misma. Esta es la magia de los *tries*, que no tiene $O(n)$, sino $O(x)$, donde x es el largo de la palabra a chequear. La misma logica sirve para insertar y demas.

Pero bueno, suficiente del ejemplo, vamos a lo concreto. Un *trie* consiste en un arbol, no necesariamente binario, en el que la informacion se encuentra en las "flechas", y no en los nodos. A su vez, tenemos un diccionario o alfabeto donde se encuentran todos nuestros posibles simbolos o caracteres (en nuestro ejemplo, consisitia de todas las letras del abecedario), y se incluye un caracter especial para indicar que la palabra o secuencia de simbolos finalizo (en nuestro ejemplo, es el \$).

La unica regla a tener en cuenta, evidentemente es que el simbolo no pueda formar parte de las claves. Es decir, no podriamos incluir a la lista la palabra *cas*\$, porque el ultimo simbolo se perderia.

Ahora, quizas esten pensando *vos dijiste que ibamos a tener un algoritmo lineal, esto no depende de la entrada, pero lineal no es, Tomi, nos mentiste*, pero ¡no menti!. La suposicion que podemos hacer aca, es que en nuestro alfabeto español no existen palabras de mas de cierta cantidad de letras. No lo voy a chequear, pero supongamos que la palabra mas larga es de 25 letras, entonces sabemos que como maximo tenemos una complejidad de $O(25)$, que como sabemos es identico a $O(1)$. Por lo que, efectivamente, un *trie* tiene complejidad lineal.

Obviando el tema de la complejidad, seria valido preguntarnos que podemos describir o representar con esto de los *tries*, ya que si solo pudieramos listas de palabras o numeros largos tampoco le sacariamos tanto provecho. Pero si recuerdan, hay una estructura a la que le viene como anillo al dedo este concepto: los *diccionarios*.

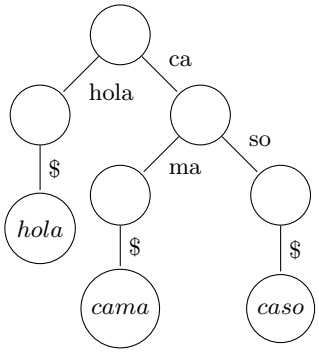
Piensen que, recorremos el arbol siguiendo una palabra concreta (por ejemplo, *cama*). Primero pasariamos por la raiz, y buscamos la rama con la primera letra, es decir, la *c*. Luego vamos al siguiente nodo, y repetimos la operacion. Asi hasta llegar al simbolo reservado. En nuestro ejemplo, cuando llegamos al final, en el ultimo nodo tenemos la palabra escrita (*casa*), pero podriamos haber tenido otra cosa guardada ahi. Si se dan cuenta, este es exactamente la funcionalidad de los diccionarios: tenemos claves (las palabras con las que nos movemos en el trie) y los significados (las hojas que le siguen al simbolo reservado). Si representamos asi a los diccionarios, tendriamos una complejidad de busqueda increiblemente baja, aparte de que podriamos guardar la cantidad de claves que queramos y conservar la complejidad.

Para continuar con el ejemplo, si tenemos el diccionario $\{hola : 1, cama : 2, caso : 4\}$, simplemente reemplazariamos en los nodos hoja los valores numericos correspondientes.

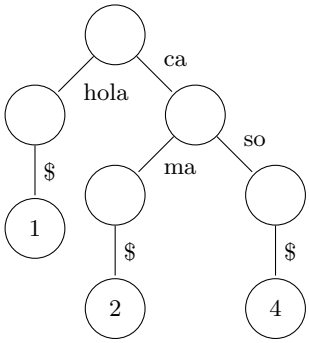
Aunque parezca sorprendente, esta estructura se puede optimizar todavia mas. Fijense que, en muchas

ocasiones hay *redundancias*. Es decir, hay veces donde podriamos compactar varios nodos en una unica rama. Por ejemplo, el caso de *cama* y *caso*. Veamos que ambos comparten la primera parte de la palabra (*ca*), y no hay mas palabras, por ahora, que se bifurquen de ahi. Podriamos crear una unica rama con la clave *ca*, y ahorrarnos un nodo de busqueda.

