

colección
amalgama



MATERIAL
de cátedra

Algoritmos y estructuras de datos en Python

Un enfoque ágil y estructurado

Walter Bel

]

EDITORIAL UADER
FACULTAD DE
CIENCIA Y TECNOLOGÍA



Libro
Universitario
Argentino



UADER

colección
amalgama

Algoritmos y estructuras de datos en Python

Un enfoque ágil y estructurado

Facultad de Ciencia y Tecnología

EDITORIAL  UADER

UNIVERSIDAD AUTÓNOMA DE ENTRE RÍOS

Abog. Luciano Filipuzzi | RECTOR

Ing. Esp. Rossana Sosa Zitto | VICERRECTORA

Mg. Ma. Florencia Walz | DIRECTORA EDITORIAL UADER

Algoritmos y estructuras de datos en Python

Un enfoque ágil y estructurado

Walter Bel

Bel, Walter

Algoritmos y estructuras de datos en Python: un enfoque ágil y estructurado / Walter Bel. - 1a ed. - Paraná: Editorial Uader, 2020.

Libro digital, PDF - (Amalgama)

Archivo Digital: descarga y online

ISBN 978-950-9581-60-9

1. Algoritmo. 2. Lenguajes de Programación. 3. Ingeniería Informática. I. Título.

CDD 005.131

©EDITORIAL UADER

Diseño Gráfico: Alfredo Molina

Edición y corrección: Sebastián Galizzi



Razón social: UADER/Editorial UADER

Avda. Ramírez 1143, E3100FGA

Paraná, Entre Ríos, Argentina

editorial@uader.edu.ar

www.uader.edu.ar

Hecho el depósito que marca la ley 11.723

Prohibida su reproducción total o parcial

Derechos reservados

ISBN 978-950-9581-60-9

A standard linear barcode representing the ISBN number 978-950-9581-60-9. The barcode is composed of vertical black bars of varying widths on a white background. Below the barcode, the numbers 9 789509 581609 are printed vertically.

Índice

Introducción	9
CAPÍTULO I Una breve introducción a algoritmos, estructuras de datos y Python	11
CAPÍTULO II Auto llamadas con algoritmos recursivos	19
CAPÍTULO III Analizando algoritmos en búsqueda de eficiencia	27
CAPÍTULO IV Buscar es más fácil cuando las cosas están ordenadas	43
CAPÍTULO V Representando modelos reales con tipos de datos abstractos	67
CAPÍTULO VI Desde la cima de la pila, intentando evitar el desbordamiento	77
CAPÍTULO VII Por favor espere su turno en la cola que será atendido	87
CAPÍTULO VIII Como vagones de un tren enlazamos elementos para construir una lista enlazada	101
CAPÍTULO IX Utilizando mapas para acceder rápidamente a los datos con tablas <i>hash</i>	119
CAPÍTULO X Desde la raíz hasta las hojas, entendiendo los árboles desde otro punto de vista	135
CAPÍTULO XI Pintando nodos de color rojo y negro, ¡conozcamos los árboles rojinegros!	173
CAPÍTULO XII Simplifiquemos las cosas usando montones, ¡montículos para todo!	187
CAPÍTULO XIII Conectando puntos. Cómo formar una red conocida como grafo	199
CAPÍTULO XIV Técnicas de diseño de algoritmos, intentando desarrollar algoritmos eficientes	237

Introducción

El contenido de este libro se enfoca en dos ejes temáticos troncales del área programación fuertemente relacionados, *algoritmos* y *estructuras de datos*, está orientado a lectores que tengan conocimientos básicos de programación –que se darán por sentado en este libro– y destinado principalmente para estudiantes de carreras informáticas. Ambos son partes de los tres pilares fundamentales del desarrollo de sistemas: diseño, programación –o desarrollo– y almacenamiento. Dentro del segundo de estos pilares es donde tienen mayor relevancia los algoritmos y las estructuras de datos.

Es un punto crítico elegir la estructura de datos apropiada que almacenará la información en memoria principal de acuerdo al dominio del problema a tratar, para que los algoritmos de *búsqueda* y *ordenamiento* que se utilicen sean eficientes, sobre todo cuando el volumen de datos es grande y es necesario que el sistema responda en buen tiempo. Respecto al tercero de los pilares, –el almacenamiento–, los distintos motores de bases de datos utilizan diversas estructuras de datos para su administración interna, como árboles para hacer consultas e índices *hash* que permiten optimizar las consultas, el acceso a memoria secundaria y también a los datos, en el caso de las bases de datos claves-valor.

Es fundamental conocer estos aspectos para poder determinar con buen criterio la elección de una base de datos para el desarrollo de un sistema. O en el caso que la persistencia sea en un archivo, se deben conocer los algoritmos y estructuras de datos que permiten administrar, buscar y ordenar estos datos en un tiempo aceptable de respuesta, siempre esperando que sea el menor posible, ya que en estos casos es crucial el tiempo de acceso a la memoria secundaria –es decir el acceso al disco–.

Por su parte el pilar de diseño o *ingeniería de software* es esencial para que el desarrollo del sistema sea exitoso. Para esto, además de los aspectos mencionados, se requiere una visión global del sistema, como también de sus requerimientos, arquitectura y tecnologías que se utilizarán para el desarrollo del mismo al momento de modelarlo. **Las estructuras de datos son fundamentales para establecer el nexo que permite intercambiar información entre las capas de *frontend* –o presentación– y la de *backend* –o persistencia–.**

Respecto a los algoritmos, existen diversos factores que condicionan su programación: eficiencia (referido a las técnicas, tiempos y recursos utilizados), forma de estructurarlo o escribirlo (referido a la tabulación, organización y legibilidad del código), estilo de escritura o nomenclatura utilizada, uso de buenas prácticas y documentación dentro del código. A excepción del primero, el resto se puede controlar con la ayuda de un Entorno de Desarrollo Integrado (IDE) y un poco de experiencia como desarrollador. En relación a la eficiencia de los algoritmos, es necesario comprender cómo poder analizar y medir dicha eficiencia para poder comparar con criterio algoritmos similares que utilicen distintas técnicas, es decir, que sirvan para resolver un mismo problema. Estos contenidos se abordarán en detalle en los capítulos II, III, IV y XIV.

En cuanto a las estructuras de datos cabe señalar que son recursos o herramientas disponibles en todo lenguaje de programación. Permiten almacenar o agrupar un conjunto de elementos o datos. Muchas de estas estructuras de datos son utilizadas internamente por los distintos tipos de bases de datos para gestionar la información (que es la manera más utilizada para almacenar información desde la década de 1980) y por los sistemas operativos para gestionar los distintos recursos de una computadora. Es importante entender que no existe una estructura de datos mejor que las demás,

cada una tiene sus ventajas y desventajas. Por este motivo, es imprescindible que el programador comprenda cómo funciona cada una y cuándo es conveniente utilizarlas para sacarles mayor provecho. En ocasiones, suele ocurrir que es necesario trabajar con más de una estructura de datos a la vez, combinándolas en un mismo algoritmo. Además nos sirven como nexo entre los datos dentro de nuestro algoritmo (en memoria principal) y los datos almacenados físicamente en disco (ya sea en una base de datos o en un archivo). Desde el capítulo V al XIII se presentarán y analizarán en detalle las distintas estructuras de datos y cómo se modelan.

En resumen, el énfasis está puesto en comprender cómo funcionan los algoritmos y estructuras de datos, con el objetivo de poder implementar todos los conceptos planteados en cada capítulo. De esta manera, el lector podrá llevar a la práctica los conceptos teóricos y fortalecer sus habilidades de programador, adquirir experiencia para decidir con criterio cómo y cuándo utilizar los distintos tipos de algoritmos y estructuras de datos. Para ello, al final de cada capítulo se facilitará una guía de ejercicios para comenzar a incorporar estos recursos y desarrollar el pensamiento algorítmico.

CAPÍTULO I

Una breve introducción a algoritmos, estructuras de datos y Python

Para comenzar a abordar los diferentes temas referidos a algoritmos y estructuras de datos y aprovechar las potencialidades que el lenguaje de programación Python presenta para su implementación es imprescindible establecer las bases conceptuales mínimas, necesarias para comprender el resto de los capítulos de este libro.

Algoritmo ¿Qué cosa es?

El nombre algoritmo (del latín *algorithmus*) proviene del matemático persa del siglo IX Abu Abdallah Muḥammad ibn Mūsā al-Khowârizmî, conocido como al-Juarismi. Podemos encontrar muchas definiciones en distintas fuentes de bibliografías dependiendo del enfoque de cada autor:

Informalmente, un algoritmo es cualquier procedimiento computacional bien definido que toma algún valor, o conjunto de valores, como entrada y produce algún valor, o conjunto de valores, como salida. (Cormen at. al., 2009: 5)

Un algoritmo es una secuencia de pasos computacionales que transforma la entrada en la salida. (Cormen at. al., 2009: 5)

Podemos ver un algoritmo como una herramienta para resolver un problema computacional bien especificado. La declaración del problema especifica en términos generales la relación de entrada/salida deseada. El algoritmo describe un procedimiento computacional específico para lograr esa relación de entrada/salida. (Cormen at. al., 2009: 5)

Un conjunto de reglas para efectuar algún cálculo, bien sea a mano o en una máquina. (Brassard y Bratley, 1997: 2)

Un conjunto de instrucciones sencillas, claramente especificado, que se debe seguir para resolver un problema. (Weiss, 1995: 17)

Según el Diccionario de la lengua española (DLE) es un “Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema”; o un “Método y notación en las distintas formas del cálculo”.

Una secuencia de instrucciones que representan un modelo de solución para determinado tipo de problemas. (Joyanes, 2003: 4)

Un conjunto o grupo de instrucciones que realizadas en orden conducen a obtener la solución de un problema. (Joyanes, 1990: 7)

De forma general, se puede definir un algoritmo como una secuencia de instrucciones o pasos que al ejecutarlos recibe datos de entrada, de manera directa o indirecta y los transforma en una salida. También se lo puede definir con un conjunto de instrucciones ordenadas de manera lógica tal que al ser ejecutadas por una computadora o persona dan solución a un determinado problema para el cual fue diseñado.

Los algoritmos son independientes de los distintos lenguajes de programación en los que se desee implementarlos. Para cada problema se puede escribir un algoritmo, que luego puede codificarse y ejecutarse en diferentes lenguajes de programación. El algoritmo es la infraestructura de la solución a un problema, que luego puede ser implementada en cualquier lenguaje de programación.

Es importante destacar que los algoritmos deben cumplir determinadas características para ser considerados como tales. En primer lugar un algoritmo debe ser preciso, es decir debe definirse de manera rigurosa, sin dar lugar a ambigüedades en las instrucciones que lo forman. Este también debe ser definido, esto implica que si se ejecuta dos veces el mismo algoritmo con la misma entrada –salvo sean algoritmos aleatorios– se debe obtener el mismo resultado de salida. Además tiene que ser finito. Esto quiere decir que debe terminar de ejecutarse en algún momento. Pueden tener cero o más elementos de entrada y a su vez debe producir un resultado, los datos de salida serán los resultados de efectuar las instrucciones que lo forman sobre la entrada. Finalmente se concluye que un algoritmo debe ser suficiente para resolver el problema para el cual fue diseñado.

Frente a dos algoritmos que lleven cabo un mismo objetivo, es decir que resuelvan el mismo problema –ocurre bastante a menudo–, siempre será preferible el más corto. Esto no es una tarea trivial, dado que se debe tener en cuenta el análisis del algoritmo para la optimización de los tiempos de ejecución y recursos que consume, como se verá más adelante en profundidad y detalle en el capítulo III.

Además los algoritmos pueden clasificarse en algoritmos cualitativos y cuantitativos. En el caso de los primeros, refieren a los que en sus pasos o instrucciones que lo forman no están involucradas o intervienen cálculos numéricos. Por ejemplo, las instrucciones para desarrollar una actividad física o encontrar un tesoro, tomar mate, buscar una palabra en el diccionario, etc. Mientras que los segundos son aquellos algoritmos en los que dentro de los pasos o instrucciones involucran cálculos numéricos. Por ejemplo, solución de una ecuación de segundo grado, conteo de elementos que cumplen una determinada condición, cálculo de un promedio, etc.

¿Programa y algoritmo es lo mismo? Es importante destacar que no es lo mismo un programa que un algoritmo. Un programa es una serie de instrucciones ordenadas, codificadas en un lenguaje de programación que expresa uno o varios algoritmos y que puede ser ejecutado en una computadora. Por lo general un programa contempla un conjunto de instrucciones encargadas de controlar el flujo de ejecución del programa –a través de una interfaz que puede ser de consola o gráfica– y hacer las llamadas a los distintos algoritmos que lo forman.

Algoritmia, una manera de estudiar los algoritmos

A su vez, el término *algoritmia* está relacionado de manera directa con los algoritmos, o podríamos decir que deriva del mismo. Existen varias definiciones para este término:

Ciencia que nos permite evaluar el efecto de diferentes factores externos (como los números implicados, la forma en que se presenta el problema, o la velocidad y capacidad de almacenamiento de nuestro equipo) sobre los algoritmos disponibles, de tal modo que sea posible seleccionar el que mejor se ajuste a nuestras circunstancias particulares. (Brassard y Bratley, 1997: 3)

El estudio de los algoritmos. (Brassard y Bratley, 1997: 3)

Según DEL es “Ciencia del cálculo aritmético y algebraico, teoría de los números”.

Se puede definir entonces la algoritmia como la ciencia que estudia a través de determinadas técnicas y herramientas cuál es el mejor algoritmo dentro de un conjunto de algoritmos diseñados para resolver un mismo problema, de acuerdo a un determinado criterio, necesidades y características del problema.

¿Para qué necesitamos las estructuras de datos?

Las estructuras de datos son las principales herramientas o recursos que tienen los programadores para el desarrollo de sus algoritmos. Es difícil pensar en un algoritmo o conjunto de estos que no tenga como macroestructura de soporte una estructura de datos. Entre las definiciones más conocidas de estructuras de datos podemos mencionar:

Una estructura de datos es una forma de almacenar y organizar datos para facilitar el acceso y las modificaciones. Ninguna estructura de datos funciona bien para todos los propósitos, por lo que es importante conocer las fortalezas y limitaciones de cada una de ellas. (Cormen et. al., 2009: 9)

Una estructura de datos es una colección de datos (normalmente de tipo simple) que se caracterizan por su organización y las operaciones que se definen en ellos. (Quetglás et. al., 1995: 171)

En otras palabras, una estructura de datos es una colección de datos simples o compuestos –como los registros– con un sentido u orden concreto para quien lo está utilizando, sobre la cual se definen un conjunto de operaciones que por lo general describen el comportamiento de la estructura.

La estructura de datos más simple que disponemos es una *variable*. Con esta podemos realizar las siguientes operaciones: *obtener* y *modificar* su valor de manera directa utilizando su identificador. Por su parte, un *registro* define un conjunto de atributos o campos a los que podemos acceder mediante su identificador y luego utilizar el operador punto seguido del nombre del campo para acceder a su valor: “registro.altura”, “registro.peso”.

Existen numerosos tipos de estructuras de datos, generalmente construidas sobre otras más simples. Por ejemplo, vector, matriz, pila, cola, lista, árboles, montículo, grafos, archivo, etc. Estas son lineales o no lineales –también llamadas ramificadas– las mismas pueden ser estáticas o dinámicas. Por lo general, al trabajar sobre estas estructuras de datos se suelen contemplar algunas actividades mínimas como *insertar*, *eliminar* y *buscar* un dato en la misma. A lo largo de los capítulos de este libro se describen en profundidad varias de las estructuras de datos clásicas.

Pensamiento algorítmico ¿De qué se trata?

Quizás, uno de los aspectos más importantes de esta obra es su intención de crear y fortalecer el pensamiento algorítmico del lector. Por esta razón, en cada guía de ejercicios se encontrarán algunos problemas sencillos para aplicar los conceptos abordados en el capítulo y luego otros más avanzados e integradores, donde se deberá atacar los problemas con un enfoque algorítmico para poder obtener una solución. Para comenzar a introducir al lector en esta mecánica práctica se puede plantear un primer ejercicio: “resuelva el problema de mover el caballo de ajedrez sobre el teclado de un teléfono”.

Suponga que tiene a su disposición dicha ficha del ajedrez, y que se puede mover en ciertas formas particulares como se observa en la figura 1. Ahora, desea saber cuántos movimientos válidos pueden realizarse partiendo con el caballo desde todos los números del teclado realizando un movimiento desde cada número.

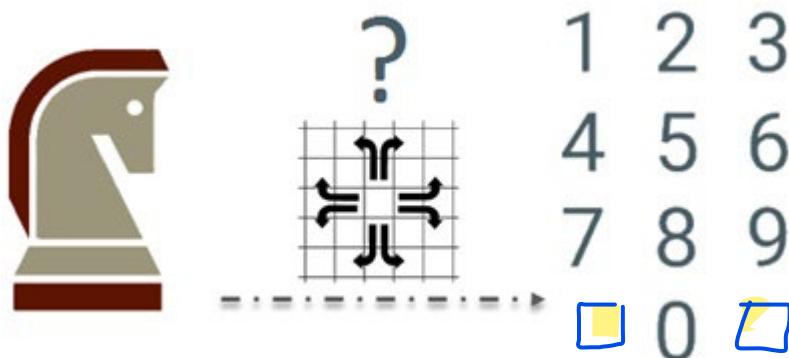


Figura 1. Problema de los movimientos de caballo de ajedrez

Si se parte del 1 se puede ir al 6 y al 8 (dos movimientos); si se sale del 2 se puede llegar al 7 y al 9 (dos movimientos más); iniciando desde el 3 se puede arribar al 4 y al 8 (se suman nuevamente dos); si se arranca desde el 4 las posibilidades son 3, 9 y 0 (ahora se acumulan tres movimientos); pero si la posición inicial es el 5 no se puede mover la ficha a ningún lugar dado que no hay movimientos válidos –sin embargo aún restan varias posibilidades más para seguir probando–; desde el 6 se pueden alcanzar el 1, 7 y 0 (nuevamente se agregan tres más); por su parte desde el 7 se puede mover la ficha hasta el 2 y el 6 (la cantidad se incrementa en dos); si se toma el 8 como inicio se pueden alcanzar el 1 y el 3 (se adicionan dos movimientos); si se posiciona la ficha en el 9 las opciones para moverse son 2 y 4 (nuevamente se tienen dos movimientos); y por último si se sale desde el 0 los movimientos válidos son 4 y 6 (se suman los últimos dos). En total se pueden realizar veinte movimientos válidos con esta ficha.

Ahora, diseñe un algoritmo que permita calcular cuántos posibles movimientos válidos puede realizar la ficha del caballo, recibiendo como entrada la cantidad de movimientos a realizar desde el inicio, partiendo de todos los números. Por ejemplo, como mostramos anteriormente si la cantidad de movimientos es uno, la cantidad de movimientos válidos son veinte. Pero si la cantidad de movimientos son dos y se sale desde el 1 se puede ir hasta el 6 y el 8 (un movimiento), a continuación a partir del 6 hasta el 1, 7 y 0 (dos movimientos de la ficha), luego se sigue desde el 8 hasta el 1 y 3 (para alcanzar los dos movimientos de la ficha). En resumen se tienen cinco posibles movimientos válidos partiendo desde el 1 (1-6-1, 1-6-7, 1-6-0, 1-8-1 y 1-8-3) a estos se deben sumar todos los movimientos que resulten de partir de los demás número. En total la cantidad de posibles movimientos válidos para dos movimientos son 46. Una vez desarrollado el algoritmo complete la siguiente tabla.

Cantidad de movimientos	Posibilidades válidas
1	20
2	46
3	104
5	
8	
10	
15	
18	
21	
23	
32	

Pasemos a trabajar en otro ejemplo para fortalecer aún más nuestro pensamiento algorítmico, en este caso el problema de las n-reinas, el mismo consiste en ubicar n reinas en un tablero de ajedrez de tamaño $n \times n$, sin que las mismas se amenacen. Recuerde que la reina desplaza de manera horizontal, vertical y diagonal como se puede observar en la figura 2, además podemos ver una solución al problema de las 4 reinas. Nótese que una parte importante para resolver un problema es de que manera representar la solución, para este caso particular usamos un vector de n posiciones (columnas) y el valor almacenado representa la fila donde se ubica dicha reina.

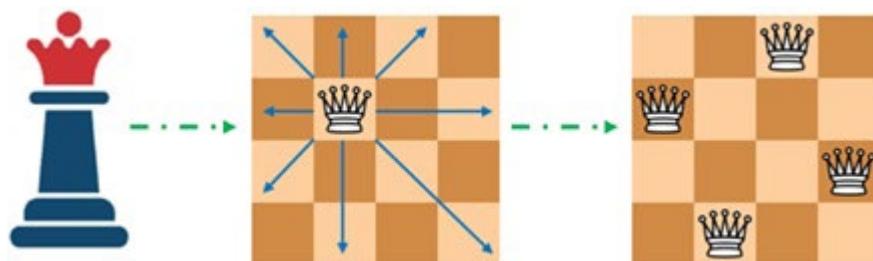


Figura 2. Problema de las n-reinas

Cuando haya entendido el problema y tenga una solución en mente, desarrolle un algoritmo que permita hallar al menos una solución para distintas cantidad de reinas, y luego complete la siguiente tabla.

n-reinas	Soluciones distintas	Todas las soluciones	Una solución
1	1	1	[0]
2	0	0	- <i>no hay</i>
3	0	0	- <i>no hay</i>
4	1	2	[1, 4, 0, 3]
5	2	10	
6	1	4	
7	6	40	
8	12	92	
9	46	352	
10	92	724	
15	285 053	2 279 184	



Del algoritmo a las estructuras de datos: ¡A implementar!

Para implementar los algoritmos y definir las estructuras de datos se puede utilizar cualquier lenguaje de programación como C, Java, Delphi, C++, Javascript. Como lo indica el título de este libro, se optó por trabajar con el lenguaje Python por su versatilidad, creciente popularidad y demanda en estos últimos años. A su vez, se buscó darle un enfoque ágil sin dejar de lado las metodologías estructuradas de estos temas. Los ejemplos están codificados de la manera clásica utilizando unos pocos comandos particulares del lenguaje, para que puedan ser codificados en cualquier otro lenguaje que se prefiera, sin realizar cambios significativos. Los ejemplos pueden ser ejecutados utilizando cualquier versión “3.x” de Python dado que la versión “2.x” está descontinuada a partir de enero del 2020 (si se trabaja con alguna versión anterior se deberá hacer algunas modificaciones para adaptar el código y que funcione en dicha versión).

¿Qué es Python?

Python es un lenguaje de programación creado por Guido Van Rossum a principios de los años noventa, es un lenguaje que posee una sintaxis clara y estructurada que favorece a generar un código legible. Es administrado por la Python Software Foundation, posee una licencia de código abierto, denominada Python Software Foundation License, que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1.

Se trata de un lenguaje interpretado o de script, con tipado dinámico, multiplataforma y orientado a objetos, además cuenta con un gran cantidad de librerías y frameworks disponibles. La sintaxis de Python es sencilla y cercana al lenguaje natural. Por estas razones se trata de uno de los mejores

lenguajes para comenzar a programar, –ya que es como programar en pseudocódigo–, y la curva de aprendizaje es bastante alta y rápida.

Es una tecnología muy utilizada en la actualidad, por su simplicidad y potencia permite realizar aplicaciones de una manera rápida, sencilla y óptima para el despliegue en distintas plataformas, además permite utilizar algoritmos de *machine learning* e inteligencia artificial en producción. Existen muchos casos de grandes empresas que utilizan Python en sus aplicaciones con gran éxito, tal es el caso de Google, la NASA, Netflix, Mercado libre, Spotify, Dropbox, Instagram, Pinterest, Globant, Sattellogic y un largo etcétera.

Un lenguaje interpretado o de script es aquel que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora. La ventaja de los lenguajes compilados es que su ejecución es más rápida. Sin embargo los lenguajes interpretados son más flexibles y más portables.

Python tiene, no obstante, muchas de las características de los lenguajes compilados, por lo que se podría decir que es seminterpretado. En Python, como en Java, Delphi y muchos otros lenguajes, el código fuente se traduce la primera vez que se ejecuta a un pseudocódigo máquina intermedio llamado *bytecode*, generando archivos “.pyc” o “.pyo”, que son los que se ejecutarán en sucesivas ocasiones.

La característica de tipado dinámico se refiere a que no es necesario declarar el tipo de dato que va a contener una variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo, lo cual le otorga una gran flexibilidad a la hora de programar.

En los lenguajes con esta característica no se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si tenemos una variable que contiene un texto (variable de tipo cadena o *string*) no podremos tratarla como una variable numérica, sin previamente realizar una conversión.

La programación orientada a objetos es un paradigma en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones e intercambio de mensajes entre los objetos. No obstante el lenguaje soporta también los paradigmas de programación imperativa y funcional (este último le permite al lenguaje añadir características avanzadas muy interesantes).