Si repetimos este procedimiento para todos los nodos, podriamos formar el siguiente trie:



Trie de antes modificado



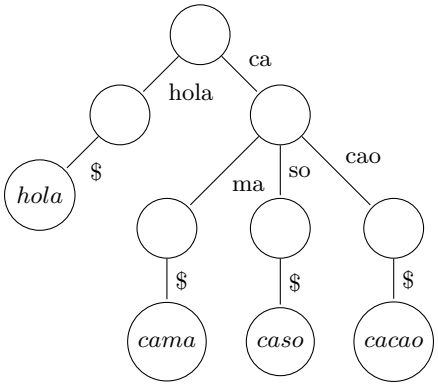
Uso de trie para un diccionario

Que claramente es mucho mas chico que el anterior, lo que significa menos tiempo de busqueda y menos espacio de memoria utilizado.

Ahora bien, veamos por arriba los algoritmos clasicos. El de busqueda ya lo vimos, basicamente vamos iterando en las letras que coincidan hasta que encontremos el simbolo reservado, y si en algun momento no podemos seguir avanzando o no encontramos el simbolo, significa que la palabra no se encuentra en el trie.

Por otro lado, la insercion consiste en, de igual manera, iterar por las letras que coincidan, pero cuando no podamos continuar (porque nos faltan letras o porque tenemos que "separar" una rama con mas de una letra), agregamos esa rama nueva y creamos nuestros propios nodos.

Si quisieramos agregar a nuestro trie de antes la palabra *cacao*, quedaria asi:



8 Sorting

8.1 Tecnicas de ordenamiento

Como vimos antes, lo que siempre se busca en general son algoritmos rapidos o que consuman poco espacio. Ese suele ser un problema recurrente en la computación. Una de las cosas que mas se hace al programar es utilizar listas, o arreglos, y buscar elementos. Existen muchas formas de representarlos y aprovecharnos de eso, como ya mostramos, pero hay un metodo que supera todas las complejidades de busqueda de antes: la *busqueda binaria*. Ya lo explicamos al final del capitulo 6, justo antes de saltar a los arboles.

A lo que queremos llegar es que solo sabiendo que la lista esta ordenada, ya podemos reducir el tiempo de busqueda mucho. Por lo tanto, nos interesa ser capaces de ordenar listas, ya que no siempre vamos a poder darnos el lujo de tener un AVL o un heap, aparte de que no son precisamente faciles de diseñar.

Por eso mismo, esta ultima parte va a consistir en ver y comparar distintos metodos de ordenar listas. Vamos a dar por hecho que son listas de numeros, pero se puede aplicar la misma logica a cualquier tipo de dato comparable.

8.2 Selection sort

El primer algoritmo que vamos a ver es uno de los mas intuitivos y el que, probablemente, la gente utilice en la vida cotidiana a la hora de ordenar cosas en la vida real.

Consiste en buscar el minimo y ponerlo primero. Luego, buscamos el minimo de lo que nos queda de la lista, es decir, todo salvo la primera posicion. Ese minimo seria el segundo elemento, y lo ponemos donde va, "arrastrando" todo lo demas una posicion. Repetimos este proceso, hasta llegar al final de la lista.

La verdad que no tiene mucha mas profundidad que esa. Dano que buscar el minimo en una lista tiene costo $O(n)$, y que estamos buscando el minimo de una lista cada vez mas pequeña, el costo en tiempo de este algoritmo lo podemos escribir como:

$$\sum_{i=0}^{n-2}(n-i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Si hacemos la cuenta, podemos ver que terminamos teniendo una complejidad de $O(n^2)$, es decir, de orden cuadratico.

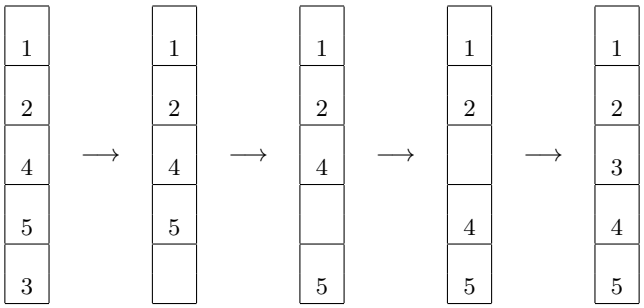
Lo que hace particularmente lento a este algoritmo es el hecho de que, por mas que este ordendo completa o parcialmente, va a continuar haciendo todo los chequeos de todas formas. Eso quiere decir, buscar el minimo de cada subsecuencia, por mas que ya este ordenado, o que tras haber hecho unos pocos cambios, lo que quede de la lista ya se encuentre ordenada. Podriamos modificar el algoritmo para que haga estos chequeos y ahorrarnos un poco de tiempo, pero en casos de verdadero desorden, eso simplemente nos haria la tarea mas lenta.

8.3 Insertion sort

Este algoritmo es *un poquito* mejor que el anterior, pero no mucho.

Consiste en iterar sobre la lista y, comparar ese i-esimo elemento con todos los elementos anteriores. Lo que haremos con estas comparaciones es ponerlo "donde va" entre los elementos que ya ordenados. Asi es, lo que procuramos con este algoritmo es que, lo que ya revisamos, esta ordenado. No quiere decir que los elementos esten en sus posiciones finales, significa que se encuentran *relativamente* ordenados.

Por ejemplo, si tuvieramos la lista $[1, 2, 4, 5, 3]$, lo que deberiamos hacer, al llegar a la posicion del 3 es:



Si nos ponemos a hacer las cuentas, que son semejantes a las del *selection sort*, llegariamos al siguiente resultado:

$$\sum_{i=1}^{n-1}(i-1) = \frac{(n-1)(n-2)}{2}$$

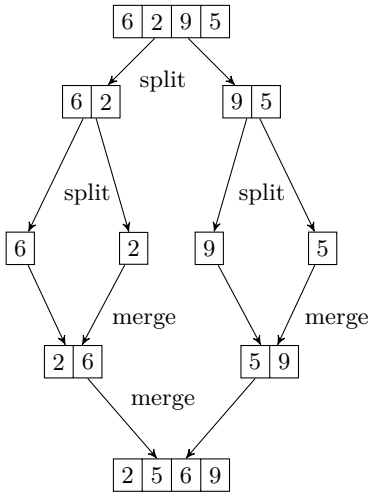
De lo que concluiriamos lo mismo: este algoritmo tambien es de orden cuadratico, es decir, tiene complejidad $O(n^2)$. Pero si ignoramos el orden y vemos la cuenta especifica, este algoritmo es *un poquito* mejor.

8.4 Merge sort

La verdad que estos algoritmos no fueron muy alentadores, pero es posible bajar la complejidad. Eso si, el algoritmo no es nada lindo, o mejor dicho, es super eficiente, pero dificil de implementar.

La idea basica radica en el concepto de *divide and conquer* (o *divide y conquistaras*), que consiste en separar un problema grande en varios problemas pero mas chicos o faciles de resolver. Llevado al mundo del ordenamiento, el plan es el siguiente: dividi la lista a la mitad, y ordena cada mitad por separado. Una vez las tengas ordenadas, combinalas (*merge*, en ingles) ordenadamente. Lo bueno de este algoritmo, es que podemos plantearlo recursivamente, teniendo en cuenta que toda lista la podemos dividir a la mitad. Bueno, todas, salvo las de un unico elemento, pero esas ya estan ordenadas. Muy conveniente para un caso base.

En el dibujo de la derecha se puede ver graficamente una implementacion del *merge sort* con la lista $[6, 2, 9, 5]$, donde se ordena y finaliza siendo $[2, 5, 6, 9]$.