El intérprete de Python está disponible en muchas plataformas de las más comunes (Linux, UNIX, Mac OS, Solaris, DOS, Windows, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin tener que realizar ningún tipo de cambio. Esto nos permite desarrollar programas que sean portables de un sistema operativo a otro.

La filosofía de los creadores de Python entiende que la gran sencillez, flexibilidad y potencia del lenguaje debe ir acompañada de una correcta formación del programador que lo utiliza para poder realizar las actividades de desarrollo con criterio, dado que la inexperiencia de los nuevos programadores puede conducirlos a programar de manera incorrecta (aunque lleguen a un resultado).

CAPÍTULO II

Auto llamadas con algoritmos recursivos

Cuando se habla de recursividad se hace referencia a problemas en los cuales, dicho problema es parte de la solución. Se trata de un modelo matemático cuya definición incluye a sí mismo “se autodefine”; es decir que se llama a sí misma repetidamente hasta que satisface una determinada condición para evitar caer en un bucle infinito, y donde cada resultado depende de la resolución de la siguiente o anterior repetición, hasta alcanzar dicha condición de fin. Entonces, este modelo recursivo tiene dos características fundamentales para ser denominado como tal:

1. Debe tener al menos una condición clara de fin o caso base del problema, puede tener más de una de ser necesario.
2. Debe llamarse a sí mismo, es decir que el modelo o solución sea parte de su propia definición. Esta llamada puede ser única, si se llama una sola vez dentro de la función, o múltiple, si se realizan más de una llamada.



Figura 1. Definición gráfica de recursividad

A partir de esta definición, se puede decir que la recursividad es una técnica muy útil para desarrollar algoritmos recursivos que den solución a distintos problemas. Este modelo recursivo se lo debe pensar como una función recursiva que servirá para dar solución a ciertos problemas –estos son casos particulares donde la naturaleza del problema es recursiva–. No cualquier problema puede resolverse de forma recursiva, aunque muchas veces puede implementarse una solución recursiva para problemas que no son de esta naturaleza. Es importante aclarar que dentro de una función o algoritmo recursivo no deben existir ciclo o bucles, salvo algunos casos muy particulares para resolver alguna operación transversal a la función.

Se puede clasificar la recursividad de acuerdo a su tipo, pueden ser algoritmos **recursivos directos** o **indirectos**, los primeros se llaman a sí mismos mientras que los segundos llaman a un algoritmo (intermedio) y este llama al algoritmo inicial para generar la recursividad. Otra manera de clasificarla es respecto a la finalización de los algoritmos si son finales o no finales, es decir, si al finalizar las llamadas recursivas no queda más nada por hacer o aún quedan instrucciones por ejecutar.

Para comenzar, podemos observar un ejemplo sencillo: el problema del factorial, si se quiere calcular el factorial de un número n . La solución iterativa sería la siguiente (si $n=5$):

$$5! = 5 * 4 * 3 * 2 * 1$$

Pero si se lo piensa de forma recursiva, es decir, si se descompone el problema inicial en pequeños subproblemas y se resuelve una parte del problema en cada llamada (que quedará pendiente hasta que se alcance el caso base del problema), entonces se podría resolver de la siguiente manera:

$$5! = 5 * 4! \rightarrow \text{Llamada recursiva}$$

De esta forma, la solución al problema anterior quedaría así:

Problema	Solución parcial	Solución
$5!$	$5 * 4!$ (pendiente en espera de solución de $4!$)	Llamada recursiva $4!$
$4!$	$4 * 3!$ (pendiente en espera de solución de $3!$)	Llamada recursiva $3!$
$3!$	$3 * 2!$ (pendiente en espera de solución de $2!$)	Llamada recursiva $2!$
$2!$	$2 * 1!$ (pendiente en espera de solución de $1!$)	Llamada recursiva $1!$
$1!$	$1 * 0!$ (pendiente en espera de solución de $0!$)	Llamada recursiva $0!$
$0!$	$0! = 1$ (condición de fin o caso base)	1

Una vez que se alcanza la condición de fin, se empiezan a resolver las llamadas recursivas previas que quedaron pendientes.

Problema	Solución parcial	Solución
$0!$	$0!$	1 (condición de fin o caso base)
$1!$	$1 * 0!$	$1 * 0! - 1 * 1 = 1$ (utilizando resultado de $0!$)
$2!$	$2 * 1!$	$2 * 1! - 2 * 1 = 2$ (utilizando resultado de $1!$)
$3!$	$3 * 2!$	$3 * 2! - 3 * 2 = 6$ (utilizando resultado de $2!$)
$4!$	$4 * 3!$	$4 * 3! - 4 * 6 = 24$ (utilizando resultado de $3!$)
$5!$	$5 * 4!$	$5 * 4! - 5 * 24 = 120$ (utilizando resultado de $4!$)

Como se puede observar en el ejemplo anterior, mediante el modelo recursivo se puede dividir un problema en pequeños subproblemas, y para poder resolver cada uno de estos se necesitará de la solución del caso siguiente o anterior (dependiendo del problema). Estas quedarán pendiente en memoria para resolver cuando se alcance una condición de fin o caso base, y el algoritmo retroceda resolviendo cada una de estas llamadas que quedó en espera y liberando la memoria hasta obtener el resultado final.

A continuación se puede observar en la figura 2 cómo se programa esta técnica en Python para dar solución al problema antes planteado:

```
def factorial(numero):
    """Cálculo recursivo del factorial."""
    if(numero == 0):
        return 1
    else:
        return numero * factorial(numero-1)
```

Figura 2. Implementación de función recursiva en Python

Para poder entender cómo resolver un problema de forma recursiva, analicemos detalladamente los pasos del algoritmo anterior que dan solución al ejemplo del factorial:

1. Primero recuerde que en programación para representar el modelo recursivo se hace a través de una función recursiva, en la cual se define su nombre y los parámetros de entrada.
2. Mediante condicionales (*if-else*) se determinan las condiciones de fin. Debe tener al menos una.
3. Llamada recursiva de sí misma, es decir que dentro de la función debe llamarse a sí misma.
4. En el ejemplo gráfico (tablas) a medida que se reemplaza el valor de un problema por su resultado, se actualiza el valor del resultado a calcular. Cuando se desarrolla una función recursiva debemos recordar que cuando se realizan estas llamadas hay que actualizar los parámetros de entrada. En dicha llamada esto es sumamente importante ya que será lo que permitirá lograr alcanzar la condición de fin y no caer en un ciclo infinito de llamadas recursivas y que se desborde la pila de llamadas en memoria. En este caso particular cuando se llama a sí misma se le resta uno al valor de la variable *e*.
5. Como es una función debe devolver un resultado. En este caso puede devolver uno de los valores de las condiciones de fin, o el resultado de llamada recursiva con su correspondiente tratamiento.

Ahora veamos otro ejemplo. En esta ocasión en la figura 3 se puede observar el algoritmo recursivo e iterativo para resolver un mismo problema, obtener el valor en la sucesión de Fibonacci de un número n dado.

```
def fibonacciR(n):
    """Cálculo fibonacci recursivo."""
    if(n == 0 or n == 1):
        return n
    else: N <= 5 en 19 vez, 1, 2
        return fibonacciR(n-1) + fibonacciR(n-2)

def fibonacciI(n):
    """Cálculo fibonacci iterativo."""
    n0 = 0
    n1 = 1
    fib = 0
    if(n == 0 or n == 1):
        fib = n
    else:
        i = 2
        while (i <= n):
            fib = n0 + n1
            n0 = n1
            n1 = fib
            i += 1
    return fib
```

Figura 3. Algoritmo Fibonacci recursivo e iterativo

Claramente hay una diferencia entre ambas funciones. La primera –versión recursiva arriba en la figura– solo son cuatro líneas de código y refleja el modelo matemático de la sucesión de Fibonacci. En cambio la segunda –versión iterativa abajo en la figura– es necesario hacer un tratamiento particular para lograr representar el modelo de dicha sucesión que llevará algunas líneas más de código.

El modelo recursivo, como se dijo anteriormente, permite dividir un problema grande en problemas más pequeños sin alterar la naturaleza del problema y su solución. Sin embargo, esto no implica que esta sea la mejor solución. Estos problemas también pueden ser resueltos de manera iterativa, para ello es necesario utilizar alguna de las estructuras de control de ciclo y variables auxiliares para almacenar los resultados (acumuladores, contadores o auxiliares) que reemplace las llamadas recursivas. Esto muchas veces cambiará la estrategia utilizada para resolver el problema. Además al tener dos alternativas a la hora de resolver un determinado problema, es muy importante que se analicen las ventajas y desventajas de implementar y desarrollar una solución recursiva o iterativa.

El modelo recursivo brinda ciertas ventajas respecto al iterativo, entre ellas se puede destacar que el código a desarrollar es más claro y sencillo, dado que se respeta el modelo del problema y la solución. La mayor desventaja del modelo recursivo frente al iterativo es que, dependiendo de la

dimensión del problema a tratar y las estructuras de datos donde se implemente –en particular si este es muy grande–, la cantidad de llamadas recursivas necesarias para llegar hasta el caso base pueden ser muchas y podría desbordar la pila de memoria disponible para esta actividad. En cambio, el modelo iterativo solo requiere de un ciclo formado por una variable de control, más las variables auxiliares para los resultados. De tal forma se consume mucha menos cantidad de memoria.

En el siguiente capítulo se presentarán técnicas que permiten comparar un algoritmo recursivo frente a uno iterativo, de esta manera se logrará dominar los criterios suficientes para optar entre una implementación recursiva o una iterativa de un determinado algoritmo.

Guía de ejercicios prácticos

A continuación planteamos una serie de problemas. Para resolverlos se deberá desarrollar una función recursiva.

- I. Implementar una función que permita obtener el valor en la sucesión de Fibonacci para un número dado. ✓
- II. Implementar una función que calcule la suma de todos los números enteros comprendidos entre cero y un número entero positivo dado. ✓
- III. Implementar una función para calcular el producto de dos números enteros dados.
- IV. Implementar una función para calcular la potencia dado dos números enteros, el primero representa la base y segundo el exponente. ✓
- V. Desarrollar una función que permita convertir un número romano en un número decimal. ✓
- VI. Dada una secuencia de caracteres, obtener dicha secuencia invertida. ✓
- VII. Desarrollar un algoritmo que permita calcular la siguiente serie: ✓

$$h(n) = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

- VIII. Desarrollar un algoritmo que permita convertir un número entero en sistema decimal a sistema binario. ✓
- IX. Implementar una función para calcular el logaritmo entero de número n en una base b. Recuerde que:

$$\log_b\left(\frac{n}{b}\right) = \log_b n + \log_b b$$

- X. Desarrollar un algoritmo que cuente la cantidad de dígitos de un número entero.
- XI. Desarrollar un algoritmo que invierta un número entero sin convertirlo a cadena. ✓
- XII. Desarrollar el algoritmo de Euclides para calcular el máximo común divisor (MCD) de dos número entero. ✓
- XIII. Desarrollar el algoritmo de Euclides para calcular también el mínimo común múltiplo (MCM) de dos número entero. ✓
- XIV. Desarrollar un algoritmo que permita realizar la suma de los dígitos de un número entero, no se puede convertir el número a cadena. ✓

15. Desarrollar una función que permita calcular la raíz cuadrada entera de un número entero. Puede utilizar una función auxiliar para que la función principal solo reciba como parámetro el número a calcular su raíz.



16. Implementar un función recursiva que permita obtener el valor de a^n en una sucesión geométrica (o progresión geométrica) con un valor $a_1 = 2$ y una razón $r = -3$. Además desarrollar un algoritmo que permita visualizar todos los valores de dicha sucesión desde a_1 hasta a_n .

17. Escribir una función recursiva que permita mostrar los valores de un vector de atrás hacia adelante. ✓

18. Implementar una función recursiva que permita recorrer una matriz y mostrar sus valores. ✓

19. Dada la siguiente definición de sucesión recursiva, realizar una función recursiva que permita calcular el valor de un determinado número en dicha sucesión.

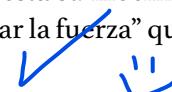
$$f(n) = \begin{cases} 2, & n = 1 \\ n + \frac{1}{f(n-1)}, & n \geq 2 \end{cases}$$



20. Desarrollar un algoritmo que permita implementar la búsqueda secuencial con centinela de manera recursiva, y permita determinar si un valor dado está o no en dicha lista. ✓

21. Dada una lista de valores ordenadas, desarrollar un algoritmo que modifique el método de búsqueda binaria para que funcione de forma recursiva, y permita determinar si un valor dado está o no en dicha lista. ✓ !!

22. El problema de la mochila Jedi. Suponga que un Jedi (Luke Skywalker, Obi-Wan Kenobi, Rey u otro, el que más le guste) está atrapado, pero muy cerca está su mochila que contiene muchos objetos. Implementar una función recursiva llamada “usar la fuerza” que le permita al Jedi “con ayuda de la fuerza” realizar las siguientes actividades:



- sacar los objetos de la mochila de a uno a la vez hasta encontrar un sable de luz o que no queden más objetos en la mochila;
- determinar si la mochila contiene un sable de luz y cuantos objetos fueron necesarios sacar para encontrarlo;
- Utilizar un vector para representar la mochila.

23. Salida del laberinto. Encontrar un camino que permita salir de un laberinto definido en una matriz de $[n \times n]$, solo se puede mover de una casilla a la vez –no se puede mover en diagonal– y que la misma sea adyacente y no esté marcada como pared. Se comenzará en la casilla $(0, 0)$ y se termina en la $(n-1, n-1)$. Se mueve a la siguiente casilla si es posible, cuando no se pueda avanzar hay que retroceder sobre los pasos dados en busca de un camino alternativo.



24. En el momento de la creación del mundo, los sacerdotes del templo de Brahma recibieron una plataforma de bronce sobre la cual había tres agujas de diamante. En la primera aguja estaban apilados setenta y cuatro discos de oro, cada una ligeramente menor que la que estaba debajo. A los sacerdotes se les encomendó la tarea de pasarlos todos desde la primera aguja a la tercera, con dos condiciones, solo puede moverse un disco a la vez, y ningún disco podrá ponerse encima de otro más pequeño. Se dijo a los sacerdotes que, cuando hubieran terminado de mover los discos, llegaría el fin del mundo. Resolver este problema de la Torre de Hanói.
- 
25. Desarrollar una función recursiva que permita calcular y mostrar por pantalla el triángulo de Pascal, para n filas utilizando una matriz auxiliar para guardar los resultados parciales.
26. Resuelva el problema de colocar las 8 reinas sobre un tablero de ajedrez sin que las mismas se amenacen.
27. El valor 1 376 256 pertenece a una sucesión geométrica cuya razón es 4, implementar un algoritmo para mostrar todos los valores de la sucesión hacia atrás hasta el valor de $a_1 = 5,25$.
28. Dada la siguiente definición de sucesión recursiva, realizar una función recursiva que permita calcular el valor de un determinado número en dicha sucesión.

$$f(n) = \begin{cases} 3, & n = 1 \\ f(n - 1) + 2n, & n \geq 2 \end{cases}$$


29. Desarrollar una función recursiva que permita calcular el método de la bisección de una función $f(x)$.
30. Desarrollar también una función recursiva que permita calcular el método de la secante de una función $f(x)$.
31. Por último, desarrollar otra función recursiva que permita calcular el método de Newton-Raphson de una función $f(x)$.

CAPÍTULO III

Analizando algoritmos en búsqueda de eficiencia

Durante el desarrollo de este capítulo se utilizarán dos términos claves, “ejemplar” y “problema”, que se deben diferenciar correctamente para no cometer errores. Cuando se hace referencia a un ejemplar se refiere a un caso particular de un problema (por ejemplo, determinar si en número 5 es primo). En cambio, cuando se menciona un problema hace referencia a todos los casos particulares comprendidos bajo el mismo problema (por ejemplo, determinar los números enteros primos).

Un algoritmo debe funcionar correctamente en todos los ejemplares o casos del problema que resuelve. Para demostrar que un algoritmo es incorrecto basta con encontrar un ejemplar de dicho problema que no se puede resolver de manera correcta.

¿Qué significa que un algoritmo sea eficiente?

Normalmente, cuando hay que resolver un problema es posible que existan distintos algoritmos adecuados para este y, obviamente, se querrá elegir el mejor. Entonces, ¿cómo elegir entre varios algoritmos para el mismo problema?

Si solo hay que resolver un par de casos de un problema sencillo, seguramente no será de mayor importancia con qué algoritmo se resuelva, seguramente la elección se inclinará hacia uno que sea más rápido programar o a uno que ya esté desarrollado, sin preocuparnos por las propiedades teóricas. Pero si se tienen que resolver muchos casos o el problema es complejo, el proceso de selección del algoritmo deberá ser más cuidadoso y con criterio.

Existen dos enfoques para la selección de un algoritmo:

1. Empírico (o a posteriori) consiste en programar las técnicas competidoras e ir probándolas en distintos casos en la computadora.
2. Teórico (o a priori) consiste en determinar matemáticamente la cantidad de recursos necesarios para cada uno de los algoritmos, en función del tamaño de los casos considerados. Los recursos que más nos interesan son el tiempo de computación o procesamiento (este es el más importante) y el espacio de almacenamiento, principalmente entendida como la memoria principal de la computadora.

A lo largo del libro compararemos los algoritmos a través del enfoque teórico tomando como base sus tiempos de ejecución. En este sentido, se considera eficiente a un algoritmo en función a su velocidad de ejecución.

Hay que tener en cuenta que el tamaño de un ejemplar se corresponde formalmente con el número de bits que se requieren para representarlo en una computadora. Sin embargo, para que el análisis sea más claro y sencillo conviene ser menos formales y utilizar la palabra tamaño para indicar un entero que mida de alguna forma el número de componentes de un ejemplar. Por ejemplo cuando se habla de ordenamiento, normalmente se medirá el tamaño de un ejemplar por la cantidad de ítems que hay que ordenar, ignorando el espacio en bit que se requerirá para almacenar cada uno de estos ítems en la computadora.

El enfoque teórico tiene la ventaja de que no depende ni de la computadora que se esté utilizando (hardware), ni del lenguaje de programación (software), ni siquiera de las habilidades del programador (persona). Además se ahorra el tiempo que se habría invertido en el desarrollo de un algoritmo ineficiente, como el tiempo de computación que se habría desperdiciado para comprobarlo. Y lo más importante, permite medir la eficiencia de un algoritmo para todos los casos de un problema (es decir todos los tamaños).

A diferencia del teórico, el enfoque empírico muchas veces propone realizar las pruebas con ejemplar de tamaño chico y moderado, dado que el tiempo de computación requerido es elevado. Esto es una desventaja, suele suceder que los nuevos algoritmos comienzan a comportarse mejor que sus predecesores cuando aumenta el tamaño del ejemplar.

Como no existe una unidad de medida para expresar la eficiencia de un algoritmo, en su lugar se la expresa según el tiempo requerido por un algoritmo. Se puede decir entonces que el algoritmo para un problema requiere un tiempo del orden de $t(n)$ para una función dada t , si existe una constante positiva c y una implementación del algoritmo capaz de resolver todos los casos de tamaño n en un tiempo que no sea superior a $ct(n)$ unidades de tiempo (la unidad de tiempo es arbitraria, pueden ser años, minutos, días, horas, segundos, milisegundos, etc.).

Esto mismo se puede definir formalmente:

$T(n) = O(f(n))$ si existe una constante c y un valor n_0 tales que $T(n) \leq c f(n)$ cuando $n > n_0$

Existen ciertos órdenes que se repiten con tanta frecuencia que tienen su propia denominación, como se describe a continuación, y sus funciones de crecimiento se observan en la figura 1 para valores de tamaño de uno a diez –si bien el rango de valores es pequeño– es suficiente para apreciar la diferencia entre los distintos órdenes de complejidad y cómo varía el tiempo de ejecución en base al tamaño de la entrada:

1. en el orden de $O(c)$, o de tiempo constante;
2. en el orden de $O(\log n)$, o de tiempo logarítmico;
3. en el orden de $O(n)$, o de tiempo lineal;
4. en el orden de $O(n \log n)$, o de tiempo casi lineal;
5. en el orden de $O(n^2)$, o de tiempo cuadrático;
6. en el orden de $O(n^3)$, o de tiempo cúbico;

7. en el orden de $O(n^k)$, o de tiempo polinómico;
8. en el orden de $O(c^n)$, o de tiempo exponencial;
9. en el orden de $O(n!)$, o de tiempo factorial.

Entre mayor sea n ,
más tiempo de
ejecución.

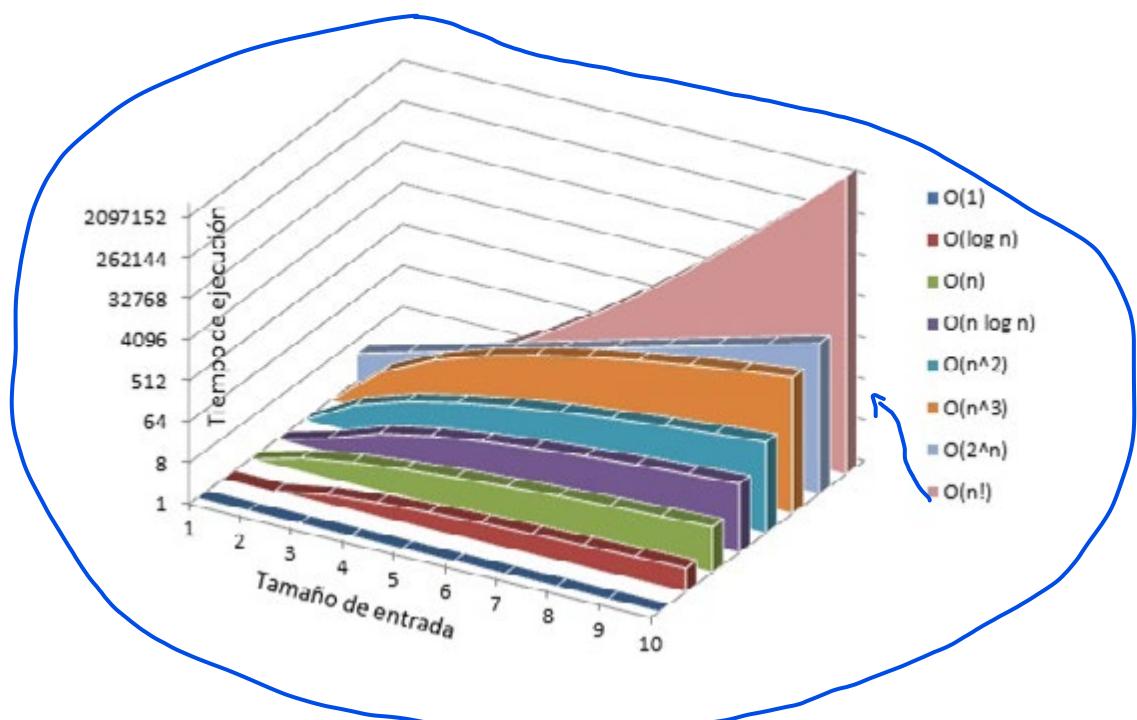


Figura 1. Gráfica de comparación de órdenes de crecimiento de algoritmos

Cuando se comparan dos algoritmos no deben olvidarse las constantes ocultas (o magnitud del orden) es decir un algoritmo en el orden de $O(n)$ es más eficiente que uno en el orden de $O(n^2)$, pero si la magnitud del primero es “hora” y la del segundo es “segundos”, el segundo algoritmo será más eficiente.

El peor caso: la alternativa más útil

Cuando se analiza un algoritmo se lo puede realizar desde tres enfoques útiles:

1. Mejor caso, poco útil dada la poca o escasa ocurrencia de dicho caso.
2. Caso medio, útil cuando los casos a resolver son muy volátiles e igualmente probables que ocurrán (es decir equiprobables). Requiere de mucha información a priori respecto de los casos a resolver. Esto en general es difícil de conocer.
3. Peor caso, es adecuado para problemas críticos donde se debe conocer el tiempo requerido para el peor caso. No requiere información respecto de los casos a resolver.

A lo largo del libro se trabajará con el análisis del peor caso, salvo que se indique lo contrario, esta elección se debe a que este enfoque nos garantiza que en el peor de los casos nuestro algoritmo se ejecutará en ese tiempo –es decir su cota superior–, aunque esto no implica que siempre tendrá dicho tiempo, en algunas ocasiones puede ser mejor. Ahora veamos algunos términos básicos requeridos para comprender el resto del capítulo:

Operación elemental: es aquella que se puede acotar por una constante que depende solamente de la implementación de dicha instrucción en una computadora o lenguaje de programación (por ejemplo una suma, una multiplicación o una asignación). Ya que el tiempo exacto requerido por cada operación elemental no es importante, se simplifica diciendo que estas se pueden ejecutar a coste unitario.

Notación asintótica: se denomina de esta manera porque trata acerca del comportamiento de funciones en el límite, es decir para valores significativamente grandes de sus parámetros. Esta notación nos permite determinar que un algoritmo es preferible respecto a otro incluso en casos de tamaño moderado o grande. Se puede pensar que n representa el tamaño del ejemplar sobre el cual es preciso que se aplique un determinado algoritmo, y en $t(n)$ representa la cantidad de un recurso dado que se invierte en ese ejemplar para la implementación de un algoritmo particular.

Considere ahora un algoritmo que requiere tres pasos para ser ejecutado, y que dichos pasos requieren un tiempo de $O(n)$, $O(n^3)$ y $O(n \log n)$, por lo tanto el algoritmo requiere un tiempo de $O(n + n^3 + n \log n)$.

Pero aunque el tiempo requerido por el algoritmo es lógicamente la suma de los tiempos requeridos por cada uno de sus pasos, el tiempo del algoritmo es el mayor de los tiempos requeridos para ejecutar cada uno de los pasos de dicho algoritmo.

$$O(n + n^3 + n \log n) = O(\max(n, n^3, n \log n)) = O(n^3)$$

Reglas elementales para el análisis de algoritmos

Cuando se dispone de distintos algoritmos para resolver un problema, es necesario definir cuál de ellos es el más eficiente. Una herramienta esencial para este propósito es el análisis de algoritmos. Este análisis suele efectuarse desde dentro del algoritmo hacia afuera, determinando el tiempo requerido por las instrucciones secuenciales y cómo se combinan estos tiempos con las estructuras de control que enlazan las distintas partes del algoritmo. A continuación se presentan las reglas básicas para el análisis de algoritmos:

Operaciones elementales: el coste computacional de una operación elemental es i (asignación, entrada/salida, operación aritmética que no desborde el tamaño de la variable utilizada). La mayoría de las instrucciones de un algoritmo son operaciones elementales que están en el orden de $O(i)$.

Secuencia: en un fragmento de código, su coste computacional será la suma del coste individual de cada una de sus operaciones o fragmentos de código, aplicando la regla anterior. Es decir si se

cuenta con un fragmento de código compuesto de cinco operaciones elementales, el coste de dicho fragmento será $O(5)$.

Si se tiene dos fragmentos de códigos F_1 de coste α y F_2 de coste β que forman un programa, el coste de dicho programa será la suma de ambos fragmentos $O(\alpha + \beta)$.

Condicional *if-else* (decisión): si g es el coste de la evaluación de un condicional *if*, su valor es de $O(1)$ por cada condición que se deba evaluar (considerando siempre el peor caso, en el que se deba realizar todas las comparaciones dependiendo del operador lógico usado). A esto se le suma el coste de la rama de mayor valor, sea esta la verdadera o falsa (considerando las anidadas si corresponde), aplicando las reglas anteriores.

Es decir el coste de un fragmento condicional es del orden de $O(g + \max(\text{rama verdadera}, \text{rama falsa}))$.

Ciclo *for*: en los ciclos con contador explícito (ciclo desde-hasta) que encierran un fragmento de código F_1 de coste α , el coste total será el producto del número de iteraciones del ciclo n por el coste del fragmento de código a ejecutar F_1 , si dentro de dicho fragmento de código no existe presencia de otro ciclo y trabajamos con un n , significativamente grande que varía en función del tamaño de entrada. El coste de dicho fragmento se considera como una constante. Y el mismo se considera despreciable o de $O(1)$ respecto al número de iteraciones del ciclo.

Es decir el coste total estará en el orden de $O(n * O(1))$ lo que significa que es de $O(n)$. Si dentro del F_1 tenemos la presencia de otro ciclo del mismo tipo, el coste total estará en el orden de $O(n * n * O(1))$ lo que significa que es de $O(n^2)$.

Ciclo *while*: en este tipo de ciclos se aplica la regla anterior, solo se debe tener en cuenta que a veces no se cuenta con una variable de control numérica, sino que depende del tamaño de las estructuras con las que se esté trabajando y su dimensión, o del tipo de actividad que se realice dentro de dicho ciclo.

Recursividad: el cálculo del coste total de un algoritmo recursivo no es tan sencillo como los casos que vimos previamente. Para realizar este cálculo existen dos técnicas comúnmente utilizadas: ecuaciones recurrentes y teoremas maestros. Para realizar la primera de ellas se busca una ecuación que elimine la recursividad para poder obtener el orden de dicha función, muchas veces esto no es una tarea sencilla. En cambio, el segundo utiliza funciones condicionales y condiciones de regularidad para realizar el cálculo del coste. Estas técnicas antes mencionadas son avanzadas y exceden los alcances de este libro por lo que solo se explicarán algunos ejemplos utilizando la técnica de ecuaciones recurrentes para determinar el orden de funciones recurrentes.

Ahora se analizará el siguiente ejemplo siguiendo las reglas previamente definidas para determinar el orden del algoritmo “numeros_pares_impares” que se observa en la figura 2, dicho algoritmo es iterativo.

```

def numeros_pares_impares(numero):
    """Muestra los números pares e impares."""
    cont_imp = 0
    for i in range(1, numero+1):
        if(i % 2 == 0): → O(n)
            print(numero, 'Es Par') O(1)
        else: → O(2)
            print(numero, 'Es Impar')
            cont_imp += 1 → O(2)

    print('Cantidad de Números Impares:', cont_imp)

```

Figura 2. Código de algoritmo “numeros_pares_impares”

El análisis de un algoritmo se realiza desde adentro hacia afuera, entonces lo primero que determinamos es el orden del condicional ($\text{if } (i \% 2 == 0)$) y sus dos ramas la verdadera y la falsa. Entonces según la regla del condicional:

1. solo tenemos una comparación en el $\text{if } O(1)$,
2. la rama verdadera solo tiene una operación elemental $O(1)$,
3. la rama falsa tiene dos operación elementales $O(2)$.

El resultado del condicional es del orden $O(3)$.

Continuamos el análisis con el ciclo for (de i hasta n), según la regla del punto ciclo for :

1. el orden del ciclo for es de $O(n)$,
2. el orden del condicional es despreciable respecto al del ciclo for y se considera en el orden de $O(1)$.

El resultado del ciclo for es del orden $O(n * 1)$ o sea $O(n)$, las otras dos instrucciones restantes del algoritmo (la primera y la última) son operaciones elementales de $O(1)$ y se desprecian respecto al orden del ciclo for . El resultado final es que el algoritmo “numeros_pares_impares” es del orden de $O(n)$.

Entonces se puede decir que la función de complejidad de un algoritmo es una función $f: \mathbb{N} \rightarrow \mathbb{N}$ tal que $f(n)$ es la cantidad de operaciones que realiza el algoritmo en el peor caso cuando toma una entrada de tamaño n .

Algunas observaciones a tener en cuenta:

1. Se mide la cantidad de operaciones en lugar del tiempo total de ejecución (ya que el tiempo de ejecución es dependiente del hardware y no del algoritmo).
2. Interesa el peor caso del algoritmo.

3. La complejidad se mide en función del tamaño de la entrada y no de una entrada en particular.

Ahora resolveremos mediante ecuaciones recurrentes los ejemplos de las funciones factorial, Fibonacci y búsqueda binaria recursivas vistas en el capítulo anterior:

Para el caso de factorial recursivo la ecuación recursiva queda de la siguiente manera, tenemos la llamada recursiva con $(n-1)$ y se suma una constante que es el caso base que contempla una operación elemental: $T(n) = T(n-1) + c$ con $T(0) = 1$.

Ahora resolveremos la ecuación de recurrencia sustituyéndola por su igualdad hasta llegar a una cierta $T(n)$ conocida para calcular la cota superior:

$$T(n) = T(n-1) + c$$

$$= T(n-2) + c + c$$

$$= T(n-3) + c + 2c$$

$$= T(n-k) + k*c$$

Cuando $k = n-1$ tenemos que

$$T(n) = T(1) + (n-1)*c$$

$$= 1 + (n-1)$$

$$= n$$

Finalmente obtendremos que la función factorial recursivo es del orden de $T(n) = O(n)$.

Para el caso de Fibonacci recursivo la ecuación queda de la siguiente forma: tenemos las llamadas recursivas $(n-1)$ y $(n-2)$ y se suma constante dado que es el caso base, el mismo es una operación elemental: $T(n) = T(n-1) + T(n-2) + c$ con $T(0) = 0$ y $T(1) = 1$

Ahora debemos resolver la ecuación de recurrencia igual que en caso anterior:

$$T(n) = T(n-1) + T(n-2) + c$$

$$= 2T(n-1) + c$$

$$= 2(2T(n-2) + c) + c$$

$$= 2^2T(n-2) + 2^2*c$$

$$= 2^kT(n-k) + 2^{k*c}$$

Cuando $k = n-1$ tenemos que

$$T(n) = 2^{n-1}T(0) + 2^{n-1}*c$$

$$= I + (n-1)$$

$$= 2^{n-1}$$

Entonces podemos establecer que Fibonacci recursivo es del orden de $T(n) \cong 2^{n-1}$.

Finalmente veamos el caso de la búsqueda binaria recursiva. La ecuación recursiva para la misma quedaría de la siguiente forma: $T(n) = T(n/2^i) + c$ con $T(n \leq 1) = I$ y la variable i que toma valores de I a $\log n$ como se demuestra a continuación.

Ahora pasemos a resolver la ecuación recursiva:

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

$$= T(n/4) + c + c$$

$$= T(n/8) + c + 2c$$

$$= T(n/2^k) + k*c$$

Podemos determinar entonces una ecuación que nos indique el valor de k , es decir, cuantas veces tenemos que particionar el vector hasta que solo quede un elemento para comparar:

$$I = n/2^k$$

$$I * 2^k = n$$

$$2^k = n$$

$$\log_2 2^k = \log n$$

$$k \log_2 2 = \log n$$

$$k * I = \log n$$

$$k = \log n$$

Como la ecuación $n/2^k = I$ se resuelve en tiempo $k = \log_2 n$, nos queda que

$$T(n) = T(n/2^k) + k*c$$

$$= T(I) + \log_2 n$$

Entonces de esta manera obtenemos que la función búsqueda binaria recursiva es del orden de $T(n) \cong \log n$, dado que si n no es múltiplo de dos no obtendremos un numero entero como resultado.

Como puede observarse, el cálculo de la complejidad de una función recursiva no es una actividad trivial. Puede realizarse de distintas manera pero no siempre de forma clara y sencilla –como en una función iterativa–. Es más bien una tarea bastante artesanal que requiere mucho criterio y experiencia por parte de quien la realice.

Guía de ejercicios prácticos

A continuación se plantean una serie de fragmentos de códigos. Habrá que realizar un análisis de algoritmo para determinar su coste computacional y obtener su orden.

1. Fragmento de código condicional:

```
if num > 10:  
    print('es mayor que 10')  
    print('su cuadrado es 100')  
    aux = num % 2  
else:  
    print('es menor que 10')  
    print('su cubo es 1000')  
    aux = num // 2  
    print(aux)  
    producto = num * 2
```

2. Fragmento de código condicional:

```
if num > 10:  
    print('es mayor que 10')  
    print('su cuadrado es 100')  
    aux = num % 2  
    if(num % 2 == 0):  
        print('es divisible por 2')  
    else:  
        print('es divisible por 2')  
        num = num + 1  
        aux = num * num * num  
else:  
    print('es menor que 10')  
    print('su cubo es 1000')  
    aux = num // 2  
    print(aux)  
    producto = num * 2  
    if(producto % 3 == 0):  
        print('es divisible por 3')
```

3. Fragmento de código de ciclo:

```
for i in range(0, n):
    print('inicio ciclo j')
    for j in range(0, m):
        print('inicio ciclo k')
        for k in range(0, 3):
            print(lista[k])
```

4. Fragmento de código de ciclo:

```
for i in range(0, n):
    if(aux is not None):
        print('inicio ciclo j')
        for j in range(0, m):
            print(num + (j * valor))
    else:
        print('inicio ciclo k')
        for k in range(0, 1000):
            print(aux[k] + num)
```

5. Fragmento de código condicional:

```
if num > 10:
    print('es mayor que 10')
    print('su cuadrado es 100')
    aux = num % 2
    print('tabla de multiplicar')
    for i in range(1, 16):
        print(i, num * i)
else:
    print('es menor que 10')
    print('su cubo es 1000')
    aux = num // 2
    print(aux)
    producto = num * 2
    if(producto % 3 == 0):
        print('es divisible por 3')
```

6. Fragmento de código:

```
numero = int(input('ingrese un número'))
while (numero != 0) and (len(lista) < 10000):
    lista.append(numero)
    numero = int(input('ingrese número'))

for i in range(0, len(lista)):
    print(lista[i])
```

7. Búsqueda binaria iterativa:

```
while (p <= u) and (pos == -1):
    med = (p + u) // 2
    if(lista[med] == x):
        pos = med
    else:
        if(x > lista[med]):
            p = med + 1
        else:
            p = med - 1
```

8. Fragmento de código (mezcla de listas):

```
i = 0
j = 0

while (i < len(L1)) and (j < len(L2)):
    if(L1[i] < L2[j]):
        L3.append(L1[i])
        i += 1
    else:
        L3.append(L2[j])
        j += 1

if(i == len(L1)):
    for k in range(j, len(L2)):
        L3.append(L2[k])
else:
    for k in range(j, len(L1)):
        L3.append(L1[k])
```

9. Fragmento de código:

```
for i in range(0, n):
    ac = 0
    for j in range(0, m):
        ac = ac + lista[i, j]
    aux[i] = ac

i = 0
while (i < len(aux)):
    print(aux[i])
    i += 1
```

10. Fragmento de código:

```
for i in range(0, n-1):
    min = lista[inicio, n-1]
    pos_min = n-1
    if(min == 0):
        min = lista[inicio, n-2]
        pos_min = n-2
    for j in range(0, n-1):
        if(min > lista[inicio, j] and lista[inicio, j] != 0
           and j not in camino):
            min = lista[inicio, j]
            pos_min = j
    camino.append(pos_min)
    inicio = pos_min
```

II. Código de multiplicación de dos matrices M1[n x m] y M2[m x o]:

```
for i in range(0, n):
    for k in range(0, o):
        aux = 0
        for j in range(0, m):
            aux = aux + (M1[i][j] * M2[j][k])
        M3[i][k] = aux
```

12. Código de suma de dos matrices de [n x n].

```
for i in range(0, n):
    for j in range(0, n):
        M3[i][j] = M1[i][j] + M2[i][j]
```

13. Código para calcular la traza de un matriz cuadrada.

```
traza = 0
for i in range(0, n):
    traza += M2[i][i]
print(traza)
```

14. Código para calcular el determinante de una matriz cuadrada de [3 x 3], regla de Sarrus.

```
aux = 0
for o in range(0, m):
    temp = 1
    k = o
    for i in range(0, m):
        temp = temp * M1[i][k]
        k += 1
        if(k == m):
            k = 0
    aux += temp

for o in range(m-1, -1, -1):
    temp = 1
    k = o
    for i in range(0, m):
        temp = temp * M1[i][k]
        k -= 1
        if(k == -1):
            k = m-1
    aux -= temp

print('Determinante:', aux)
```

15. Función para determinar si un número es primo.

```
def es_primo(numero):
    """Determina si un número entero positivo es primo o no."""
    if(numero <= 1):
        return False
    else:
        cont = 0
        for i in range(2, numero+1):
            if(numero % i == 0):
                cont += 1
        if(cont == 1):
            return True
        else:
            return False
```

16. Función factorial iterativa.

```
def factorial(numero):
    """Cálculo iterativo del factorial."""
    factorial = 1
    for i in range(1, numero+1):
        factorial = factorial * i
    return factorial
```

17. Función Fibonacci iterativa.

```
def fibonacci(numero):
    """Cálcula el valor de un numero en la sucesión de fibonacci."""
    fib1 = 0
    fib2 = 1
    if(numero == 0):
        return fib1
    elif(numero == 1):
        return fib2
    else:
        resultado = 0
        for i in range(2, numero+1):
            resultado = fib1 + fib2
            fib1 = fib2
            fib2 = resultado
        return resultado
```


CAPÍTULO IV

Buscar es más fácil cuando las cosas están ordenadas

Cuando se trabaja con conjuntos de datos almacenados en estructuras de datos simples sean en memoria o en disco, vectores o archivos respectivamente, surge la necesidad de llevar a cabo dos actividades fundamentales para la gestión de estos datos el *ordenamiento* y la *búsqueda*.

Pero, ¿*Por qué tenemos que ordenar los datos?* Básicamente por dos razones, primero es mucho más fácil para una persona poder leerlos y segundo facilita poder buscar un elemento dentro de dichos datos. Normalmente cuando se trabaja en la gestión de datos uno de los principales requerimientos que aparece es la necesidad de listarlos de manera ordenada, ya sea de forma ascendente o descendente (o de acuerdo a algún criterio específico). Independientemente de la técnica, método o forma de hacerlo, no es común mostrar los datos de forma desordenada o respetando el orden en que han sido cargados. Por ello, es necesario *ordenar* los datos, por más que se sigan manteniendo almacenados de forma desordenada. A continuación vamos a analizar los algoritmos clásicos de ordenamiento y además haremos la correspondiente implementación en Python. También realizaremos una comparación entre los distintos algoritmos de acuerdo a varios aspectos: su complejidad temporal usando notación O grande como vimos en el capítulo anterior, complejidad espacial (en cuanto al uso de memoria), la estabilidad de dicho método –es decir que si hay valores repetidos estos mantengan su orden original respecto de los demás–, si utiliza almacenamiento interno o externo –es decir si almacena los datos en memoria o en disco–, si es un algoritmo recursivo o no recursivo y, finalmente, el tipo de comparación utilizada por el algoritmos, es decir si es por comparación o no. Estos métodos suelen aplicarse sobre vectores, en nuestro caso como en Python los vectores no son una estructura por defecto del lenguaje utilizaremos listas.

Las burbujas, nuestras aliadas para ordenar de una manera simple

Este es un algoritmo cuyo funcionamiento es de tipo *intercambio*. Su nombre se deriva de cómo los datos se desplazan por la lista durante el intercambio como si fueran burbujas, las más grandes (en nuestro caso los valores mayores) llegan antes a la superficie (el final de la lista) y las más pequeñas quedan debajo (el principio de la lista). La esencia del algoritmo es realizar iteraciones sobre los elementos de la lista y comparar cada par de valores adyacentes y si estos no están en orden los intercambia –sino quedan en el mismo lugar– y continúa con el siguiente par de valores, logrando que los elementos mayores se desplacen al final de la lista. En la figura 1 se presenta el esquema de funcionamiento de este algoritmo para ordenar una lista de números.

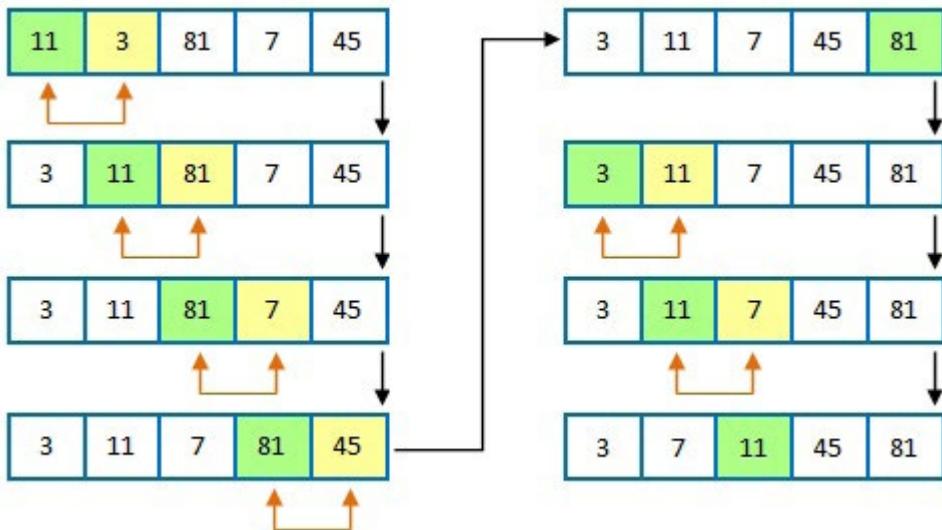


Figura 1. Esquema de funcionamiento del algoritmo burbuja

Por su parte la implementación de dicho algoritmo es muy sencilla y la podemos observar en la figura 2.

```
def burbuja(lista):
    """Método de ordenamiento burbuja."""
    for i in range(0, len(lista)-1):
        for j in range(0, len(lista)-i-1):
            if(lista[j] > lista[j+1]):
                lista[j], lista[j+1] = lista[j+1], lista[j]
```

Figura 2. Método de ordenamiento burbuja

Interpretando el código de la figura anterior, básicamente el algoritmo está compuesto por dos ciclos *para* el primero que indica la cantidad de elementos a ordenar con la variable *i* de 0 a (*n*-1) donde *n* representa el tamaño de la lista –se le resta uno dado que si la lista por ejemplo tiene 5 elementos al ordenar 4 el restante ya queda en su correspondiente lugar–, el segundo es controlado por la variable *j* de 0 a (*n*-*i*-1) donde *n* es el tamaño de la lista, *i* representa las iteraciones del ciclo del punto anterior, y se le resta uno debido a que la lista inicia en cero. Este ciclo se repite por cada iteración del ciclo anterior, pero en cada iteración la cantidad de elementos tratados se decrementa, durante las iteraciones se comparan los elementos de la posición actual del índice *j* con el siguiente (*j*+1) están desordenados. Si la comparación del punto anterior es verdadera, se realiza el *intercambio* entre los elementos de las posiciones *i* y *j*. Esto produce el desplazamiento de los valores por la lista.

Uno de los problemas que presenta este algoritmo es que si la lista está ordenada o al ordenar una cierta cantidad de elementos la lista ya queda ordenada –como en la figura 1–. El algoritmo seguirá haciendo iteraciones sin realizar intercambios. Es decir, estará desperdiциando tiempo de procesamiento y recursos, para lo cual podemos hacer una modificación del algoritmo anterior para mejorar su rendimiento en determinados casos, pero cabe aclarar que la complejidad del algoritmo será la misma en el peor de los casos. Esta nueva versión del algoritmo la denominaremos “burbuja mejorado” y su implementación se presenta en la figura 3.

```

def burbuja_mejorado(lista):
    """Método de ordenamiento burbuja mejorado."""
    i = 0
    control = True
    while (i < len(lista)-2) and control:
        control = False
        for j in range(0, len(lista)-i-1):
            if(lista[j] > lista[j+1]):
                lista[j], lista[j+1] = lista[j+1], lista[j]
                control = True
        i += 1

```

Figura 3. Método de ordenamiento burbuja mejorado

Realicemos un breve análisis de las modificaciones del algoritmo, en primer lugar necesitaremos una variable booleana de control, para determinar si se debemos seguir ordenando o la lista ya se encuentra ordenada y tenemos que detener el algoritmo. Además remplazamos el primer ciclo (*para*) del algoritmo original con un *mientras*, esto nos permitirá salir del ciclo si al cabo de cierta cantidad de iteraciones la lista ya se encuentra ordenada utilizando las variables de control. En cada iteración de este ciclo *mientras* asignamos a nuestra variable de control (o *flag*) el valor falso, luego si se realiza algún *intercambio* dentro del ciclo *para* le asignaremos valor verdadero. Entonces luego de cada iteración del ciclo mientras, si se produjo algún *intercambio* en la iteración anterior la variable control tendrá valor verdadero y continuara el algoritmo una iteración mas; en el caso contrario si no hubo intercambios la variable control tiene valor falso y significa que el vector esta ordenado por lo cual el algoritmo se detendrá.

Otra implementación de este algoritmo es el ordenamiento cóctel también conocido como “burbuja bidireccional”, de hecho la esencia del algoritmo es similar por lo cual su complejidad será la misma. La diferencia como su nombre lo indica es su cualidad bidireccional, es decir, luego de cada iteración *ordena* el elemento mayor (cuando va desde el principio al final) y el menor (cuando vuelve del final al principio). Este algoritmo también utiliza un ciclo mientras para detenerse cuando el vector se encuentre ordenado, su implementación se puede ver en la figura 4, como el algoritmo no varía demasiado de lo visto anteriormente la interpretación del mismo quedará a cargo del lector.

```

def coctel(lista):
    """Método de ordenamiento cóctel o burbuja bidireccional."""
    izquierda = 0
    derecha = len(lista)-1
    control = True
    while (izquierda < derecha) and control:
        control = False
        for i in range(izquierda, derecha):
            if(lista[i] > lista[i+1]):
                control = True
                lista[i], lista[i+1] = lista[i+1], lista[i]
        derecha -= 1
        for j in range(derecha, izquierda, -1):
            if(lista[j] < lista[j-1]):
                control = True
                lista[j], lista[j-1] = lista[j-1], lista[j]
        izquierda += 1

```

Figura 4. Método de ordenamiento cóctel

Solo nos resta hacer una tabla comparativa de este algoritmo de acuerdo a los aspectos mencionados anteriormente.