La cuenta es muy larga y fea para hacerla aca, pero, basandonos en la idea recursiva, el tiempo que tarda el programa se puede escribir como:

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ 2T(\frac{n}{2}) + O(n-1) & \text{si } n > 1 \end{cases}$$

Y si hacemos un exhaustivo analisis de calculo sobre la formula y el comportamiento del algoritmo, llegamos a que la complejidad del mismo es de $O(n \log(n))$.

Una nota aparte, pero relevante para el caso, es que si se dan cuenta esto de partir siempre la lista a la mitad esta muy bueno y se puede, siempre y cuando la lista tenga una cantidad par de elementos. El mejor caso seria que nuestra lista tenga $n = 2^k$ elementos, pero normalmente no podemos dar por sentado eso. Cuando tenemos una cantidad de elementos que no cumple esto, lo mejor seria separar la secuencia "a mano" de modo que se acerque a la potencia de 2 mas cercana.

Por ejemplo, si tuvieramos una lista con 12 elementos, seria mucho mejor dejar los ultimos 4 por un lado, y quedarnos con una lista de 8 elementos por el otro. Y si fuera impar, por ejemplo, 13 elementos, hariamos lo mismo y nos quedariamos con uno de 5 y otro de 8. Ahora repetimos la logica con la lista de 5, llegando a 4 que es una potencia de 2. Como ven, existe una solucion para estos casos.

8.5 Quick sort

Este sorting tiene complejidad identica a los casos primeros (a saber, $O(n^2)$), pero es uno de los mas usados aun asi. Esto tiene que ver con el hecho de que en los casos promedio la complejidad baja a $O(n \log(n))$.

Primero partimos de una premisa algo exagerada, pero sirve para el ejemplo. Supongamos que conocemos el elemento mediano del arreglo (el que separa en dos mitades iguales, o casi). En ese caso, semejante a lo explicado con el *merge sort*, podemos separar la lista en dos: por un lado los elementos que son menores a esa mediana, y por el otro los que son mayores. Una vez hecho esto, podemos repetir el proceso en cada mitad,

buscando su mediana, y separando esas listas en dos mitades, a las cuales ordenamos, y asi sucesivamente. Finalmente, unimos todo y terminamos con una lista completamente ordenada.

Como podran imaginar, este algoritmo es buenisimo siempre que sepamos quien es la mediana. Ahora, supongamos que no conocemos la mediana (que es lo que va a pasar casi siempre). Entonces, vamos a tener que elegir a algun elemento como punto medio para partir la lista. Aca tenemos un problema grave, ya que la eleccion de este pivot es clave.

Supongamos que elegimos el elemento mas chico como pivot. Terminariamos por poner todos los elementos a su derecha, ya que al ser el minimo todos van a ser mayores a el. Esto no ordena nada, de hecho, deja todo como estaba.

Por eso, aunque parezca poco logico, lo mejor para elegir el pivote, es elegirlo al azar. La probabilidad de elegir justo el elemento minimo es $\frac{1}{n}$, identica a la probabilidad de elegir al mayor. Por otro lado, la probabilidad de elegir cualquier elemento que no sea ni el maximo ni el minimo es de $\frac{(n-2)}{n}$, que es bastante.

Es por esto que se dice que, en el caso promedio, y bajo una buena eleccion de pivote, nuestros tiempos de ordenamiento no suben radicalmente.

8.6 Menciones honorificas

Vamos a mencionar dos metodos mas de ordenamiento que, quizas no se usen tan seguido, pero son interesantes y sirven para ver la variedad de algoritmos que existen. Puntualmente, vamos analizar el *bubble sort* y el *heap sort*.

Vamos con el *bubble sort* primero. Para este algoritmo lo que haremos sera recorrer la lista e ir comparando los elementos "de a dos". El primero con el segundo, luego el segundo con el tercero, y asi sucesivamente. La idea seria, cuando encontramos a alguien mas grande que el elemento que tiene a su derecha, los intercambiamos de lugar. Es decir, si un elemento en la posicion i es mas grande que el que se encuentra en la posicion $i+1$ los intercambiamos entre si.