Complejidad temporal	O(n ²)
Complejidad espacial	En el lugar
Estabilidad	Estable
Interno /Externo	Interno
Recursivo/No recursivo	No recursivo
Ordenamiento por comparación	Comparativo

La selección natural también se valida en los algoritmos aplicando ordenamiento por selección

Este es un algoritmo cuyo funcionamiento es del tipo *selección*, su naturaleza de funcionamiento es recorrer iterativamente la lista de elementos en busca del menor y luego lo selecciona para ubicarlo a la izquierda de la lista, pero también se puede buscar el mayor y moverlo a la derecha. Es decir que en cada iteración se recorre el conjunto de elementos de la lista y se selecciona el “mejor”, es decir el menor o mayor, para ubicarlo en su correspondiente posición. Veamos a continuación en la figura 5 un ejemplo del funcionamiento del algoritmo.

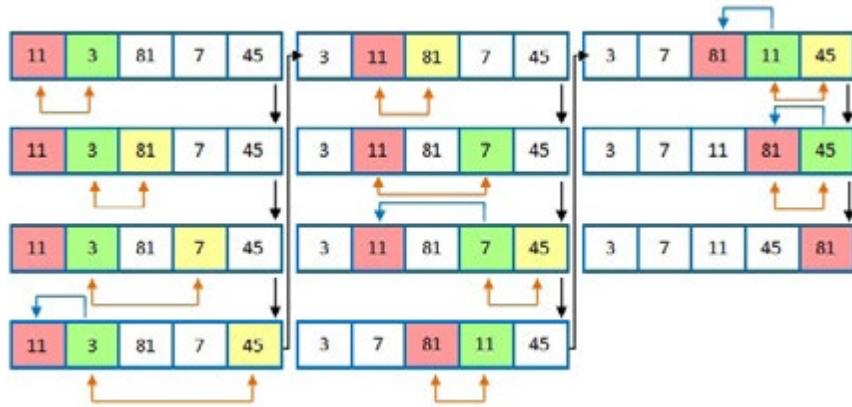


Figura 5. Funcionamiento del algoritmo selección

En cuanto a la implementación de este algoritmo de ordenamiento la podemos ver a continuación en la figura 6.

```
def seleccion(lista):
    """Método de ordenamiento selección."""
    for i in range(0, len(lista)-1):
        minimo = i
        for j in range(i+1, len(lista)):
            if(lista[j] < lista[minimo]):
                minimo = j
        lista[i], lista[minimo] = lista[minimo], lista[i]
```

Figura 6. Método de ordenamiento selección

Analicemos ahora el algoritmo en detalle. Este está compuesto de dos ciclos: un ciclo *para* de 0 a $(n-1)$ determina la cantidad de iteraciones para ordenar los elementos y el segundo ciclo *para* de $(i+1)$ a n , donde n es el tamaño de la lista e i las iteraciones del ciclo anterior. Al comienzo de cada iteración del primer ciclo se determina el valor del índice del elemento mínimo parcial en la lista, durante cada iteración del segundo ciclo *para* se compara si el elemento de la posición j es menor que el de la posición mínimo. Si la comparación es verdadera se actualiza el valor de la variable *minimo* por el valor de j y finalmente se realiza el *intercambio* de los valores almacenados en la posición i (donde debería quedar el menor de la iteración actual) y en la posición *minimo* (el elemento menor de los no ordenados).

Finalmente haremos la tabla comparativa de este algoritmo siguiendo los aspectos anteriormente especificados.

Complejidad temporal	$O(n^2)$
Complejidad espacial	En el lugar
Estabilidad	Inestable
Interno /Externo	Interno
Recursivo/No recursivo	No recursivo
Ordenamiento por comparación	Comparativo

Insertar elementos en el orden correcto aplicando ordenamiento por inserción

Este algoritmo cuyo funcionamiento es del tipo de *inserción*, opera de una forma muy natural para las personas. Inicialmente toma como único elemento ordenado el primero de la lista (k elemento), luego toma el siguiente elemento ($k+1$ elemento) y se compara con todos los elementos ordenados, desplazando a la derecha los elementos mayores y se detiene cuando encuentra un elemento menor o cuando no hay más elementos a la izquierda de la lista (es decir es el elemento menor). Veamos la forma de operar de este algoritmo en la figura 7 así podremos comprender su esencia.

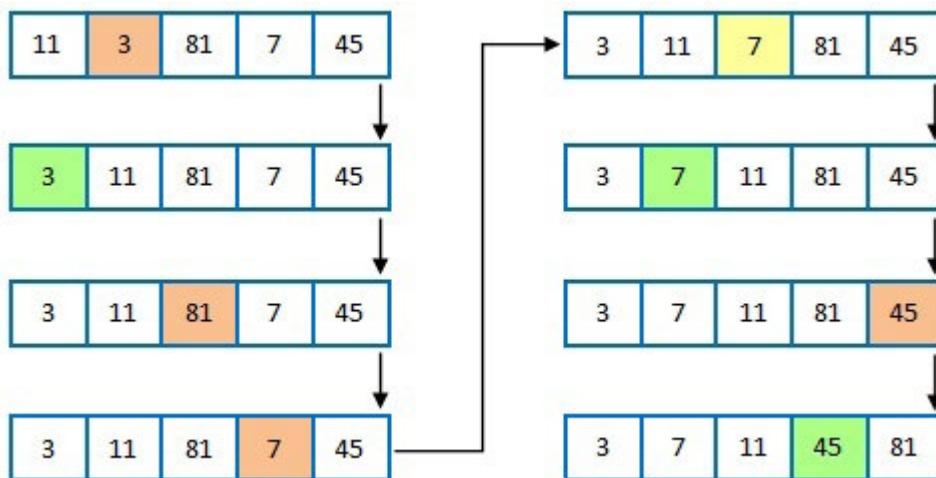


Figura 7. Forma de operar del algoritmo inserción

A continuación en la figura 8 se presenta la implementación del algoritmo por inserción para ordenar una lista.

```
def insercion(lista):
    """Método de ordenamiento inserción."""
    for i in range(1, len(lista)+1):
        k = i-1
        while (k > 0) and (lista[k] < lista[k-1]):
            lista[k], lista[k-1] = lista[k-1], lista[k]
            k -= 1
```

Figura 8. Método de ordenamiento inserción

Vamos a describir en detalle este algoritmo, el cual utiliza dos ciclos, un ciclo *para* de i a (n) donde n es el número de elementos de la lista, además comienza desde uno dado que se considera que el primer elemento de la lista está ordenado. A la variable k se le asigna el valor (i), es decir, el elemento que se va a *ordenar* (o *insertar*) en cada iteración del ciclo *para* y el ciclo *mientras* el valor k sea menor que el anterior o hasta el principio de la lista, se deben *intercambiar* los valores de las posiciones k y ($k-1$), generando un desplazamiento a la izquierda hasta encontrar la posición de *inserción* del elemento y se decremente la variable k para ver si es necesario continuar con el desplazamiento del elemento a la izquierda.

Complejidad temporal	$O(n^2)$
Complejidad espacial	En el lugar
Estabilidad	Estable
Interno /Externo	Interno
Recursivo/No recursivo	No recursivo
Ordenamiento por comparación	Comparativo

Los algoritmos que describimos a continuación son recursivos, por lo que se recomienda que si el lector no conoce el concepto de recursividad, se remita al capítulo II, para comprender el concepto antes de continuar con la lectura de este capítulo y así poder comprender en detalle estos algoritmos.

Ordenando elementos a toda velocidad con ordenamiento rápido

Este es un algoritmo cuyo funcionamiento es del tipo *partición* que fue creado por Charles Antony Richard Hoare en 1961¹ también conocido como *quicksort*, funciona seleccionando un elemento de lista a ordenar, al que denominaremos pivote, luego se reubicarán los elementos menores a la izquierda y los mayores a la derecha; luego el pivote se reubica en el lugar que le corresponde en la lista, de esta manera nos quedan dos sublistas separadas por el elemento pivote. Finalmente se llama recursivamente a cada sublista mientras tengan más de un elemento. A continuación en la figura 9 podremos apreciar el esquema del funcionamiento de este algoritmo.

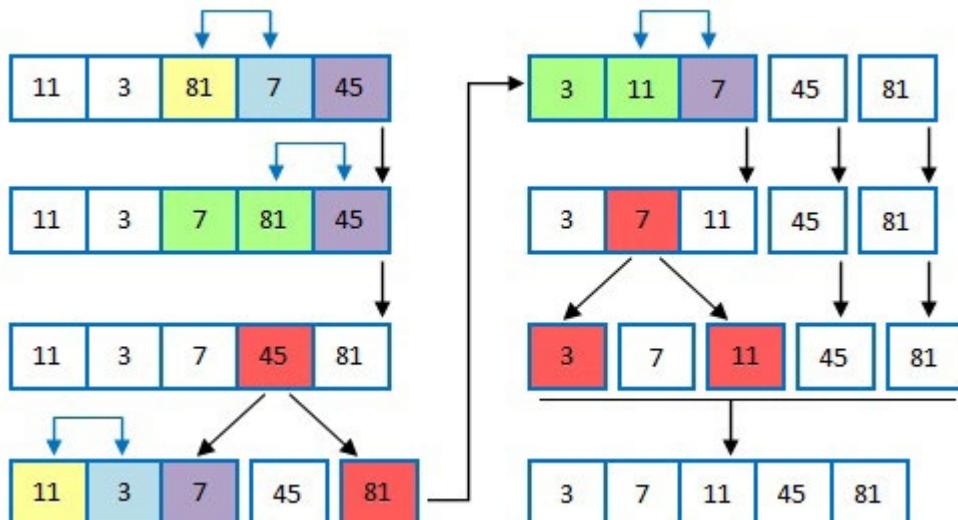


Figura 9. Esquema de funcionamiento del algoritmo de ordenamiento rápido

¹ Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". Comm. ACM. 4 (7): 321. doi:10.1145/366622.366644

En cuanto a la implementación de este algoritmo la podemos observar a continuación en la figura 10.

```
def quicksort(lista, primero, ultimo):
    """Método de ordenamiento quicksort."""
    izquierda = primero
    derecha = ultimo-1
    pivote = ultimo
    while (izquierda < derecha):
        while (lista[izquierda] < lista[pivote]) and (izquierda <= derecha):
            izquierda += 1
        while (lista[derecha] > lista[pivote]) and (derecha >= izquierda):
            derecha -= 1
        if(izquierda < derecha):
            lista[izquierda], lista[derecha] = lista[derecha], lista[izquierda]
        if(lista[pivote] < lista[izquierda]):
            lista[izquierda], lista[pivote] = lista[pivote], lista[izquierda]
        if(primer < izquierda):
            quicksort(lista, primero, izquierda-1)
        if(ultimo > derecha):
            quicksort(lista, izquierda+1, ultimo)
```

Figura 10. Método de ordenamiento quicksort

Desglosemos analíticamente la figura anterior, se asignan los valores iniciales a la variables índices izquierda, derecha y pivote, donde izquierda y derecha se utilizan para barrer la lista desde la izquierda a derecha y de derecha a izquierda respectivamente; y pivote almacena el índice del elemento a ordenar para separar la lista en dos sublistas, donde la sublista de la izquierda tendrá los elementos menores al elemento en la posición pivot y en la sublista derecha los elementos mayores. Luego se procede a ubicar los elementos menores al pivot a la izquierda y los mayores a la derecha con un ciclo *mientras* el índice izquierda sea menor que el de la derecha, en donde se usan dos ciclos *mientras*: uno cuando el elemento de la lista en la posición izquierda es menor que el elemento en la posición pivot se incrementa el índice izquierda, el segundo si el elemento de la lista en la posición derecha es mayor que el elemento en la posición pivot se decremente el índice derecha. Cuando estos dos ciclos se detienen, si el índice izquierda es menor que el derecha significa que deben *intercambiar* los valores almacenados en dichos índices. Luego se continua con el ciclo del *mientras* izquierda sea menor que derecha para terminar de reubicar los valores menores y mayores, cuando se terminan de intercambiar estos elementos, se procede a ubicar el elemento pivot a su posición final en la lista (quedando de esta manera ya ordenado). De esta manera queda dividida la lista en dos sublistas parcialmente ordenadas, la de los menores sublista izquierda y la de los mayores sublista derecha. Para terminar se evalúan si las sublistas izquierda y derecha tienen elementos para *ordenar*, si es así, entonces se llama recursivamente a la función *quicksort* actualizando los parámetros primero y último; para la llamada recursiva de la sublista izquierda se actualiza el valor del último por (izquierda-1) y para la sublista derecha actualizamos el valor de primero por (izquierda+1). Esto produce que se ejecute recursivamente la función antes descrita, *particionando* la lista en sublistas hasta que las sublistas ya no puedan ser particionadas y queden ordenadas.

De la misma forma que hemos hecho con los algoritmos anteriores nos resta hacer la tabla comparativa del algoritmo basándonos en los aspectos preestablecidos. Cabe destacar que el algoritmo *quicksort* es uno de los métodos más eficientes de ordenamiento utilizados por distintos lenguajes: como Javascript (cuando llamamos al método *sort* para ordenar un vector), y también es el caso Ruby (que dispone de un método *qsort* para ordenar vectores), en el caso particular de Python (ya que estamos utilizando este lenguaje) no utiliza este método de ordenamiento, en su lugar implementa el método de ordenamiento híbrido llamado *timsort*.

Complejidad temporal	$O(n \log n)$
Complejidad espacial	En el lugar
Estabilidad	Inestable
Interno /Externo	Interno
Recursivo/No recursivo	Recursivo
Ordenamiento por comparación	Comparativo

Mezclando los elementos de la receta en el orden correcto gracias al ordenamiento por mezcla

Este algoritmo fue inventado por John Von Neumann en 1945² es un algoritmo cuyo principio de funcionamiento es del tipo *mezcla*, que se basa en la técnica de divide y vencerás. La esencia del algoritmo es la siguiente, si el tamaño de la lista es uno o menor la lista está ordenada; sino, se divide la lista en dos sublistas de la mitad del tamaño de la lista y se llama recursivamente a ordenar las sublistas usando ordenamiento por *mezcla*. Y por último se *mezclan* las sublistas ordenadas en una sola lista, es decir la idea que persigue es que al ordenar una lista pequeña es más rápido que ordenar una lista grande, y construir una lista ordenada a partir de dos listas ordenadas también es una actividad rápida. A continuación en la figura II se ilustra la mecánica de funcionamiento del algoritmo.

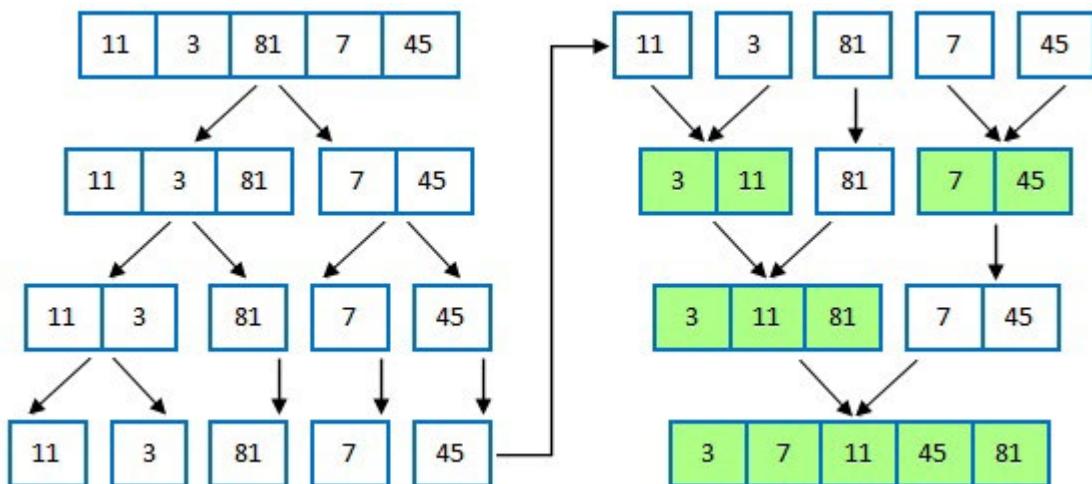


Figura II. Funcionamiento del ordenamiento por mezcla

² Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *Sorting and Searching. The Art of Computer Programming*. 3 (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.

Ahora pasemos al implementación del algoritmo, esta estará compuesta por dos funciones, la primera la podemos ver en la figura 12.

```
def mergesort(lista):
    """Método de ordenamiento mergesort."""
    if (len(lista) <= 1):
        return lista
    else:
        medio = len(lista) // 2
        izquierda = []
        for i in range(0, medio):
            izquierda.append(lista[i])
        derecha = []
        for i in range(medio, len(lista)):
            derecha.append(lista[i])
        izquierda = mergesort(izquierda)
        derecha = mergesort(derecha)
        if(izquierda[medio-1] <= derecha[0]):
            izquierda += derecha
        return izquierda
    resultado = merge(izquierda, derecha)
    return resultado
```

Figura 12. Método de ordenamiento mezcla

Continuemos con el análisis del algoritmo. Primero se chequea si el tamaño de la lista es menor o igual a uno, condición de fin de las llamadas recursivas y devuelve la lista, sino se calcula el valor del índice medio dividiendo la cantidad de elementos de la lista dividido por dos –utilizando división entera– para comenzar a dividir la lista. Se crean las listas izquierda y derecha, luego se utilizan dos ciclos (*para*) para cargar los elementos en dichas listas, un ciclo de cero al valor medio y el otro de medio al tamaño de la lista. Luego se llama recursivamente a ordenamiento *mezcla* con las sublistas izquierda y derecha, asignando los resultados en las mismas variables. Una vez que finalizan las llamadas recursivas de las sublistas, es decir que ya no pueden ser particionadas y cada una de estas se consideran ordenadas comienza el proceso de combinación de dichas sublistas.⁷ Para lo cual se chequea si el último elemento de la sublista izquierda es menor o igual que el primero de la sublista derecha, de ser así se concatena la sublista derecha al final de la izquierda; dada que ambas sublistas están ordenadas y por ende significa que todos los elementos de la sublista derecha son mayores o iguales que los de la sublista izquierda. En el caso contrario, es decir si no se pueden concatenar las listas, es necesario *mezclarlas* para obtener una sola ordenada, para resolver esto llamamos a la función *mezcla* cuya implementación se describe a continuación de la figura 13. Finalmente una vez que hemos combinado todas las sublistas en una, se devuelve la lista ordenada.

```

def merge(izquierda, derecha):
    """Función para mezclar listas."""
    lista_mezclada = []
    while (len(izquierda) > 0) and (len(derecha) > 0):
        if (izquierda[0] < derecha[0]):
            lista_mezclada.append(izquierda.pop(0))
        else:
            lista_mezclada.append(derecha.pop(0))
    if(len(izquierda) > 0):
        lista_mezclada += izquierda
    if(len(derecha) > 0):
        lista_mezclada += derecha
    return lista_mezclada

```

Figura 13. Función para mezclar listas

La segunda función de este algoritmo se encarga de mezclar dos listas que ya se encuentran ordenadas, para lo cual se crea una lista y utilizamos un ciclo *mientras* las dos sublistas tengan al menos un elemento, entonces determinamos que elemento es el menor, el primero de la sublista izquierda o el primero de la sublista derecha, y luego quitamos el menor de estos para agregarlo a lista creada anteriormente. Cuando finaliza el ciclo *mientras* significa que una de las dos sublistas está vacía y la otra aún tiene elementos, entonces determinamos cual es la sublista que aún tiene elementos para luego concatenarlos al final de la lista creada y por último se retorna la lista que contiene las dos sublistas ordenadas.

Finalmente vamos a hacer la tabla comparativa de este algoritmo al igual que hemos hecho con los algoritmos anteriores, previamente mencionamos que Python utiliza el método de ordenamiento *timsort*, este combina ordenamiento por *inserción* y ordenamiento por *mezcla* aprovechando lo mejor de cada uno de estos para lograr un rendimiento aún mejor que el del *quicksort*.

Complejidad temporal	$O(n \log n)$
Complejidad espacial	Fuera de lugar
Estabilidad	Estable
Interno /Externo	Externo
Recursivo/No recursivo	Recursivo
Ordenamiento por comparación	Comparativo

Flotando y hundiendo los elementos mediante ordenamiento por montículo

Es un algoritmo cuyo funcionamiento es del tipo *selección*, se basa en el uso del TDA montículo. Para poder entender y realizar este tipo de ordenamiento es necesario conocer y entender el funcionamiento de dicho TDA, para ello se recomienda que el lector se remita al capítulo XII; en el cual se describe y desarrolla el TDA montículo y después el ordenamiento por montículo. A modo de

resumen se presenta la tabla comparativa de dicho algoritmo, pero todos los detalles se describen en el capítulo antes mencionado.

Complejidad temporal	$O(n \log n)$
Complejidad espacial	En el lugar
Estabilidad	Inestable
Interno /Externo	Interno
Recursivo/No recursivo	No recursivo
Ordenamiento por comparación	Comparativo

Contar ovejas, ¿truco para dormir o ejercicio de ordenamiento de conteo?

Este es un algoritmo que solo sirve para ordenar números enteros positivos –dado que utiliza los valores a ordenar como índices–, fue desarrollado por Harold H. Seward en 1954³, además este algoritmo asume que se conoce de ante mano el rango de valores a ordenar, es decir el valor mínimo y el máximo. Este algoritmo a diferencia de los anteriores no ordena por comparación, sino que utiliza un vector de conteo para determinar la posición de cada elemento en la nueva lista de los elementos ordenados. A continuación en la figura 14 y 15 se presentan los pasos de funcionamiento del algoritmo, en la primera podemos ver como se genera la lista de conteo y en la segunda vemos como se ordenan los elementos.

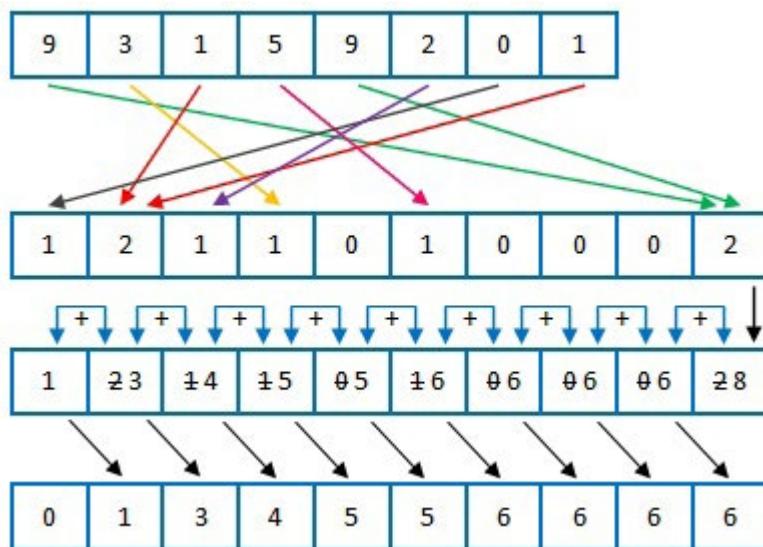


Figura 14. Generación de vector de conteo

³ Seward, H. H. (1954), "2.4.6 Internal Sorting by Floating Digital Sort", Information sorting in the application of electronic digital computers to business operations, Master's thesis, Report R-232, Massachusetts Institute of Technology, Digital Computer Laboratory, pp. 25–28.

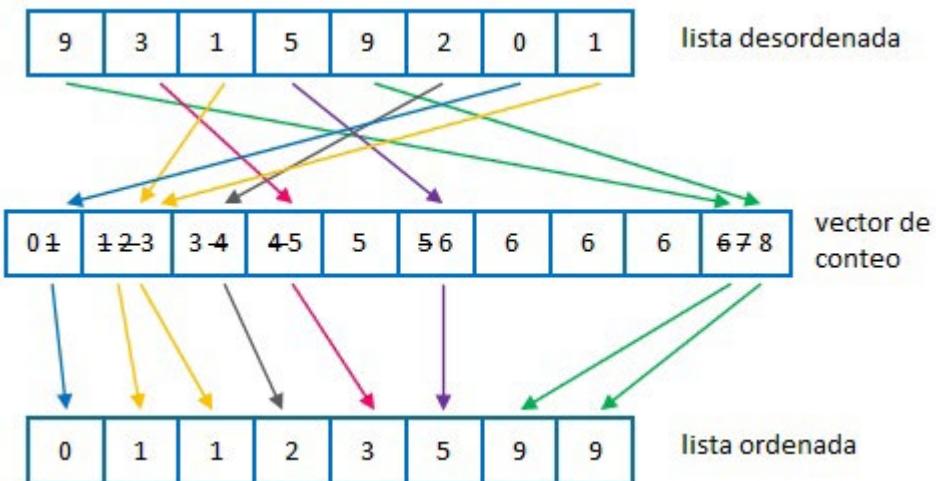


Figura 15. Funcionamiento del algoritmo ordenamiento por conteo

Ahora continuemos con la implementación del algoritmo, este lo describiremos en la figura 16. Nótese que al no ser un algoritmo de ordenamiento por comparación no requiere utilizar una estructura de ciclo de ciclo, lo cual le permite mejorar significativamente su rendimiento respecto de los visto anteriormente.

```
def count_sort(lista, maximo):
    """Método de ordenamiento countsort."""
    lista_conteo = [0] * (maximo + 1)
    lista_ordenada = [None] * len(lista)

    for i in lista:
        lista_conteo[i] += 1

    total = 0
    for i in range(len(lista_conteo)):
        lista_conteo[i], total = total, total + lista_conteo[i]

    for indice in lista:
        lista_ordenada[lista_conteo[indice]] = indice
        lista_conteo[indice] += 1
    return lista_ordenada
```

Figura 16. Método de ordenamiento por conteo

Realicemos ahora una descripción del algoritmo desarrollado en la figura anterior. En primer lugar, creamos el vector de conteo, cuya longitud es el rango entre el valor mínimo y el máximo –por defecto tomaremos como valor mínimo cero, dado que es la posición inicial por defecto de una lista–, y además creamos una lista para almacenar los elementos ordenados de la misma longitud que la lista a ordenar. Luego, completamos la lista de conteo, para esto tomamos cada valor de la lista a ordenar y lo usamos como índice para incrementar en uno el valor almacenado en dicha posición en la lista de conteo. Después, tenemos que realizar una sumatoria y desplazamiento de los elementos

de la lista de conteo, tomamos los elementos de a dos, los sumamos y los almacenamos en la posición del segundo elemento, y repetimos esto para cada par de elementos desde el primer elemento al último de la lista a ordenar. Para finalizar el algoritmo nos falta ubicar los elementos de la lista a ordenar en su lugar correspondiente en la lista ordenada, para lo cual tomamos cada uno de estos elementos y los utilizamos como índice para acceder a la lista de conteo, a partir del valor almacenado en dicha posición determinamos la ubicación de dicho elemento en la lista ordenada y también debemos incrementar el valor almacenado en la lista de conteo por si hay elementos repetidos.

Para terminar con este algoritmo haremos la correspondiente tabla comparativa. Nótese que la notación de orden incluye además del tamaño de la entrada n , el valor k que representa el rango de los valores de la entrada. Aunque solo se considera el uso de k si este es mayor que el número de entrada, en el caso contrario el algoritmo tiende a ser de orden lineal.

Complejidad temporal	$O(n + k)$
Complejidad espacial	Fuera de lugar
Estabilidad	Estable
Interno /Externo	Interno
Recursivo/No recursivo	No recursivo
Ordenamiento por comparación	No comparativo

Para cerrar esta sección de algoritmos de ordenamiento, a continuación en la siguiente tabla, presentamos en detalle la comparación de los tiempos requeridos por los distintos algoritmos de *ordenamiento* descritos anteriormente, frente a distintos tamaños de entrada. Mientras que en la figura 17 se puede observar gráficamente el comportamiento de estos algoritmos cuando varía el tamaño de la entrada. Claramente queda determinado que los algoritmos de orden $O(n \log n)$ son mucho más eficientes que los de orden de $O(n^2)$ y esta eficiencia mejora notablemente a medida que aumenta el tamaño de la entrada.

Tamaño de entrada	Ordenamiento	
	Burbuja - Selección - Inserción $O(n^2)$	Rápido - Mezcla - Montículo $O(n \log n)$
10	100	40
100	10 000	700
1 000	1 000 000	10 000
10 000	100 000 000	140 000
100 000	10 000 000 000	1 700 000
1 000 000	1 000 000 000 000	20 000 000
10 000 000	100 000 000 000 000	240 000 000

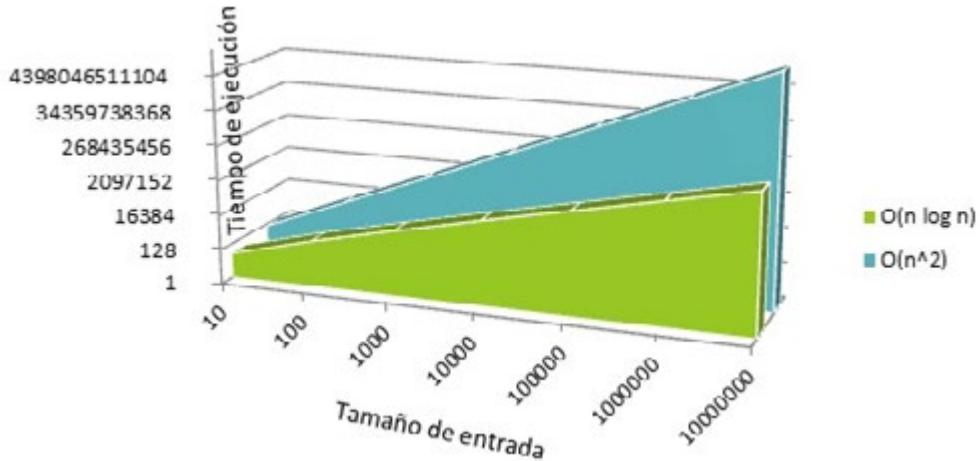


Figura 17. Comparación comportamiento de algoritmos de ordenamiento

Es momento de pasar a estudiar otra operación esencial que podemos hacer sobre un conjunto de datos, independientemente del orden en que estos estén. Siempre es necesario *buscar* un elemento para poder determinar si está presente o no dentro de nuestro conjunto de datos. Por ejemplo, se podría necesitar saber si un determinado elemento está en una estructura de datos o no para determinar qué actividades podrían realizarse: *inserción* en el caso de no estar y *modificación* o *eliminación* en caso de estar, sin importar la técnica o forma que se utilice. A continuación describiremos y analizaremos los algoritmos clásicos de *búsqueda* con su correspondiente implementación en Python. Además realizaremos una comparación de los algoritmos de acuerdo a los siguientes aspectos: su complejidad temporal usando notación O grande, si los datos deben estar ordenados previamente para poder utilizar el método –dado que esto implicaría un incremento en la complejidad–, si es un algoritmo recursivo o no recursivo y finalmente si buscamos un elemento que esta repetido en la estructura, cuál de estos encontraremos o nos devolverá el algoritmo. Estos métodos al igual que los de ordenamiento suelen aplicarse sobre vectores, en nuestro caso particular utilizaremos listas como mencionamos anteriormente.

Uno a uno revisamos cada elemento buscando a la fuerza aplicando búsqueda secuencial

Este es el algoritmo de *búsqueda* más simple, también llamado “búsqueda por fuerza bruta”, como su nombre lo indica compara secuencialmente el elemento buscado con cada elemento de la lista. El elemento buscado puede existir o no dentro de la lista y la única forma de poder determinarlo es recorrer la lista completa. A continuación en la figura 18 se puede ver la implementación del algoritmo.

```
def secuencial(lista, buscado):
    """Método de búsqueda secuencial."""
    posicion = -1
    for i in range(0, len(lista)):
        if(lista[i] == buscado):
            posicion = i
    return posicion
```

Figura 18. Método de búsqueda secuencial con centinela

Realicemos una breve descripción del algoritmo, en primer lugar asignamos a la variable posición el valor -1, esto indica que el elemento buscado no ha sido encontrado. Luego utilizamos un ciclo para de o a la cantidad de elementos de la lista, para recorrer secuencialmente todos los elementos de la lista, y chequeamos si el elemento buscado es igual al elemento de la lista en la posición de la variable de control (lista[i]). Si la comparación es verdadera significa que se encontró el elemento buscado y asignamos a la variable posición el valor de la variable *i*, es decir la posición donde fue encontrado. Si al terminar el ciclo la variable posición aún tiene valor -1 significa que el elemento buscado no se encuentra dentro de la lista, y finalmente devolvemos la variable posición.

A este algoritmo podemos agregarle una sutil mejora, esta consiste en detener el ciclo de recorrido de la lista cuando se encuentra el elemento buscado. Por ejemplo si la lista tiene 10 000 elementos y el elemento buscado está en la posición 50, el ciclo debería detenerse para no recorrer las restantes 9 950 elementos de la lista, dado que no tiene sentido realizarlo porque el elemento buscado ya fue encontrado y estaríamos desperdiciando tiempo de procesamiento. Este algoritmo “mejorado” lo llamaremos “búsqueda secuencial con centinela” y se observa en la figura 19.

```
def centinela(lista, buscado):
    """Método de búsqueda secuencial con centinela."""
    posicion = -1
    for i in range(0, len(lista)):
        if(lista[i] == buscado):
            posicion = i
            break
    return posicion
```

Figura 19. Método de búsqueda secuencial con centinela usando break

Este algoritmo funciona exactamente igual que el anteriormente, solo que agregamos la sentencia *break* cuando se encuentra el elemento buscado, entonces rompe el ciclo y devuelve el valor de posición. Existen autores que no utilizan la sentencia *break* dentro de sus algoritmos por criterios particulares, una de las principales objeciones es que no les parece lógico “romper” el funcionamiento de un ciclo. En ese caso podemos utilizar un ciclo *mientras* en lugar de un *para* evitando utilizar la sentencia *break* como se aprecia a continuación en la figura 20.

```
def centinela(lista, buscado):
    """Método de búsqueda secuencial con centinela."""
    posicion = -1
    i = 0
    while (i < len(lista)) and (posicion == -1):
        if(lista[i] == buscado):
            posicion = i
        i += 1
    return posicion
```

Figura 20. Método de búsqueda secuencial con centinela sin usar break

Nótese que solo reemplazamos el ciclo *para* por un *mientras* con una condición extra, que la variable posición sea igual a -1, esto significa que el elemento buscado aún no ha sido encontrado. Cuando posición sea distinto de -1 se detendrá el ciclo *mientras* logrando el funcionamiento buscado sin utilizar la sentencia *break*. De todas maneras esta modificación del algoritmo solo mejora su rendimiento en ciertos casos, por lo cual la complejidad será la misma.

Para terminar con este algoritmo vamos a realizar la tabla comparativa en base a los aspectos establecidos previamente.

Complejidad temporal	$O(n)$
Datos ordenados/desordenados	Desordenados
Recursivo/No recursivo	No recursivo
Elemento returned si hay repetidos	*Primero con centinela *Último sin centinela

Particionando nuestra lista de elementos una vez tras otra con búsqueda binaria

Este algoritmo también llamado “búsqueda logarítmica” es muy eficiente en cuanto al número de comparaciones requeridas para determinar si un elemento existe o no dentro de una lista de elementos, pero tiene un requisito no excluyente para que el mismo **funcione, la lista de elementos debe estar ordenada**. Su principio de funcionamiento se basa en determinar si el elemento buscado se encuentra en el medio de la lista dividiendo la lista en dos sublistas, la izquierda y la derecha. Si el elemento buscado no está en el medio, dependiendo si este es menor o mayor, se repite el mismo procedimiento con la sublista izquierda o la sublista derecha respectivamente descartando la otra. Este procedimiento se repite mientras las sublistas puedan ser particionadas o se encuentre el elemento buscado descartando una sublista en cada partición, esto hace que este algoritmo sea muy eficiente. A continuación en la figura 21 se presenta el esquema de funcionamiento y en la figura 22 podemos ver la implementación del algoritmo.

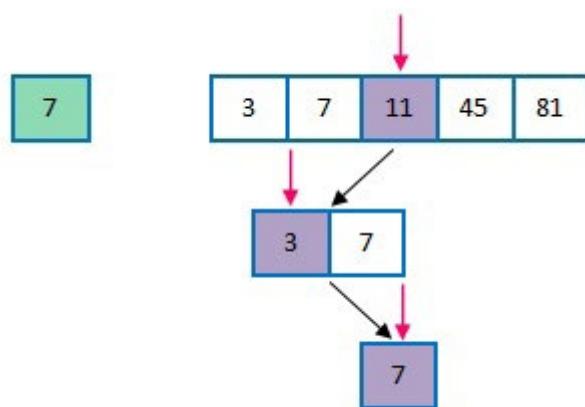


Figura 21. Esquema de funcionamiento de búsqueda binaria

```

def binaria(lista, buscado):
    """Método de búsqueda binaria."""
    posicion = -1
    primero = 0
    ultimo = len(lista)-1
    while (primero <= ultimo) and (posicion == -1):
        medio = (primero + ultimo) // 2
        if(lista[medio] == buscado):
            posicion = medio
        else:
            if(buscado < lista[medio]):
                ultimo = medio-1
            else:
                primero = medio+1
    return posicion

```

Figura 22. Método de búsqueda binaria

Es momento de desglosar analíticamente este algoritmo, primero le asignamos a la variable posición el valor `-1`, para indicar que el elemento buscado no ha sido encontrado y además se asignan los valores iniciales a las variables índice `primero` y `último`, cero y número de elementos de la lista `-1` respectivamente. Luego utilizamos un ciclo `mientras`, el índice `primero` sea menor o igual que `último` (es decir aún se puede *particionar* o *buscar* en las sublistas) y todavía no se haya encontrado el elemento buscado, se calcula el valor del índice medio de la lista, sumando los valores de `primero` y `último`, y luego se los divide por dos, utilizando división entera. Después chequeamos si el elemento en el medio de la lista es igual al buscado, de ser así a la variable `posición` se le asigna el valor de `medio`, sino se procede a *particionar* la lista actualizando los índices `primero` o `último`, según corresponda, y descartando el resto de la lista. Si el elemento buscado es menor que el valor de la lista en la posición medio se actualiza el índice `último` asignándole el valor `medio - 1`, sino se actualiza el valor del índice `primero` asignándole el valor `medio + 1`. Y por último devolvemos el valor de la variable `posición` para poder determinar si el elemento buscado fue encontrado o no.

Ahora solo nos resta presentar la tabla comparativa de este algoritmo de la misma manera que lo hemos hecho anteriormente:

Complejidad temporal	$O(\log n)$
Datos ordenados/desordenados	Si o si ordenados
Recursivo/No recursivo	No recursivo (por defecto) *puede ser recursivo
Elemento retornado si hay repetidos	Se desconoce

Finalmente para hacer un cierre de la sección de búsqueda, a continuación se presenta la comparación de los tiempos requeridos por los distintos algoritmos de *búsqueda* descritos anteriormente, frente a distintos tamaños de entrada, mientras que en la figura 23 se puede observar gráficamente el comportamiento de estos algoritmos cuando varía el tamaño de la entrada. Claramente queda

determinado que los algoritmos del orden de $O(\log n)$ son mucho más eficientes que los del orden de $O(n)$ y esta eficiencia mejora significativamente a medida que aumenta el tamaño de la entrada.

Tamaño de entrada	Búsqueda	
	Secuencial $O(n)$	Binaria $O(\log n)$
10	10	4
100	100	7
1 000	1 000	10
10 000	10 000	14
100 000	100 000	17
1 000 000	1 000 000	20
10 000 000	10 000 000	24
100 000 000	100 000 000	27
1 000 000 000	1 000 000 000	30

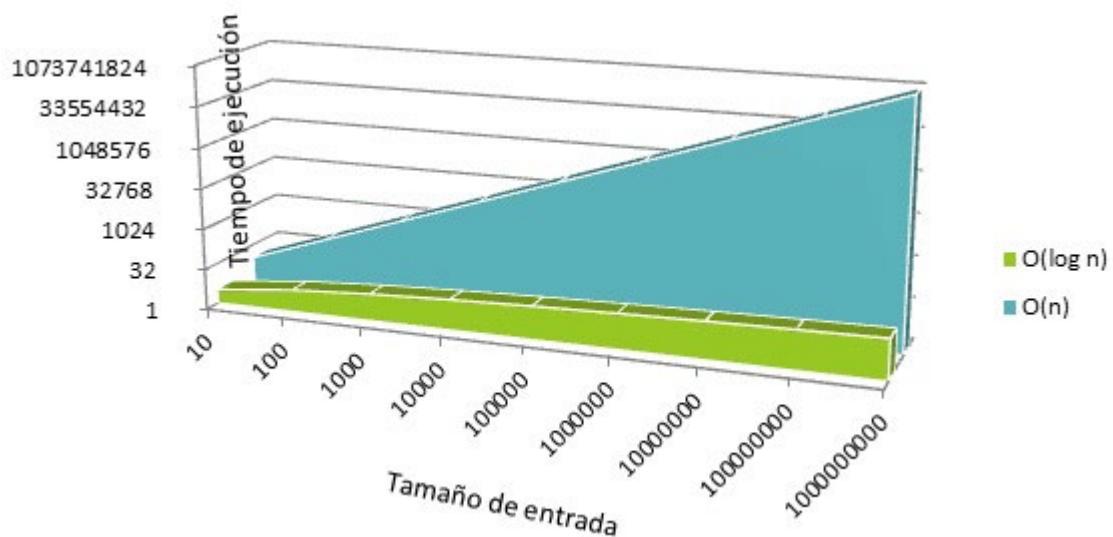


Figura 23. Comparación comportamiento de algoritmos de búsqueda

Guía de ejercicios prácticos

A continuación se plantean una serie de actividades que nos permitirán comparar el rendimiento y eficiencia de los algoritmos de *ordenamiento* y *búsqueda*.

1. Desarrollar el algoritmo de ordenamiento por casilleros (*bucket sort*). ✓
2. Desarrollar el algoritmo de ordenamiento Shell (*shell sort*). Tiene este nombre debido a su inventor Donald Shell. ✓
3. Desarrollar el algoritmo de ordenamiento Radix (*radix sort*). ✓
4. Explicar brevemente el funcionamiento del algoritmo de ordenamiento *Timsort* y su orden de complejidad, –el cual fue desarrollado por Tim Peters en lenguaje Python– este algoritmo es de tipo *híbrido* utilizado de forma nativa para ordenar los elementos de una colección en Python. ✓
5. Generar una lista de elementos aleatorios de distintos tamaños (100 000, 1 000 000, 10 000 000), para probar los distintos algoritmos de *ordenamiento* vistos. Además agregar las instrucciones necesarias para medir su tiempo de ejecución y poder compararlos. ✓
6. Generar también otra lista de elementos aleatorios de distintos tamaños (100 000, 1 000 000, 10 000 000), para probar los distintos algoritmos de *búsqueda* vistos. Además agregar las instrucciones necesarias para medir su tiempo de ejecución para poder compararlos. ✓
7. Dada una lista de personajes de la saga de Star Wars de las que se conoce su nombre, resolver las siguientes tareas:
 - a. listar los personajes ordenados alfabéticamente de manera ascendente;
 - b. determinar si el personaje Darth Maul está cargado y en qué posición se encuentra;
 - c. mostrar la información de los personajes que se encuentran antes y después de Hera Syndulla;
 - d. listar todos los personajes que comienzan con la letra *L*;
 - e. utilizar los métodos de ordenamiento y búsqueda más eficiente de acuerdo al tipo de dato.
8. Se dispone de la lista de superhéroes y villanos de la saga de Marvel Cinematic Universe (MCU) de los que contamos con la información de nombre del personaje y año de la primera película en la que apareció; a partir de estos resolver las siguientes actividades:
 - a. realizar un listado ascendente de los personajes ordenados por su nombre;
 - b. indicar quien fue el primer y el último personaje en aparecer en una película sin realizar un recorrido de la lista (podrían ser más de uno tanto el primero como el último);
 - c. realizar un listado de los personajes ordenados de manera descendente por año de aparición;

- d. calcular el rango de años entre el primer personaje en aparecer y el último;
 - e. determinar si los personajes Capitan America y Rocket Raccoon están en la lista y en qué año aparecieron.
9. Se cuenta con una lista de Pokémons, de cada uno de estos se sabe su nombre, número y tipo (solo considerar uno el principal), con los cuales deberá resolver las siguientes tareas:
- a. mostrar un listado de los Pokémons ordenados por números usando el método de ordenamiento por conteo;
 - b. realizar un listado ordenado por nombre utilizando el método de ordenamiento rápido;
 - c. mostrar toda la información de pokémon número 640;
 - d. listar todos los Pokémons que comienzan con la letra T ;
 - e. determinar si existe el Pokémon Cobalion y mostrar toda su información;
 - f. realizar un listado de todos los Pokémon de tipo eléctrico y calcular cuántos son.
10. Un vendedor de “Fuko Pop”, tiene la lista de todos los funko que tiene disponibles en su tienda con la siguiente información: número, nombre, colección y precio; para lo cual nos solicita le desarrollemos un algoritmo que contemple los siguiente requerimientos:
- a. realizar un listado ordenado por número;
 - b. realizar un listado de los funko pop de una colección;
 - c. determinar si tiene disponible el funko pop 130 de la colección de Star Wars y mostrar toda la información;
 - d. mostrar todos los funko pop disponible número 295;
 - e. listar todos los modelos de Darth Vader y Capitana Marvel;
 - f. determinar si existen funko pop de Red Skull, Thanos y Galactus, además mostrar la información de estos;
 - g. calcular el costo de todos los modelos de Tony Stark e Iron Man disponibles;
 - h. calcular el promedio de costo de todos los funko pop y el costo total de las colecciones de Rocks y Harry Potter.
- II. A partir de una lista con todos los caballeros Jedi y los lores Sith, de los que conocemos su nombre y el de su maestro (considere solo uno), resuelva las siguientes actividades:
- a. realizar un listado ordenado por nombre;

- b. listar todos los Jedi ordenados por nombre;
 - c. listar todos los Sith ordenados por nombre;
 - d. mostrar los aprendices de Palpatine y de Obi-Wan kenobi, además contar cuantos aprendices tuvo cada uno;
 - e. determinar si Dath Malak está en la lista y mostrar su información;
 - f. mostrar la posición en la que se encuentra Yoda.
12. Dada una lista de dioses griegos a partir de la cual debemos desarrollar las funciones necesarias para resolver los siguientes ítems:
- a. emitir un listado de todos los dioses ordenados;
 - b. determinar si Atenea está en la lista;
 - c. indicar en qué posición se encuentra Deméter;
 - d. listar todos los dioses que comienzan con la letra *H* y además determinar cuántos son;
 - e. agregar al dios Helios y volver a listar los dioses ordenados alfabéticamente.
13. Se dispone de una lista de películas con la siguiente información: título, año de estreno, recaudación y valoración del público (de 1 a 5), los cuales debemos procesar contemplando las siguientes tareas:
- a. realizar un listado ordenado por título, año de lanzamiento y por recaudación este último de manera descendente;
 - b. mostrar toda la información de la película que mas recaudo;
 - c. listar todas las películas que tenga valoración 5;
 - d. determinar si la película “Avengers: Infinity War” está en la lista y mostrar toda su información;
 - e. indicar en qué posición se encuentra la película “Star Wars: The Return of Jedi”;
 - f. calcular el total recaudado por las películas que en su título incluyen la palabra “Avengers”;
 - g. mostrar todas las películas que se estrenaron en el año 2017;
 - h. listar todas las películas que comienzan con la palabra “Iron”.
14. La empresa Netflix nos pone a nuestra disposición los datos de todas las series cargadas en esta plataforma con los siguientes datos: nombre, cantidad de temporadas y cantidad de capítulos

totales de la serie. Los cuales debemos procesar desarrollando las funciones necesarias para dar solución a las siguientes ítems:

- a. listar las series ordenadas a criterio a los siguientes criterios: por nombre, por cantidad de temporadas, por cantidad de capítulos;
- b. mostrar toda la información de la serie “Los 100”;
- c. determinar cuál es la serie con mayor cantidad de temporadas y mayor cantidad de capítulos;
- d. calcular cuantas series dispone la plataforma y el promedio de temporadas;
- e. determinar el promedio de capítulos por temporada de la serie “Star Wars: Rebels”;
- f. listar el TOP 5 de series con mayor cantidad de capítulos y el top 10 de las series con mayor cantidad de temporadas;
- g. mostrar todas las series con tres o menos temporadas.

15. Se cuenta con una lista de canciones, de cada una de estas conocemos su nombre, nombre de la banda o artista, y el año de lanzamiento del álbum; desarrollar las funciones necesarias para dar solución a las siguientes tareas:

- a. realizar un listado ordenado por canción, por banda o artista y por año de lanzamiento, utilizando el método que sea más óptimo para cada tipo de dato;
- b. determinar si en la lista existe alguna canción de Audioslave y Rolling Stone;
- c. mostrar todas las canciones de Nirvana;
- d. agregar una nueva canción a la lista, y volver a realizar un listado ordenado por nombre de canción;
- e. determinar cuantas canciones de los Red Hot Chili Peppers hay en la lista;
- f. mostrar toda la información de las canciones “Fake tales of San Francisco” y “Black hole sun”.

16. Dada una lista de las naves (y vehículos) de Star Wars –consideraremos a todos como naves– de las que conocemos su nombre, largo, tripulación y cantidad de pasajeros, desarrollar los algoritmos necesarios para resolver las actividades detalladas a continuación:

- a. realizar un listado ordenado por nombre de las naves de manera ascendente y por largo de las mismas de manera descendente;
- b. mostrar toda la información del “Halcón Milenario” y la “Estrella de la Muerte”;
- c. determinar cuáles son las cinco naves con mayor cantidad de pasajeros;

- d. indicar cuál es la nave que requiere mayor cantidad de tripulación;
- e. mostrar todas las naves que comienzan con *AT*;
- f. listar todas las naves que pueden llevar seis o más pasajeros;
- g. mostrar toda la información de la nave más pequeña y la más grande.