Tras la primer corrida del algoritmo, el elemento mas grande terminaria en la ultima posicion, ya que efectivamente va a ser mas grandes que todos los que tiene despues. La siguiente parte del algoritmo es repetir lo mismo, una y otra vez, una cantidad de veces igual a la cantidad de elementos de la lista. Por este mismo paso, la "doble iteracion", por decirle de algun modo, nos genera una complejidad de $O(n^2)$.

Este algoritmo no es muy bueno, pero es tierno. Al menos lo intenta.

Por otro lado, tenemos al *heap sort*. Partamos de la base de que podemos representar un heap, o bien *heapificamos* un arreglo, o bien el contexto del programa ya nos lo venia pidiendo, o lo que sea. Tenemos nuestra lista de elementos ordenada o con la estructura interna de un heap. Bien, esa es la parte dificil.

Si tenemos un heap, ordenar la lista es bastante sencillo, ya que un heap siempre nos puede devolver el maximo (o el minimo) con una complejidad bastante baja. Eso quiere decir que podemos ir desencolando el maximo y meterlo en el final de otro arreglo (o el mismo heap si procuramos que no se desarme). Vamos "de atras hacia adelante", pidiendo los mas grandes y tirandolos al fondo. Semejante al *selection sort*, pero aca la idea radica en implementar un heap, por su facilidad de encontrar al maximo.

Las complejidades y propiedades de estos algoritmos se ven al final del capitulo, comparadas con los demas.

8.7 Estabilidad del ordenamiento

Existe un concepto que no mencionamos hasta ahora que es el de *estabilidad*.

Cuando nosotros ordenamos elementos en una lista, a veces nos interesa que estos elementos mantengan su *orden relativo original*. Quizas tenemos dos o mas criterios de ordenamiento, y queremos reordenar la lista sin perder el orden previamente establecido.

Un ejemplo comun de esto, pueden ser datos varios en cualquier tabla como las de excell. Supongamos que el profesor quiere ordenar a sus alumnos por nota, porque quiere establecer un nuevo criterio de promocion o lo que sea. Ahora, va a haber muchos empates en ese criterio, puede haber muchos que se sacaron un 10, otros tantos un 9, y asi sucesivamente. En este ejemplo solo hay 10 numeros para elegir, y quizas tenemos mas de 200 alumnos.

Ahora bien, resulta que la lista de alumnos, previo a ser ordenada por nota, esta ordenada alfabeticamente. Primero esta Alvarez, luego Benitez, y asi. El profesor no quiere perder ese orden al ponerse a ordenar en base a la nota, por lo que quiere que la lista quede ordenada por nota, pero en caso de empate, mantener el orden alfabetico.

Bueno, esto lo podria hacer a mano, lo cual es un delirio personalmente, pero si, es posible. Pero tambien, podria usar algunos de los algoritmos que vimos antes. Veran, como dijimos antes, hay algoritmos que son *estables*, es decir, que no van a romper ese criterio previamente establecido.

Un "algoritmo" que sirve para ordenar listas con muchos criterios de orden es el llamado *radix*, que consiste en ordenar primero por el criterio menos relevante con cualquier algoritmo, y luego el mas relevante con un algoritmo estable, para no perder el orden anterior. Digo *algoritmo* entre comillas porque no es un metodo de ordenamiento como tal, sino una forma de mantener los distintos criterios de ordenamiento.

De los que vimos, en particular, los algoritmos estables son *insertion*, *merge* y *bubble*, y los demas son no estables.

Todas las complejidades de estos algoritmos, para ir haciendo un resumen de lo visto, se pueden ver comparadas en la siguiente tabla:

Nombre	Mejor caso	Caso promedio	Caso peor	Es estable
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	No
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Si
Merge	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	Si
Quick	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	No
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	Si
Heap	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	No