CAPÍTULO V

Representando modelos reales con tipos de datos abstractos

Una estructura de datos es una colección de elementos, como ya mencionamos previamente, pero además en muchas ocasiones estas representan un modelo de la vida real. Más allá de la manera en que se almacenan los elementos –independientemente de si se utilizan estructuras estáticas o dinámicas– el aspecto más importante son las actividades que la misma permite realizar sobre los elementos. Las actividades mínimas que normalmente se realizan sobre cualquier estructura de datos son *insertar*, *eliminar*, *buscar* y hacer un *listado* o *barrido* de sus elementos. Estas estructuras de datos pueden ser lineales y no lineales –o ramificadas–, dependiendo de su tipo las operaciones mencionadas pueden cambiar su orden de complejidad.

Las estructuras de datos están presentes en la vida cotidiana de las personas de manera implícita, de hecho es común interactuar con estas diariamente. Por ejemplo cuando se dobla la ropa se la suele apilar, en el supermercado se debe hacer cola para poder pagar, cuando se realiza una actividad muchas veces se hace una lista de tareas que conforman la misma, etcétera. Al momento de representar el modelo real de estas estructuras en un modelo virtual es necesario extraer de estas sus principales características, que permitan representarla virtualmente, y su mecánica de funcionamiento –es decir que actividades se pueden realizar–.

Para poder efectuar esta abstracción de un modelo real a uno virtual se utiliza una técnica denominada tipo de dato abstracto (TDA), que permite definir atributos que describen las características deseadas del modelo real y funciones que representan las actividades –o comportamiento– que se pueden realizar con el mismo. Con esta técnica se logra una representación parcial del modelo real, suficiente para poder representar dicho modelo y usarlo de manera virtual. A lo largo de este libro se utiliza el TDA para representar las distintas estructuras de datos clásicas de las ciencias de la computación: pila, cola, lista, tabla de dispersión, árbol, montículo, grafo; cada una de estas estructuras se desarrollan en detalle en distintos capítulos de este libro. Existen dos alternativas para realizar estos TDA, utilizando variables estáticas –mediante el uso de vectores o matrices– o variables dinámicas –recurriendo al uso de punteros y variables dinámicas–. Cabe destacar que dependiendo del modelo a representar con una simple estructura estática es más que suficiente y no es necesario recurrir a estructuras dinámicas.

Para poder definir la estructura interna de un TDA y sus funciones asociadas previamente es necesario que se definan algunos conceptos básicos:

- I. **Variable dinámica:** una variable dinámica es aquella que se puede crear y destruir en tiempo de ejecución a diferencia de una variable estática que lo hace en tiempo de compilación. Este tipo de variables son administradas mediante el uso de punteros.

2. Puntero: el puntero es un tipo de dato que se utiliza para almacenar una dirección de memoria de una variable dinámica –es decir se podría decir que un puntero apunta a una variable dinámica– esto también se conoce como dirección de referencia.
3. Nodo: es un tipo de datos que se define mediante el uso de una clase –dado que en Python no existen los registros– para representar a cada elemento de nuestra estructura de datos. Dependiendo de esta pueden variar sus atributos, pero en su esencia un nodo debe tener mínimamente un campo donde se almacene la información y otro que lo enlace con el siguiente o anterior nodo de la estructura.
4. Variables huérfanas: cuando una variable dinámica deja de ser apuntada por un puntero, quedará en memoria pero no podrá accederse a ella porque su dirección se perdió. En Python cuando ocurre esto el intérprete elimina automáticamente dicha variable para liberar espacio en memoria. Por lo que para eliminar una variable dinámica solo basta con cambiar la dirección del puntero que la señala.

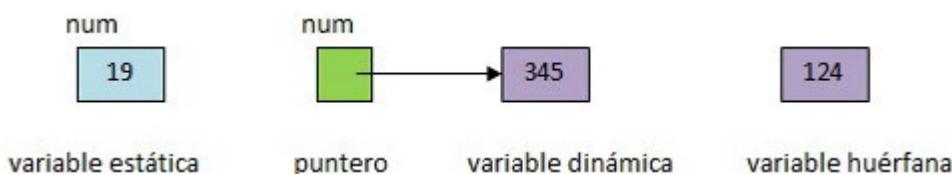


Figura 1. Variable estática, dinámica y huérfana

En la figura 2 se puede observar el funcionamiento asociado entre un puntero y una variable dinámica. Cabe hacer la aclaración de que en Python todas las variables son de tipo puntero a excepción de las de tipo primitivo (booleanas, enteras, reales, complejas y cadenas de textos “String”). En el primer caso la respuesta será (valor num1 1 valor num2 4) claramente se observa que son tratadas como variables estáticas y el cambio en una no afecta a la otra. En cambio para el segundo caso (valor lista1 [1, 2, 3] valor lista2 [1, 2, 3, 87]) las variables son punteros que apuntan a la misma variable dinámica, si se realiza el cambio con una de las variables que apunta a la variable dinámica. Al acceder con la otra se refleja el cambio porque en efecto apunta a la misma variable, –codifique el ejemplo para poder probarlo y ver los resultados–.

```
# variables estaticas
num1 = 1
num2 = num1
num2 = num2 + 3
print("valor num1", num1, "valor num2", num2)

# variables dinamicas
lista1 = [1, 2, 3]
lista2 = lista1
lista2[1] = 7
lista2.append(87)
print("valor lista1", lista1, "valor lista2", lista2)
```

Figura 2. Variables estáticas y dinámicas

Una ventaja de trabajar con punteros es que almacenan una dirección de memoria por lo que se puede tener más de un puntero que apunten a una misma variable dinámica. Las operaciones útiles de comparaciones que se puede hacer entre punteros son dos, si ambas variables son iguales –es decir si están apuntando a la misma variable dinámica– o si los punteros son distintos –si no están apuntando a la misma variable dinámica–. Esto se puede ver en la figura 3, la salida del programa será para el primer caso “Apuntan a diferentes variable dinámica” y “Apuntan a la misma variable dinámica” para el segundo caso. El único valor conocido que el programador le puede asignar a un puntero es vacío o *None* (en el caso de Python) –dado que no es posible conocer las direcciones de memoria–. Esto significa que el puntero no apunta a ninguna variable, también se puede asignar una dirección de memoria que este almacenada en otro puntero. Las demás direcciones son asignadas automáticamente por el sistema operativo cuando creamos las variables dinámicas.

```
puntero1 = [0, 1]
puntero2 = [0, 1]

if(puntero1 == puntero2):
    print('Apuntan a la misma variable dinámica')
else:
    print('Apuntan a diferentes variable dinámica')

puntero2 = {}
puntero1 = puntero2

if(puntero1 == puntero2):
    print('Apuntan a la misma variable dinámica')
else:
    print('Apuntan a diferentes variable dinámica')
```

Figura 3. Comparación de punteros

Un nodo es un tipo de datos que se define con una clase como se observa en la figura 4, con el cual se representan cada uno de los elementos de nuestra colección o estructura de datos. A diferencia de las estructuras estáticas en las que se accede a los elementos mediante índices o posiciones como en el caso de los vectores o las matrices. Las estructuras dinámicas son un conjunto de nodos enlazados entre sí –mediante el uso de punteros–, de manera tal que a partir de un determinado nodo se puede acceder al anterior, al siguiente o a ambos, dependiendo de la estructura del nodo definida y de la naturaleza de la estructura que se quiera modelar.

```
class Nodo(object):
    """Clase nodo simplemente enlazado."""

    info, sig = None, None
```

Figura 4. Nodo simplemente enlazado

Una característica importante que diferencia a las estructuras dinámicas de las estáticas es que por su naturaleza dinámica pueden incorporar o eliminar elementos en tiempo de ejecución –dado que utilizan variables dinámicas– esto le permite optimizar el espacio en memoria utilizando solo lo necesario. En cambio, las segundas definen su cantidad de elementos en tiempo de compilación lo que genera dos problemas: determinar el tamaño óptimo de dicha estructura (quedando casi siempre espacio vacíos en el vector ocupando memoria) y la incapacidad de agregar o eliminar posiciones cuando se lo requiera.

Un nodo simplemente enlazado básicamente está compuesto por dos campos, como vimos previamente, información (o dato) y siguiente (o enlace). Aunque puede tener más si la estructura a modelar lo requiere. El campo información puede ser un tipo de dato simple o una clase que agrupe un conjunto de atributos y el campo siguiente es un puntero que almacenará la dirección del siguiente o anterior elemento de la estructura (nodo), como se puede observar en la figura 5.

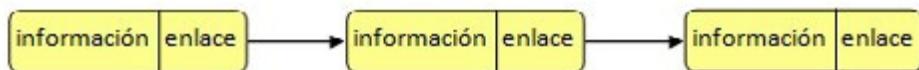


Figura 5. Nodos enlazados

Este tipo particular de nodo también se lo denomina “tipo de dato recursivo”, dado que si se analiza en detalle su definición se puede observar que un nodo está compuesto de dos campos; información donde se almacenan los datos y el siguiente que es un puntero que apunta a un nodo del mismo tipo. El nodo simplemente enlazado es un tipo de dato, que por su definición representa el modelo recursivo que ya se analizó antes. En el que la estructura tiene el campo enlace que representa las “llamadas recursivas” y en la cual la “condición de fin” es cuando el enlace siguiente es vacío o *None*, es decir es el último elemento de la lista.

A continuación en la figura 6 se presenta un ejemplo en el cual se utiliza el tipo de dato nodo simplemente enlazado definido anteriormente, y como se implementa con este la carga de un conjunto de palabras que luego se listan. Vale destacar que los nodos son creados a medida que se necesitan para cargar una palabra y que solo es necesaria una variable de tipo puntero para poder realizar un *barrido* de todos los nodos –dado que a partir de campo siguiente se puede pasar de un nodo a otro–.

```

aux = Nodo()
aux.info = "Primer nodo"
palabra = input('Ingrese una palabra: ')
naux = aux
while (palabra != ""):
    nodo = Nodo()
    nodo.info = palabra
    naux.sig = nodo
    naux = nodo
    palabra = input('Ingrese una palabra: ')

while (aux is not None):
    print(aux.info)
    aux = aux.sig
  
```

Figura 6. Manejo de nodos enlazados

Ahora hay que aplicar los conceptos vistos para crear un TDA, –se desarrollará el ejemplo del TDA polinomio–, para ver en detalle cómo se define una estructura de datos y sus funciones asociadas con un ejemplo sencillo.

Un TDA está compuesto de dos bloques básicos, estructura o definición de tipos e interfaz o comportamiento. En el primero de estos bloques se hace la definición formal de la estructura es decir las características que se utilizan para representar el modelo real de manera virtual. Mientras que en el segundo se define mediante funciones el comportamiento de dicho modelo; estas funciones serán los eventos mediante los cuales se debe manejar su funcionamiento.

En el bloque estructura además del tipo de dato nodo definido previamente, se deben definir los tipos de datos datoPolinomio y Polinomio, donde el primero almacena el valor y el término de dicho valor –es decir si es x^n , x^{n-1}, \dots, x^0 – y el segundo es la estructura de datos polinomio que está compuesto de dos campos el grado del polinomio y un puntero que apunta al término mayor del polinomio, como se puede observar en la figura 7.

```
class datoPolinomio(object):
    """Clase dato polinomio."""

    def __init__(self, valor, termino):
        """Crea un dato polinomio con valor y termino."""
        self.valor = valor
        self.termino = termino


class Polinomio(object):
    """Clase polinomio."""

    def __init__(self):
        """Crea un polinomio del grado cero."""
        self.termino_mayor = None
        self.grado = -1
```

Figura 7. Definición de tipos de datos

Por su parte en el bloque de la interfaz se desarrollan inicialmente los eventos necesarios para administrar el polinomio, es decir, para poder cargar un elemento, modificarlo y obtener su valor. Se detalla sus correspondientes implementaciones en las figuras 8 y 9.

```
def agregar_termino(polinomio, termino, valor):
    """Agrega un termino y su valor al polinomio."""
    aux = Nodo()
    dato = datoPolinomio(valor, termino)
    aux.info = dato
    if(termino > polinomio.grado):
        aux.sig = polinomio.termino_mayor
        polinomio.termino_mayor = aux
        polinomio.grado = termino
    else:
        actual = polinomio.termino_mayor
        while(actual.sig is not None and termino < actual.sig.info.termino):
            actual = actual.sig
        aux.sig = actual.sig
        actual.sig = aux
```

Figura 8. Eventos para administrar el polinomio parte 1

```
def modificar_termino(polinomio, termino, valor):
    """Modifica un termino del polinomio."""
    aux = polinomio.termino_mayor
    while(aux is not None and aux.info.termino != termino):
        aux = aux.sig
    aux.info.valor = valor

def obtener_valor(polinomio, termino):
    """Devuelve el valor de un termino del polinomio."""
    aux = polinomio.termino_mayor
    while(aux is not None and aux.info.termino > termino):
        aux = aux.sig
    if(aux is not None and aux.info.termino == termino):
        return aux.info.valor
    else:
        return 0
```

Figura 9. Eventos para administrar el polinomio parte 2

Luego se definen las funciones específicas para realizar operaciones con polinomios, como sumar, multiplicar y mostrar su contenido. Su código se puede ver en la figuras IO y II.

```
def mostrar(polinomio):
    """Muestra el polinomio."""
    aux = polinomio.termino_mayor
    pol = ''
    if (aux is not None):
        while(aux is not None):
            signo = ''
            if(aux.info.valor >= 0):
                signo += '+'
            pol += signo + str(aux.info.valor) + "x^" + str(aux.info.termino)
            aux = aux.sig
    return pol

def sumar(polinomio1, polinomio2):
    """Suma dos polinomios y devuelve el resultado."""
    paux = Polinomio()
    mayor = polinomio1 if (polinomio1.grado > polinomio2.grado) else polinomio2
    for i in range(0, mayor.grado+1):
        total = obtener_valor(polinomio1, i) + obtener_valor(polinomio2, i)
        if(total != 0):
            agregar_termino(paux, i, total)
    return paux
```

Figura IO. Eventos de operación sobre polinomios parte I

```
def multiplicar(polinomio1, polinomio2):
    """Multiplica dos polinomios y devuelve el resultado."""
    paux = Polinomio()
    pol1 = polinomio1.termino_mayor
    while(pol1 is not None):
        pol2 = polinomio2.termino_mayor
        while(pol2 is not None):
            termino = pol1.info.termino + pol2.info.termino
            valor = pol1.info.valor * pol2.info.valor
            if(obtener_valor(paux, termino) != 0):
                valor += obtener_valor(paux, termino)
                modificar_termino(paux, termino, valor)
            else:
                agregar_termino(paux, termino, valor)
        pol2 = pol2.sig
    pol1 = pol1.sig
return paux
```

Figura II. Eventos de operación sobre polinomios parte 2

Ahora quedará a cargo del lector completar la funcionalidad del TDA polinomio, dado que solo se desarrollaron algunas funciones, agregándole la capacidad de eliminar términos, y de determinar si en un polinomio existe un término, para evitar tener que llamar a la función “obtener_valor” y luego consultar si el resultado es distinto de cero para determinar si el polinomio tiene ese término o no. Esta última debe ser una función booleana.

Guía de ejercicios prácticos

A continuación se plantean una serie de ejercicios. Para resolverlos primero se deberá desarrollar los TDA, para luego dar solución a los problemas planteados.

1. implementar un TDA que le permita realizar el manejo de archivos. Debe contemplar las siguientes actividades:
 - a. abrir(ruta);
 - b. cerrar(archivo);
 - c. leer(archivo, posición);
 - d. guardar(archivo, dato);
 - e. modificar(archivo, posición, dato);
 - f. utilice la librería *shelve*.
2. Desarrollar un TDA que permita manejar el calendario del mes actual, sobre el que se puedan resolver las siguientes actividades:
 - a. obtener el día actual (número y día de la semana);
 - b. obtener los días feriados;
 - c. obtener los días en que inicia luna llena, luna nueva, cuarto menguante y cuarto creciente;
 - d. Obtener los días y nivel de pesca del mes (calendario de pesca).
3. Implementar sobre el TDA polinomio desarrollado previamente las siguientes actividades:
 - a. restar;
 - b. dividir;
 - c. eliminar un término;
 - d. determinar si un término existe en un polinomio.

4. Implementar el TDA matriz que permita resolver las siguientes actividades:
- a. crear una matriz de $[n \times n]$;
 - b. cargar una matriz de manera aleatoria;
 - c. sumar matrices;
 - d. restar matrices;
 - e. multiplicar matrices.

Desde la cima de la pila, intentando evitar el desbordamiento

Esta estructura de datos es una de las más simples, pero a su vez una de las más importantes dentro de las ciencias de la computación. Es utilizada por diversas técnicas, algoritmos y estructuras más avanzadas. Una pila es una colección de elementos que se agregan y se eliminan siguiendo el principio de último en entrar-primer en salir (LIFO, *Last In-First Out*), es decir, el último elemento insertado en la pila es el primero en ser eliminado. Un elemento puede ser agregado en cualquier momento a una pila, pero solo se puede acceder o eliminar el elemento que esté en la cima o tope de la misma, como se observa en la figura 1.

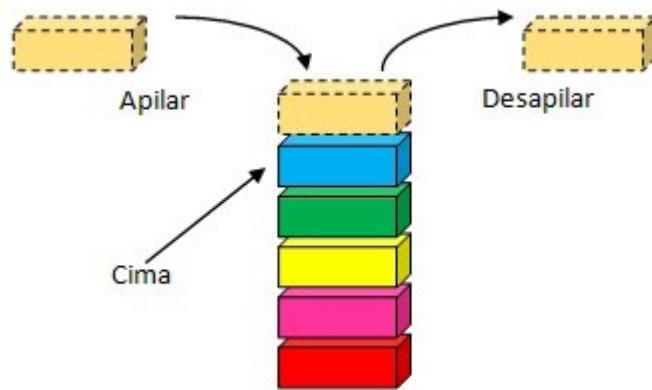


Figura 1. Funcionamiento de una pila

Como se observa en la figura 1, existen dos actividades o eventos que administran los elementos de una pila, esto también implica que en realidad el valor del dato almacenado no influye en el funcionamiento de la estructura –por lo que se podría decir que en realidad la relevancia del valor de los datos es secundaria para la estructura, pero sí son importantes para el sistema que los necesita almacenar-. Estos eventos son: apilar cuando se agrega un nuevo elemento en la cima y desapilar cuando se saca el elemento que está en la cima.

Esto implica que solo se puede acceder al elemento que está en la cima de la pila. La forma de acceder a los demás elementos de la colección es desapilar cada uno de estos, de a uno a la vez.

Algunos ejemplos donde implícitamente interactuamos con pilas en la vida cotidiana: los productos en una góndola, libros sobre una mesa, la ropa dobrada, cajas en un depósito, platos o vasos en una alacena, etcétera. En cualquiera de estos casos para poder quitar o acceder al último elemento –es decir el que está más abajo sobre la base de la pila– es necesario quitar los anteriores que están arriba, de lo contrario se caerían todos los elementos y se destruye la pila. Si se observa la pila con un enfoque abstracto, es decir su modelo virtual, también se lo utiliza sin tenerlo en cuenta, por ejemplo, cuando se navega por internet el navegador trabaja con dos pilas que permiten ir a la

dirección anterior, es decir volver a la página que estaba antes o ir hacia adelante a una página que ya se visitó; cuando se trabaja en un editor de texto existen los botones “hacer” y “deshacer”, esto permite al usuario volver a versiones anteriores o posteriores del texto que se está editando de manera rápida; lo mismo ocurre con los editores de imágenes, etcétera. Ya a un nivel más abstracto los sistemas operativos utilizan pilas para la gestión de la memoria de los procesos, para poder ejecutar llamadas recursivas, para gestión de archivos, etc.

Por lo cual, una pila puede definirse de la siguiente manera: si consideramos su estructura y funcionamiento, es una estructura lineal dinámica de datos que no están ordenados y cuyas actividades de inserción y eliminación se realizan a través de un índice llamado cima o tope. Es importante remarcar que el orden de complejidad de las operaciones sobre la pila son de tiempo constante $O(1)$ dado que solo se pueden agregar y quitar elementos desde la parte superior de la misma, por lo que no importa la cantidad de elementos que tenga.

¿Están realmente desordenados los datos? En una pila por la naturaleza de su funcionamiento los datos no están ordenados siguiendo un criterio particular –salvo que se desarrolle un algoritmo que se encargue de ordenarlos–, pero si consideramos el orden de inserción de los elementos, se puede decir que la pila se encuentra ordenada por dicho criterio.

Hablemos del desbordamiento de pila –*¿Qué es? ¿Por qué ocurre?*– Esto está ligado casi directamente al tipo de implementación que se utilice –estática o dinámica como ya vimos en el capítulo IV–, en la figura 2 se muestran ambas alternativas de representación. Además se debe considerar que las pilas no tienen un límite superior entonces, si optamos por una implementación estática –a la izquierda en la figura– tenemos la limitación de la cantidad de elementos del vector esto implica que si la cantidad de elementos a insertar es mayor se produce un desbordamiento de la pila porque no hay espacio para almacenarlos (*stack overflow*). Pero, por otro lado, si usamos una implementación dinámica –a la derecha de la figura– con nodos enlazados disponemos de toda la memoria para crear nodos por lo cual es muy poco probable que se produzca un desbordamiento, ambas. Esta última alternativa tiene la ventaja de que trabajamos en memoria principal por lo cual si la misma se llenara –lo cual es poco probable que ocurra– el sistema operativo continuará utilizando memoria virtual (intercambiando la memoria a disco) extendiendo la memoria principal y por ende la capacidad de la pila.

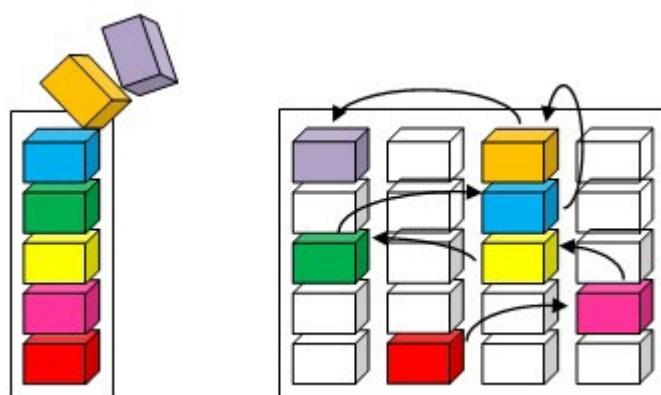


Figura 2. Tipos de representación de pila

Comencemos con el diseño del TDA pila, el mismo estará compuesto básicamente de un elemento al que denominaremos cima, este es un puntero que se encarga de apuntar a un nodoPila, que está compuesto de dos elementos información y siguiente –esto de describió en el capítulo anterior–, y un conjunto de funciones que describen su comportamiento. Estas se enumeran a continuación –adicionalmente puede agregarse el elemento tamaño en la definición del TDA para facilitar el cálculo de la cantidad de elementos en la pila y no tener que hacer una función que se encargue de contar los nodos–:

1. apilar(pila, elemento). Agrega el elemento sobre la cima de la pila;
2. desapilar(pila). Elimina y devuelve el elemento almacenado en la cima de la pila;
3. pila_vacia(pila). Devuelve verdadero (*true*) si la pila no contiene elementos;
4. cima(pila). Devuelve el valor del elemento que está almacenado en la cima de la pila pero sin eliminarlo;
5. tamaño(pila). Devuelve la cantidad de elementos en la pila.

Pasemos ahora de la definición a la implementación del TDA pila, primero en la figura 3 se puede ver la definición de la estructura. Para representar el tipo de dato pila se utiliza una clase, compuesta de dos atributos *cima* que es un puntero que apunta al nodo que está en la cima de la pila –que inicialmente tiene valor *None*–, y *tamaño* que representa la cantidad de elementos que contiene la pila cuyo valor inicial es 0; además se utiliza la clase nodoPila, compuesta de los atributos información y siguiente cuyo valor inicial es *None* en ambos casos.

```
class nodoPila(object):  
    """Clase nodo pila.  
  
    info, sig = None, None  
  
  
class Pila(object):  
    """Clase Pila.  
  
    def __init__(self):  
        """Crea una pila vacia.  
        self.cima = None  
        self.tamano = 0
```

Figura 3. Definición de la estructura del TDA pila

Luego se definen las funciones mencionadas previamente, que serán los eventos mediante los cuales se administrará el funcionamiento de la pila, estas se pueden observar en las figuras 4 y 5.

```
def apilar(pila, dato):
    """Apila el dato sobre la cima de la pila."""
    nodo = nodoPila()
    nodo.info = dato
    nodo.sig = pila.cima
    pila.cima = nodo
    pila.tamanio += 1

def desapilar(pila):
    """Desapila el elemento en la cima de la pila y lo devuelve."""
    x = pila.cima.info
    pila.cima = pila.cima.sig
    pila.tamanio -= 1
    return x
```

Figura 4. Interfaz o eventos del TDA pila parte 1

```
def pila_vacia(pila):
    """Devuelve true si la pila esta vacia."""
    return pila.cima is None

def en_cima(pila):
    """Devuelve el valor almacenado en la cima de la pila."""
    if pila.cima is not None:
        return pila.cima.info
    else:
        return None

def tamanio(pila):
    """Devuelve el numero de elementos en la pila."""
    return pila.tamanio
```

Figura 5. Interfaz o eventos del TDA pila parte 2

Cuando tenemos que apilar un nuevo elemento, se debe crear una variable del tipo nodoPila a la que se le asigna en su campo información el valor del elemento ingresado como dato. Luego en el campo siguiente de dicho nodo se le guarda la dirección de referencia de la cima y después a cima se le asigna la dirección del nodo creado. Y por último se incrementa el valor de tamaño.

Por otro lado, cuando hay que desapilar un elemento de la pila, se saca del nodo que está en la cima de la pila su información y se almacena en una variable auxiliar, luego a cima se le asigna la dirección de referencia almacenada en el atributo siguiente del nodo de la cima –es decir el siguiente nodo de la estructura, el que está abajo–. Por último se decrementa el valor de tamaño y se retorna el valor de la variable auxiliar, es decir el elemento eliminado.

El resto de las actividades son sencillas, por lo que el análisis e interpretación de las mismas quedarán a cargo del lector.

Uds. se preguntarán:

—¿Se puede hacer un barrido de una pila?

Sí, se puede. Pero no de la forma tradicional como se realiza sobre otras estructuras de datos más simples como los vectores o matrices.

—Pero ¿Cómo se hace?

Para poder barrer una pila se debe respetar la naturaleza del funcionamiento de esta estructura, es decir, solo se puede acceder al elemento almacenado en la cima. Para listar el contenido completo de una pila se debe eliminar cada elemento y mostrarlo de a uno a la vez. — ¿Pero entonces de esta forma no queda vacía la pila?— en efecto al eliminar los elementos de una pila la misma quedara vacía, pero adicionalmente se puede utilizar una pila auxiliar para almacenar temporalmente los elementos y luego reconstruir la pila original, como se puede observar en la figura 6.

```
def barrido(pila):
    """Muestra el contenido de una pila sin perder datos."""
    paux = Pila()
    while(not pila_vacia(pila)):
        dato = desapilar(pila)
        print(dato)
        apilar(paux, dato)

    while(not pila_vacia(paux)):
        dato = desapilar(paux)
        apilar(pila, dato)
```

Figura 6. Barrido y reconstrucción de pila

A modo de ejemplo en la figura 7 se presenta una implementación del TDA pila para la resolución de un problema. En este caso se tiene una pila de números enteros y se desea dividirla en dos pilas, una con los números pares y otra con los impares:

```
from tda_pila import Pila, apilar, desapilar, pila_vacia

pdatos = Pila()
ppar = Pila()
pimpar = Pila()
dato = int(input('Ingrese un número - 0 para salir '))

while(dato != 0):
    apilar(pdatos, dato)
    dato = int(input('Ingrese un número - 0 para salir '))

while(not pila_vacia(pdatos)):
    dato = desapilar(pdatos)
    if(dato % 2 == 0):
        apilar(ppar, dato)
    else:
        apilar(pimpar, dato)

while(not pila_vacia(ppar)):
    dato = desapilar(ppar)
    print(dato)

while(not pila_vacia(pimpar)):
    dato = desapilar(pimpar)
    print(dato)
```

Figura 7. Ejemplo uso del TDA pila

A continuación, se describen las acciones realizadas para la resolución del ejercicio anterior paso por paso, para poder comprender la mecánica de uso de un TDA, para ello se debe utilizar los eventos definidos en el bloque de interfaz del mismo.

Primero se deben importar del TDA pila las funciones que vamos a utilizar para resolver el problema, luego se crean las variables necesarias de tipo pila. Luego, después de cargar los datos en la pila, se procede a eliminar el elemento en la cima de la pila y se determina si el elemento desapilado es par o impar para apilarlo en la pila correspondiente (par o impar). Y se repite el mismo procedimiento para cada elemento hasta que la pila quede vacía para finalmente mostrar el contenido de ambas pilas.

Guía de ejercicios prácticos

A continuación se plantean una serie de problemas, que se deberán resolver utilizando el TDA pila.

1. Determinar el número de ocurrencias de un determinado elemento en una pila.
2. Eliminar de una pila todos los elementos impares, es decir que en la misma solo queden números pares.
3. Reemplazar todas las ocurrencias de un determinado elemento en una pila.
4. Invertir el contenido de una pila, solo puede utilizar una pila auxiliar como estructura extra.
5. Determinar si una cadena de caracteres es un palíndromo.
6. Leer una palabra y visualizarla en forma inversa.
7. Eliminar el i-ésimo elemento debajo de la cima de una pila de palabras.
8. Dada una pila de cartas de las cuales se conoce su número y palo,—que representa un mazo de cartas de baraja española—,resolver las siguientes actividades:
 - a. generar las cartas del mazo de forma aleatoria;
 - b. separar la pila mazo en cuatro pilas una por cada palo;
 - c. ordenar una de las cuatro pilas (espada, basto, copa u oro) de manera creciente.
9. Resolver el problema del factorial de un número utilizando una pila.
10. Insertar el nombre de la diosa griega Atenea en la i-ésima posición debajo de la cima de una pila con nombres de dioses griegos.
11. Dada una pila de letras determinar cuántas vocales contiene.
12. Dada una pila con nombres de los personajes de la saga de Star Wars, implemente una función que permita determinar si Leia Organa o Boba Fett están en dicha pila sin perder los datos.
13. Dada una pila con los trajes de Iron Man utilizados en las películas de Marvel Cinematic Universe (MCU) de los cuales se conoce el nombre del modelo, nombre de la película en la que se usó y el estado en que quedó al final de la película (Dañado, Impecable, Destruido), resolver las siguientes actividades:
 - a. determinar si el modelo Mark XLIV (Hulkbuster) fue utilizado en alguna de las películas, además mostrar el nombre de dichas películas;
 - b. mostrar los modelos que quedaron dañados, sin perder información de la pila.

- c. eliminar los modelos de los trajes destruidos mostrando su nombre;
 - d. un modelo de traje puede usarse en más de una película y en una película se pueden usar más de un modelo de traje, estos deben cargarse por separado;
 - e. agregar el modelo Mark LXXXV a la pila, tener en cuenta que no se pueden cargar modelos repetidos en una misma película;
 - f. mostrar los nombre de los trajes utilizados en las películas “Spider-Man: Homecoming” y “Capitan America: Civil War”.
14. Realizar un algoritmo que permita ingresar elementos en una pila, y que estos queden ordenados de forma creciente. Solo puede utilizar una pila auxiliar como estructura extra –no se pueden utilizar métodos de ordenamiento–.
15. Realizar el algoritmo de ordenamiento quicksort de manera que funcione iterativamente.
16. Se tienen dos pilas con personajes de Star Wars, en una los del episodio V de “The empire strikes back” y la otra los del episodio VII “The force awakens”. Desarrollar un algoritmo que permita obtener la intersección de ambas pilas, es decir los personajes que aparecen en ambos episodios.
17. Dado un párrafo que finaliza en punto, separar dicho párrafo en tres pilas: vocales, consonantes y otros caracteres que no sean letras (signos de puntuación números, espacios, etc.). Luego utilizando las operaciones de pila resolver las siguientes consignas:
- a. cantidad de caracteres que hay de cada tipo (vocales, consonantes y otros);
 - b. cantidad de espacios en blanco;
 - c. porcentaje que representan las vocales respecto de las consonantes sobre el total de caracteres del párrafo;
 - d. cantidad de números;
 - e. determinar si la cantidad de vocales y otros caracteres son iguales;
 - f. determinar si existe al menos una z en la pila de consonantes.
18. Dada una pila de objetos de una oficina de los que se dispone de su nombre y peso (por ejemplo monitor 1 kg, teclado 0.25 kg, silla 7 kg, etc.), ordenar dicha pila de acuerdo a su peso –del objeto más liviano al más pesado–. Solo pueden utilizar pilas auxiliares como estructuras extras, no se pueden utilizar métodos de ordenamiento.
19. Dada una pila de películas de las que se conoce su título, estudio cinematográfico y año de estreno, desarrollar las funciones necesarias para resolver las siguientes actividades:
- a. mostrar los nombre películas estrenadas en el año 2014;

- b. indicar cuántas películas se estrenaron en el año 2018;
 - c. mostrar las películas de Marvel Studios estrenadas en el año 2016.
20. Realizar un algoritmo que registre los movimientos de un robot, los datos que se guardan son cantidad de pasos y dirección –suponga que el robot solo puede moverse en ocho direcciones: norte, sur, este, oeste, noreste, noroeste, sureste y suroeste–. Luego desarrolle otro algoritmo que genere la secuencia de movimientos necesarios para hacer volver al robot a su lugar de partida, retornando por el mismo camino que fue.
21. Realizar un algoritmo que ingrese en una pila los dos valores iniciales de la sucesión de Fibonacci –o condiciones de fin de forma recursiva– y a partir de estos calcular los siguientes valores de dicha sucesión, hasta obtener el valor correspondiente a un número n ingresado por el usuario.
22. Se recuperaron las bitácoras de las naves del cazarrecompensas Boba Fett y Din Djarin (The Mandalorian), las cuales se almacenaban en una pila (en su correspondiente nave) en cada misión de caza que emprendió, con la siguiente información: planeta visitado, a quien capturó, costo de la recompensa. Resolver las siguientes actividades:
- a. mostrar los planetas visitados en el orden que hicieron las misiones cada uno de los cazzarrecompensas;
 - b. determinar cuántos créditos galácticos recaudo en total cada cazarrecompensas y de estos quien obtuvo mayor fortuna;
 - c. determinar el número de la misión –es decir su posición desde el fondo de la pila– en la que Boba Fett capturo a Han Solo, suponga que dicha misión está cargada;
 - d. indicar la cantidad de capturas realizadas por cada cazarrecompensas.
23. Dada una pila con los valores promedio de temperatura ambiente de cada día del mes de abril, obtener la siguiente información sin perder los datos:
- a. determinar el rango de temperatura del mes, temperatura mínima y máxima;
 - b. calcular el promedio de temperatura (o media) del total de valores;
 - c. determinar la cantidad de valores por encima y por debajo de la media.
24. Dada una pila de personajes de Marvel Cinematic Universe (MCU), de los cuales se dispone de su nombre y la cantidad de películas de la saga en la que participó, implementar las funciones necesarias para resolver las siguientes actividades:
- a. determinar en qué posición se encuentran Rocket Raccoon y Groot, tomando como posición uno la cima de la pila;

- b. determinar los personajes que participaron en más de 5 películas de la saga, además indicar la cantidad de películas en la que aparece;
- c. determinar en cuantas películas participo la Viuda Negra (Black Widow);
- d. mostrar todos los personajes cuyos nombre empiezan con *C*, *D* y *G*.

CAPÍTULO VII

Por favor espere su turno en la cola que será atendido

La estructura de datos cola es otra estructura lineal sencilla como la anterior, administrada por dos eventos. Dicha estructura es clave para el funcionamiento de los sistemas operativos que la utilizan para la gestión y planificación de procesos. También es usada por otras aplicaciones que deben manejar la concurrencia del uso de recursos compartidos –por ejemplo la impresora–, dado que estas pueden trabajar administrando prioridades. Una cola es una colección de elementos que se agregan y quitan basándose en el principio de primero en entrar-primer en salir (FIFO, *First In-First Out*). Esto quiere decir que el primer elemento en ser *insertado* es el primero en ser *eliminado*. Un elemento puede ser agregado en cualquier momento a una cola, esto se hace por el final, pero solo se puede acceder o *eliminar* el elemento que esté en el frente de la misma, como se observa en la figura 1.

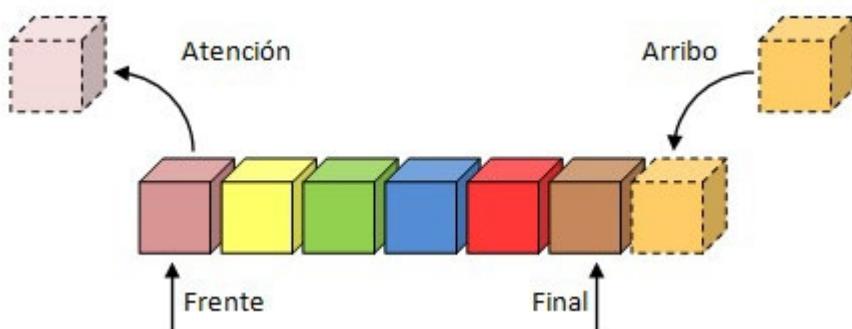


Figura 1. Funcionamiento de una cola

Como se puede ver en la figura 1, hay dos actividades o eventos que se encargan de administrar los elementos de una cola. Entonces igual que en el caso del TDA pila podríamos volver a afirmar que el valor del dato almacenado no influye en la manejo de la estructura –manteniéndose entonces una relevancia secundaria del valor de los datos para la estructura, más allá de la importancia de almacenarlos–, salvo los casos particulares en los que se aplica cola de prioridad donde, además de los eventos, la prioridad de los datos influye en el funcionamiento de la estructura. Estos eventos son dos: *arribo* cuando se agrega un nuevo elemento al final y *atención* cuando se saca el elemento que está en el frente.

Esto implica que solo se puede acceder al elemento que está en el frente de la cola, la forma de acceder a los demás elementos de la colección es atender cada uno de estos, de a uno a la vez.

Algunos ejemplos de la vida cotidiana donde se pueden encontrar el uso de cola de manera natural son en la parada del colectivos, en las cajas de los supermercados, venta de entradas de evento, en los cajeros automáticos, turnos en la guardia de un hospital o consultorio, etcétera. En cualquiera de estos casos para poder ser atendidos debemos pararnos al final de la cola y esperar que nos toque nuestro turno, es decir que atiendan a todos los que están antes. A medida que ocurre esto nos

vamos acercando al frente de la cola, de lo contrario nos estaríamos colando. Si se ve la cola desde un enfoque abstracto, es decir, la representación virtual de este modelo también lo utilizamos de manera implícita sin darnos cuenta de esto, por ejemplo si hay una impresora en una casa u organización a la cual se envían a imprimir documentos de diferentes máquinas, se imprime el primero que llega y el resto queda en espera –es decir quedan en la cola de impresión para ser atendidos una vez finalice de imprimir el documento actual–. Y a un nivel aún más abstracto los sistemas operativos utilizan cola para la gestión de procesos –de hecho tiene varias colas funcionando en simultáneo para los procesos listos, suspendidos, bloqueados, etcétera–.

Entonces se puede definir una cola del siguiente modo: basándonos en su estructura y funcionamiento: es una estructura lineal dinámica de datos que no están ordenados y cuyas actividades de *inserción* y *eliminación* se realizan a través de los índices llamados frente y final respectivamente, sobre la cual además se puede operar con criterios de prioridad. De igual manera que ocurre con la pila el orden de complejidad de las actividades para administrar elementos en la cola es del orden de $O(1)$ dado que las misma se realizan en el frente o el final sin importar la cantidad de elementos que tenga.

Al igual que vimos para pila, en una cola por la naturaleza de su funcionamiento los datos no están ordenados siguiendo un criterio particular –salvo que se desarrolle un algoritmo que se encargue de hacerlo–, pero podemos considerar que están ordenados según el orden de inserción de los elementos.

Al momento de la representación tendremos las mismas alternativas que con la pila, como ya mencionamos previamente tendremos la limitación de cantidad de elementos si optamos por la implementación estática. Además como vemos en la figura 2 si el vector está completo y de repente se elimina un elemento, quedaría un lugar disponible, pero si intentamos agregar un elemento no tendremos lugar al final del vector –dado que el hueco está al principio del mismo–. Entonces, ¿*Cómo solucionamos este problema?* Para esto tendremos dos soluciones: la primera desplazar todos los elementos a la izquierda lo cual sería poco eficiente –dado que la operación de inserción sería del orden de $O(n)$ –; la segunda es implementarlo como un vector circular entonces cuando los índices llegan a la última posición del vector pasan al inicio del mismo donde si hay lugar para hacer la inserción, con lo cual volveríamos a tener operaciones del orden de $O(1)$, ambas soluciones se pueden observar a la izquierda y derecha respectivamente al fondo de la figura. De todos modos en ambos casos estamos limitados por el espacio del vector por lo cual la mejor alternativa es implementarlo con nodos enlazados al igual que en el TDA pila.

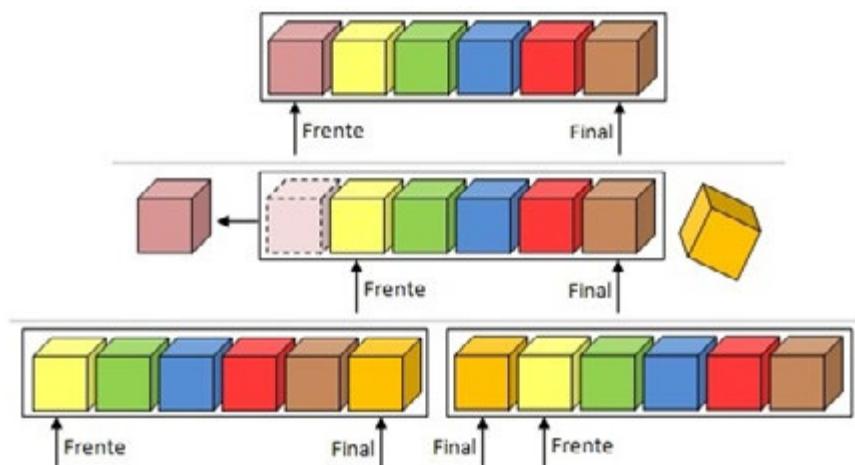


Figura 2.

Arranquemos con el diseño del TDA cola, el cual estará compuesto básicamente de dos elementos que llamaremos frente y final. Estos son punteros que se encargan de apuntar a un nodoCola, que su vez está compuesto de dos elementos, información y siguiente –como ya hemos visto anteriormente–, y las funciones que describen su comportamiento, que se enumeran a continuación –opcionalmente puede agregarse el elemento tamaño para determinar la cantidad de elementos en la cola, y evitar tener que contar los nodos con una función adicional–:

1. arribo(cola, elemento). Agrega el elemento a la final de la cola;
2. atención(cola). Elimina y devuelve el elemento almacenado en el frente de la cola;
3. cola_vacia(cola). Devuelve verdadero (*true*) si la cola no contiene elementos;
4. en_frente (cola). Devuelve el valor del elemento que está almacenado en el frente de la cola sin eliminarlo;
5. tamaño(cola). Devuelve la cantidad de elementos en la cola;
6. mover_al_final(cola). Elimina el elemento en el frente de la cola y lo inserta en el final de la misma;

Para continuar, es momento de implementar el TDA cola. En primer lugar se puede observar en la figura 3 la definición de la estructura. Para representar TDA cola volvemos a usar una clase, que consta de tres atributos frente y final que son punteros que apunta al nodo que está en la frente y final de la cola respectivamente –dichos punteros inicialmente tienen valor *None*– y tamaño que indica la cantidad de elementos que contiene la cola cuyo valor inicial es 0. Además se utiliza la clase nodoCola la cual está definida por los atributos información y siguiente cuyo valor inicial es *None* en ambos casos.

```
class nodoCola(object):  
    """Clase nodo cola.  
  
    info, sig = None, None  
  
class Cola(object):  
    """Clase Cola.  
  
    def __init__(self):  
        """Crea una cola vacia.  
        self.frente, self.final = None, None  
        self.tamamio = 0
```

Figura 3. Definición de la estructura del TDA cola

Seguimos ahora con las figuras 4 y 5 en las que se puede ver la definición de las funciones mencionadas anteriormente, estas serán los eventos que permitirán administrar el funcionamiento de la cola.

```
def arribo(cola, dato):
    """Arriba el dato al final de la cola."""
    nodo = nodoCola()
    nodo.info = dato
    if cola.frente is None:
        cola.frente = nodo
    else:
        cola.final.sig = nodo
    cola.final = nodo
    cola.tamamio += 1

def atencion(cola):
    """Atiende el elemento en el frente de la cola y lo devuelve."""
    dato = cola.frente.info
    cola.frente = cola.frente.sig
    if cola.frente is None:
        cola.final = None
    cola.tamamio -= 1
    return dato
```

Figura 4. Interfaz o eventos del TDA cola parte 1

```
def cola_vacia(cola):
    """Devuelve true si la cola esta vacia."""
    return cola.frente is None

def en_frente(cola):
    """Devuelve el valor almacenado en el frente de la cola."""
    return cola.frente.info

def tamamio(cola):
    """Devuelve el numero de elementos en la cola."""
    return cola.tamamio

def mover_al_final(cola):
    """Mueve el elemento del frente de la cola al final."""
    dato = atencion(cola)
    arribo(cola, dato)
    return dato
```

Figura 5. Interfaz o eventos del TDA cola parte 2

Al momento de arribar un nuevo elemento a la cola, se crea una variable de tipo nodoCola, a la cual en su campo información se le asigna el elemento ingresado como dato. Si el elemento ingresado es el primero, al frente de la cola se le debe asignar la dirección del nodo creado, caso contrario al final de la cola en el campo siguiente se le asigna la dirección del nodo creado. Finalmente cualquiera sea el caso que haya ocurrido al nodo apuntado por final se le asigna la dirección del nodo creado y se incrementa el valor de tamaño.

En cambio cuando se realiza una atención de un elemento de la cola, primero se obtiene la información del nodo que está en el frente de la cola y se lo almacena en una variable auxiliar, luego a frente le asignamos la dirección de referencia almacenada en el campo siguiente, del nodo que está en el frente de la cola. Además si el elemento eliminado es el último de la cola, a final se le debe asignar valor *None*. Por último se decrementa el valor de tamaño y se retorna el valor de la variable auxiliar, es decir el elemento que se quitó.

Nuevamente como en toda estructura de datos seguramente en algún momento vamos a necesitar hacer un barrido de sus elementos, al igual que pasó con pila en el capítulo anterior esta estructura no permite hacer un barrido de forma natural. Pero como ustedes ya saben, este barrido puede hacerse pero de manera tal que respete la naturaleza del funcionamiento de la cola utilizando una cola auxiliar para almacenar temporalmente los elementos y luego poder reconstruir la estructura original para no perder los datos. Esto se muestra en la figura 6, pero es importante remarcar que tanto este barrido como el de pila –dado que en esencia son iguales– son muy inefficientes ya que son del orden de $O(n^2)$.

```
def barrido(cola):
    """Muestra el contenido de una cola sin perder datos."""
    caux = Cola()
    while(not cola_vacia(cola)):
        dato = atencion(cola)
        print(dato)
        arribo(caux, dato)

    while(not cola_vacia(caux)):
        dato = atencion(caux)
        arribo(cola, dato)
```

Figura 6. Barrido y reconstrucción de cola

Pero además la cola tiene una gran diferencia respecto de la pila, la primera utiliza dos elementos para administrar la inserción y eliminación de datos mientras que la segunda solo tiene uno. Esto implica que en la cola se pueden realizar las operaciones para agregar o quitar datos de manera independiente, lo que nos permitirá mejorar la eficiencia del algoritmo de barrido. Para hacer esto vamos a utilizar la función mover_al_final, ya que esta estructura posee dos punteros uno para la inserción (frente) y otro para la eliminación (final). También vamos a necesitar la función tamaño para determinar cuántos elementos hay en la cola y repite dicha cantidad de veces la función mover_al_final, esto nos permitirá atender el elemento que está en el frente de la cola, del cual obtendremos la información para mostrar –o resolver otra acción– y luego hacer el arribo de dicho elemento al final como se puede ver en la figura 7.

Ahora si comparamos la complejidad de ambos barridos, el primer caso está en el orden de $O(n^2)$ dado que se pasan todos los elementos a una cola auxiliar y después se los vuelve a pasar a la cola original para reconstruirla; mientras que el segundo caso es del orden de $O(n)$ dado que solo pasa los elementos del frente al final logrando una mayor eficiencia por parte de este último algoritmo.

```
def barrido(cola):
    """Muestra el contenido de una cola sin perder datos."""
    i = 0
    while(i < tamanio(cola)):
        dato = mover_al_final(cola)
        print(dato)
        i += 1
```

Figura 7. Barrido usando función mover al final

Para entender cómo usar el TDA cola en la figura 8 se presenta un ejemplo de su uso para la resolución de un problema. Para este caso particular se cuenta con una cola de caracteres y se desea dejar en ella solamente las vocales:

```
from tda_colas import Cola, arribo, atencion, cola_vacia

cdatos = Cola()
cvocales = Cola()

letra = input('Ingrese un caracter ')
while(letra != ''):
    arribo(cdatos, letra)
    letra = input('Ingrese un caracter ')

while(not cola_vacia(cdatos)):
    letra = atencion(cdatos)
    if letra.upper() in ['A', 'E', 'I', 'O', 'U']:
        arribo(cvocales, letra)

print('Datos cola vocales')
while(not cola_vacia(cvocales)):
    dato = atencion(cvocales)
    print(dato)
```

Figura 8. Ejemplo de uso del TDA Cola

Ahora, se presenta una breve descripción de las acciones realizadas para la resolución del ejercicio anterior, para lo cual se utilizan los eventos definidos en el bloque de interfaz.

En primer lugar se importan del TDA cola las funciones que utilizaremos en la resolución del problema enunciado, luego se crean las variables de tipo cola que necesitaremos. Despues debemos

cargar los datos en la cola para poder procesarlos de la siguiente manera: eliminamos el elemento en el frente de la cola y determinar si el elemento atendido es una vocal o no, en el caso de serlo hacemos la inserción de dicha letra en la cola de vocales. Luego repetimos este procedimiento hasta que la cola esté vacía y terminamos mostrando el contenido de la cola de vocales.

Adelantarse en la cola de manera correcta utilizando prioridad

Como mencionamos previamente la cola permite aplicar una propiedad especial que altera en cierta medida su principio de funcionamiento, la prioridad. Es decir que al momento de insertar un elemento, no siempre irá al final, su prioridad puede alterar la posición donde el dato debe ser almacenado; logrando de esta manera un tratamiento especial para determinados datos que sean más importantes que el resto –por esta particularidad es utilizada internamente por los sistemas operativos para la gestión de recursos–. la mayoría de los ejemplos mencionados previamente utilizan colas de prioridad aunque muchas veces no nos demos cuenta, en el caso de colas de personas siempre tienen prioridad los mayores, mujeres embarazadas y con discapacidad. Por su parte en una empresa el departamento o máquina desde la cual se envíe el documento a imprimir puede tener más prioridad, por ejemplo, el documento del gerente que el del resto de los empleados. Y lo mismo ocurre con las colas de los sistemas operativos, por defecto los procesos del sistema siempre tendrán mayor prioridad que el de los usuarios.

Uds. se preguntarán *¿Cómo funciona o se aplica la prioridad en una cola? ¿Todo lo que hemos hecho hasta ahora no sirve más?* Todo lo que hemos desarrollado seguirá funcionando perfectamente y lo seguiremos usando, lo único que debemos agregar es una función más que denominaremos “arribo_con_prioridad” –la cual se describe a continuación–, además para trabajar con colas de prioridad al nodoCola debemos agregarle un atributo extra denominado prioridad:

- arribo_con_prioridad(cola, elemento, prioridad). Agrega el elemento al final de la cola y luego lo acomoda según su criterio de prioridad. Para poder aplicar el mecanismo de prioridad en una cola el nodoCola, este debe contener el atributo prioridad.

Veamos ahora cómo funciona el arribo con prioridad, para esto se trabajará con el siguiente caso: supongamos que debemos diseñar un algoritmo para administrar las atenciones de la guardia de un hospital, donde los turnos de los pacientes además de ordenarlos por orden de llegada –es un caso típico de cola hasta aquí– se los clasifica según el siguiente criterio de prioridad:

1. Consulta, por ejemplo pacientes con gripe, dolor de cabeza, control mensual, un esguince, etc.
2. Emergencia, por ejemplo pacientes con quebraduras expuestas, con heridas y pérdida de sangre, quemaduras graves, mujeres con trabajo de parto, etc.
3. Urgencia, por ejemplo pacientes con paro respiratorio, con un ataque cerebro vascular, con convulsiones, etc.

Para este ejemplo tenemos tres niveles de prioridad para consultas (1), emergencias (2) y urgencias (3), pero podrían ser dos o más, esto dependerá del tipo de problemas que se deban que resolver. Entonces de acuerdo a estos criterios si arriba un paciente con prioridad tres debería insertarse al final de la cola y adelantarse mientras su prioridad sea mayor que la del paciente que esté antes –si ya existe otro paciente con prioridad tres, este quedará atrás–.

Entonces *¿Qué sucede cuando arriba un nuevo elemento a la cola?* Si la cola está vacía el elemento quedará primero y solo se deberá utilizar la función de arribo. Pero en caso contrario lo que debería suceder –si se respeta la naturaleza del modelo real– es que el elemento se agregue al final y a partir de ese lugar se comience a adelantar mientras su prioridad sea mayor que la del elemento anterior. De acuerdo a la representación actual del TDA cola no se puede reflejar esta acción, dado que no se cuenta un puntero que permita pasar al nodo anterior de la estructura para determinar la nueva ubicación.

Si bien se podría dar solución de manera sencilla al uso de colas de prioridad sin alterar demasiado el TDA, para lo cual deberíamos mover al final todos los elementos que tengan prioridad mayor o igual al nuevo elemento, luego insertar el nuevo elemento y después el resto de los elementos que quedaron pendientes en la cola. Al hacer la inserción con prioridad de esta manera, la función arribo pasaría de estar en el orden de $O(1)$ en arribo sin prioridad a estar en el orden de $O(n)$ para arribos con prioridad, lo cual es muy ineficiente por esta razón no implementaremos aún la función “*arribo_con_prioridad*”; lo dejaremos pendiente para resolverlo más adelante en el capítulo XII utilizando la estructura de datos montículo, lo cual nos permitirá resolver la operación de arribo aplicando prioridad de manera eficiente.

Guía de ejercicios prácticos

A continuación se plantean una serie de problemas que deberán resolver utilizando los TDA cola y pila.

- I. Eliminar de una cola de caracteres todas las vocales que aparecen.
- II. Utilizando operaciones de cola y pila, invertir el contenido de una cola.
- III. Dada una secuencia de caracteres utilizando operaciones de cola y pila determinar si es un palíndromo.
- IV. Dada una cola de números cargados aleatoriamente, eliminar de ella todos los que no sean primos.
- V. Utilizando operaciones de cola y pila, invertir el contenido de una pila.
- VI. Contar la cantidad de ocurrencias de un determinado elemento en una cola, sin utilizar ninguna estructura auxiliar.
- VII. Eliminar el i-ésimo elemento después del frente de la cola.
- VIII. Realizar un algoritmo que mantenga ordenado los elementos agregados a una cola, utilizando solo una cola como estructura auxiliar.
- IX. Dada una cola de valores enteros calcular su rango y contar cuántos elementos negativos hay.
- X. Dada una cola con las notificaciones de las aplicaciones de redes sociales de un Smartphone, de las cuales se cuenta con la hora de la notificación, la aplicación que la emitió y el mensaje, resolver las siguientes actividades:
 - a. escribir una función que elimine de la cola todas las notificaciones de Facebook;
 - b. escribir una función que muestre todas las notificaciones de Twitter, cuyo mensaje incluya la palabra 'Python', si perder datos en la cola;
 - c. utilizar una pila para almacenar temporalmente las notificaciones producidas entre las 11:43 y las 15:57, y determinar cuántas son.
- XI. Dada una cola con personajes de la saga Star Wars, de los cuales se conoce su nombre y planeta de origen. Desarrollar las funciones necesarias para resolver las siguientes actividades:
 - a. mostrar los personajes del planeta Alderaan, Endor y Tatooine
 - b. indicar el planeta natal de Luke Skywalker y Han Solo
 - c. insertar un nuevo personaje antes del maestro Yoda
 - d. eliminar el personaje ubicado después de Jar Jar Binks

12. Dada dos colas con valores ordenadas, realizar un algoritmo que permita combinarlas en una nueva cola. Se deben mantener ordenados los valores sin utilizar ninguna estructura auxiliar, ni métodos de ordenamiento.
13. Dada una cola de 50000 caracteres generados aleatoriamente realizar las siguientes actividades:
- separarla en dos colas una con dígitos y otra con el resto de los caracteres.
 - determinar cuántas letras hay en la segunda cola.
 - determinar además si existen los caracteres “?” y “#”.
14. Realizar un algoritmo que permita realizar las siguientes funciones:
- cargar semáforos de una rotonda y sus respectivos tiempos de encendido en verde –cargue al menos tres semáforos–.
 - simular el funcionamiento de los semáforos cargados (cola circular).
 - debe mostrar por pantalla el cambio de colores y el número del semáforo.

15. Suponga que se escapa hacia el planeta tierra en un Caza TIE robado –huyendo de un Destructor Estelar y necesita localizar la base rebelde más cercana– y se tiene una cola con información de las bases rebeldes en la tierra de las cuales conoce su nombre, número de flota aérea, coordenadas de latitud y longitud. Desarrolle un algoritmo que permita resolver las siguientes tareas una vez que aterrice:

- determinar cuál es la base rebelde más cercana desde su posición actual.
- para el cálculo de la distancia deberá utilizar la fórmula de Haversine:

$$dist = 2 \cdot r \cdot \arcsin \left(\sqrt{ \sin^2 \left(\frac{(\varphi_2 - \varphi_1)}{2} \right)^2 + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \sin^2 \left(\frac{(\lambda_2 - \lambda_1)}{2} \right)^2 } \right)$$

donde r es el radio medio de la tierra en metros (6371000), φ_1 y φ_2 las latitudes de los dos puntos –por ejemplo coordenadas actual–, λ_1 y λ_2 las longitudes de los dos puntos –coordenadas de la base– ambos expresadas en radianes; para convertir de grados a radianes utilice la función `math.radians(ángulo coordenada)`.

- mostrar el nombre y la distancia a la que se encuentran las tres bases más cercanas y determinar cual tiene mayor flota aérea.
- determinar la distancia hasta la base rebelde con mayor flota aérea.

16. Utilice cola de prioridad, para atender la cola de impresión tomando en cuenta el siguiente criterio (1- empleados, 2- staff de tecnologías de la información “TI”, 3- gerente), y resuelva la siguiente situación:
- cargue tres documentos de empleados (cada documento se representa solamente con un nombre).
 - imprima el primer documento de la cola (solamente mostrar el nombre de este por pantalla).
 - cargue dos documentos del *staff* de TI.
 - cargue un documento del gerente.
 - imprima los dos primeros documentos de la cola.
 - cargue dos documentos de empleados y uno de gerente.
 - imprima todos los documentos de la cola de impresión.
17. Desarrollar un algoritmo que permita cargar procesos a la cola de ejecución de un procesador, de los cuales se conoce id del proceso y tiempo de ejecución. Resuelva las siguientes situaciones:
- simular la atención de los procesos de la cola transcurriendo su tiempo –utilizando la función *time.sleep* (segundos) – hasta que se vacíe la cola.
 - considerar que el quantum de tiempo asignado por el procesador a cada proceso es como máximo 4.5 segundos –si el proceso no terminó su ejecución deberá volver a la cola con el tiempo restante para terminar su ejecución–.
 - cuando se realiza el cambio de proceso, porque finalizó su ejecución o porque se le agotó el quantum de tiempo, se pueden ingresar nuevos procesos a la cola por el usuario.
 - no se aplican criterios de prioridad en los procesos.
18. Dada una cola con los códigos de turnos de atención (con el formato #@@@, donde # es una letra de la A hasta la F y “@@@” son tres dígitos desde el 000 al 999), desarrollar un algoritmo que resuelva las siguientes situaciones:
- cargar 1000 turnos de manera aleatoria a la cola.
 - separar la cola con datos en dos colas, *cola_1* con los turnos que empiezan con la letra A, C y F, y la *cola_2* con el resto de los turnos (B, D y E).
 - determinar cuál de las colas tiene mayor cantidad de turnos, y de esta cuál de las letras tiene mayor cantidad.
 - mostrar los turnos de la cola con menor cantidad de elementos, cuyo número de turno sea mayor que 506.

19. Modificar las funciones de arribo y atención del TDA cola para adaptarlo a una cola circular, que no necesite la función mover al final; y desarrollar un función que permita realizar un barrido de dicha estructura respetando el principio de funcionamiento de la cola.
20. Desarrollar un algoritmo para el control de un puesto de peaje (que posee 3 cabinas de cobro), que resuelva las siguientes actividades:
 - a. agregar 30 vehículos de manera aleatoria a las cabinas de cobro, los tipos de vehículos son los siguientes:
 - I. automóviles (tarifa \$47);
 - II. camionetas (tarifa \$59);
 - III. camiones (tarifa \$71);
 - IV. colectivos (tarifa \$64).
 - b. realizar la atención de las cabinas, considerando las tarifas del punto anterior.
 - c. determinar qué cabina recaudó mayor cantidad de pesos (\$).
 - d. determinar cuántos vehículos de cada tipo se atendieron en cada cola.
21. Desarrollar un algoritmo que permita administrar los despegues y aterrizajes de un aeropuerto que tiene una pista, contemplando las siguientes actividades:
 - a. de cada vuelo se conoce el nombre de la empresa, hora salida, hora llegada, aeropuerto de origen, aeropuerto de destino y su tipo (pasajeros, negocios o carga).
 - b. utilizar una cola para administrar los despegues, se deben cargar ordenados por horario de salida. Otra para los aterrizajes, se deben agregar a medida que arriban al aeropuerto.
 - c. en la pista solo puede haber un avión realizando una maniobra de aterrizaje o despegue.
 - d. se debe permitir agregar vuelos tanto de aterrizaje como de despegue en ambas colas después de realizar una atención.
 - e. se debe atender siempre que se pueda a los elementos de la cola de aterrizaje –dado que son aviones que están sobrevolando en la zona de espera–, salvo que sea el horario de salida del primer avión de la cola de despegue, en ese caso se deberá atender dicho despegue.

- f. cada tipo de avión tiene su tiempo de uso de la pista para la maniobra de despegue y aterrizaje –adaptados a segundo para los fines prácticos del ejercicio–:
- I. pasajeros (aterrizaje = 10 segundos, despegue = 5 segundos);
 - II. negocios (aterrizaje = 5 segundos, despegue = 3 segundos);
 - III. carga (aterrizaje = 12 segundos, despegue = 9 segundos).
- g. se debe poder cancelar vuelos de despegue y poder reprogramar un vuelo para más tarde cuando se lo atiende para despegar (en esta caso el horario de salida será mayor que el último de la cola).
22. Se tienen una cola con personajes de Marvel Cinematic Universe (MCU), de los cuales se conoce el nombre del personaje, el nombre del superhéroe y su género (Masculino *M* y Femenino *F*) –por ejemplo {Tony Stark, Iron Man, *M*}, {Steve Rogers, Capitán América, *M*}, {Natasha Romanoff, Black Widow, *F*}, etc., desarrollar un algoritmo que resuelva las siguientes actividades:
- a. determinar el nombre del personaje de la superhéroe Capitana Marvel;
 - b. mostrar los nombre de los superhéroes femeninos;
 - c. mostrar los nombres de los personajes masculinos;
 - d. determinar el nombre del superhéroe del personaje Scott Lang;
 - e. mostrar todos datos de los superhéroes o personaje cuyos nombres comienzan con la letra *S*;
 - f. determinar si el personaje Carol Danvers se encuentra en la cola e indicar su nombre de superhéroes.

Como vagones de un tren enlazamos elementos para construir una lista enlazada

Cuando hablamos de estructuras lineales significa que sus elementos están en secuencia uno detrás de otro siguiendo un orden, las listas enlazadas son su principal referente. Estas presentan muchas diferencias respecto de las estructuras anteriores: en primer lugar los eventos para *insertar* e *eliminar* un elemento pueden hacerse en cualquier lugar de la lista –ya sea al principio, en el medio o al final–. En segundo lugar, los elementos están ordenados por lo que el dato a almacenar influye en el funcionamiento de la estructura, dado que a partir del mismo se determinará la posición en la cual debe agregarse. La lista es una estructura clásica utilizada para la gestión de la información, ya que los elementos ordenados por lo cual hacer una búsqueda o emitir un listado es algo sencillo. Una lista es una colección lineal de elementos a los que podemos acceder de manera aleatoria, es decir que podemos acceder a cualquier elemento de la lista. El acceso a la lista se realiza desde su inicio, como se observa en la figura 1. Además existen distintos tipos de listas las cuales analizaremos a lo largo del capítulo.

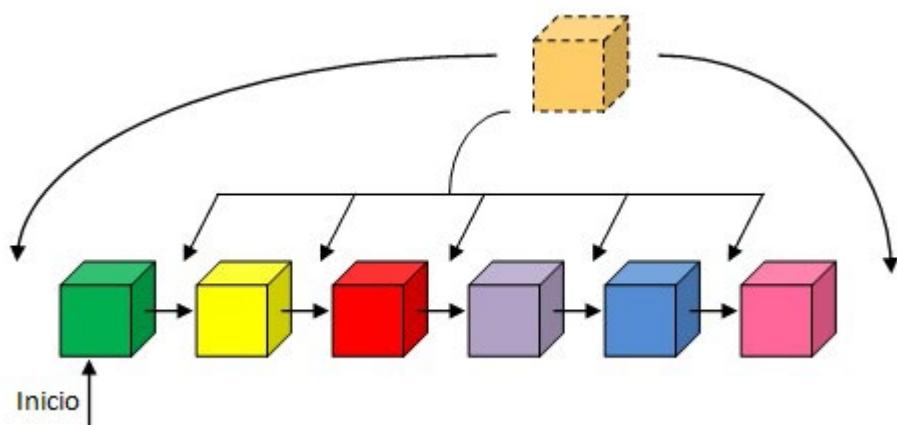


Figura 1. Funcionamiento de una lista

Una diferencia muy importante de destacar respecto de las estructuras vistas anteriormente, es que las listas pueden ser barridas de principio a fin realizando distintos tipos de actividades, como pueden ser mostrar su contenido, buscar un elemento, sumar su contenido, realizar una determinada acción con su contenido, etcétera; siendo las primeras de estas la más comunes.

Existen numerosos ejemplos del uso de listas que podemos encontrar en la vida cotidiana: lista de tareas a hacer, una receta, los alumnos de un curso, productos de un supermercado, padrón electoral, etc. Nótese que en estos tres últimos casos es muy importante que el criterio de orden sea alfabéticamente mientras que en los primeros dos el criterio podría ser el orden de las tareas a realizar.

Si se ve desde un enfoque informático, las listas están presentes en un montón de aplicaciones: contactos en un teléfono celular, archivos dentro de una carpeta, correos electrónicos en la bandeja de entrada. En cualquiera de estos casos se puede variar el criterio de orden, ya sea por nombre, fecha de edición, importancia, etcétera y además puede ser ascendente o descendente.

Por lo tanto se puede definir una lista basándose en su estructura y funcionamiento, como una estructura lineal dinámica de datos que están ordenados –por algún criterio– en la cual podemos realizar las actividades de *inserción*, *eliminación* y *búsqueda*. Estas actividades se deberán hacer por un campo clave, las dos primeras (*inserción* y *eliminación*) pueden hacerse en cualquier posición de la lista. En la inserción para determinar la posición donde se agregara el nuevo elemento para que quede ordenado –por lo general en orden ascendente–, mientras que para la eliminación la clave determinará el elemento a quitar –si la clave no se encuentra en la lista no se elimina ningún elemento–. Esto último también marca una gran diferencia respecto de las estructuras vistas anteriormente, dado que a la hora de eliminar un elemento debemos indicarle cuál se quiere quitar.

A la hora de representar la lista nuevamente nos enfrentamos al dilema de optar por una implementación estática o dinámica: con la primera de estas opciones cada vez que insertamos o eliminamos un elemento debemos hacer un desplazamiento a la derecha y a la izquierda respectivamente, por lo cual el orden de estas operaciones están en el orden de $O(n^2)$. Mientras que con la segunda no es necesario operaciones de desplazamiento ya que son variables enlazadas, esto implica que estas actividades son del orden de $O(n)$ así que en este aspecto la implementación dinámica es mucho más rápida.

Por otra parte, realizar una búsqueda con una implementación estática es eficiente ya que podremos acceder mediante índices y aplicar búsqueda binaria, lo cual no se puede aplicar con la implementación dinámica, así que en este aspecto la implementación estática es más eficiente $O(\log n)$ frente a $O(n)$. Ya que, como la implementación estática nos limita respecto de la cantidad de elementos, optaremos por hacerlo de manera dinámica.

Para poder comprender el funcionamiento básico de una lista, comencemos por observar el funcionamiento del tipo más básico, lista simplemente enlazada. Entonces pasemos ahora a definir el TDA lista: compuesto básicamente de un elemento al que denominaremos inicio (también suele llamarse cabecera), que es un puntero que se encarga de apuntar a un nodoLista, el primero de la lista, compuesto este último de dos elementos información y siguiente –al igual que los TDA anteriores–, y las operaciones que describen sus comportamientos. Además se puede agregar el elemento tamaño para facilitar el cálculo de la cantidad de elementos en la lista y no tener la tarea extra de contar los nodos:

1. `insertar(lista, elemento)`. Agrega el elemento a la lista de manera que el mismo quede ordenado;
2. `eliminar(lista, clave)`. Elimina y devuelve de la lista si encuentra un elemento que coincida con la clave dada –el primero que encuentre–, si devuelve `None` significa que no se encontró la clave en la lista, y por ende no se elimina ningún elemento;
3. `lista_vacia(lista)`. Devuelve verdadero (`true`) si la lista no contiene elementos;

4. buscar(lista, clave). Devuelve un puntero que apunta al nodo que contiene un elemento que coincida con la clave –el primero que encuentra–, si devuelve *None* significa que no se encontró la clave en la lista;
5. tamaño(lista). Devuelve la cantidad de elementos en la lista;
6. barrido(lista). Realiza un recorrido de la lista mostrando la información de los elementos almacenado en la lista.

Es tiempo de pasar a la implementación del TDA lista simplemente enlazada, para esto primero en la figura 2 se presenta la definición de la estructura. Nuevamente se utilizará una clase para representar el TDA lista, la misma estará formada por dos atributos, uno llamado inicio que es un puntero que apunta al nodo que está al principio de la lista –que inicialmente tendrá valor *None*–, y el segundo denominado tamaño que representa la cantidad de elementos que contiene la lista que comenzará con valor 0. Será necesario también definir la estructura del nodoLista como ya hicimos previamente con los campos información y siguiente.

```
class nodoLista(object):
    """Clase nodo lista."""

    info, sig = None, None


class Lista(object):
    """Clase lista simplemente enlazada."""

    def __init__(self):
        """Crea una lista vacia."""
        self.inicio = None
        self.tamano = 0
```

Figura 2. Definición de la estructura del TDA lista

Luego en las figuras 3,4 y 5 se presenta la definición de las funciones mencionadas anteriormente, estas serán los eventos que se dispondrán para administrar el funcionamiento de la lista.

```

def insertar(lista, dato):
    """Inserta el dato pasado en la lista."""
    nodo = nodoLista()
    nodo.info = dato
    if (lista.inicio is None) or (lista.inicio.info > dato):
        nodo.sig = lista.inicio
        lista.inicio = nodo
    else:
        ant = lista.inicio
        act = lista.inicio.sig
        while(act is not None and act.info < dato):
            ant = ant.sig
            act = act.sig
        nodo.sig = act
        ant.sig = nodo
    lista.tamano += 1

```

```

def lista_vacia(lista):
    """Devuelve true si la lista esta vacia."""
    return lista.inicio is None

```

Figura 3. Interfaz o eventos del TDA lista parte 1

```

def eliminar(lista, clave):
    """Elimina un elemento de la lista y lo devuelve si lo encuentra."""
    dato = None
    if(lista.inicio.info == clave):
        dato = lista.inicio.info
        lista.inicio = lista.inicio.sig
        lista.tamano -= 1
    else:
        anterior = lista.inicio
        actual = lista.inicio.sig
        while(actual is not None and actual.info != clave):
            anterior = anterior.sig
            actual = actual.sig
        if (actual is not None):
            dato = actual.info
            anterior.sig = actual.sig
            lista.tamano -= 1
    return dato

```

Figura 4. Interfaz o eventos del TDA lista parte 2

```

def tamanio(lista):
    """Devuelve el numero de elementos en la lista."""
    return lista.tamano

def buscar(lista, buscado):
    """Devuelve la direccion del elemento buscado."""
    aux = lista.inicio
    while(aux is not None and aux.info != buscado):
        aux = aux.sig
    return aux

def barrido(lista):
    """Realiza un barrido de la lista mostrando sus valores."""
    aux = lista.inicio
    while(aux is not None):
        print(aux.info)
        aux = aux.sig

```

Figura 5. Interfaz o eventos del TDA lista parte 3

Cuando tenemos que agregar un nuevo elemento a la lista se crea un nodoLista, en el cual a su campo información se le asigna el elemento ingresado como dato. Luego se procede a enlazar el nodo a la estructura donde se presentan dos casos: si el elemento a agregar es el primero de la lista o es menor que el primero de la lista se le debe asignar al campo siguiente del nodo creado la dirección de inicio de la lista. Y al inicio de la lista se le asigna la dirección del nodo creado. En el caso contrario –es decir cuando es un nodo intermedio o final– se utilizan dos punteros, que denominaremos anterior y actual, para barrer la lista y determinar la posición donde debe ser enlazado. Una vez que esté determinada la posición de inserción del nuevo nodo se procede a reconstruir el enlace de los nodos, al campo siguiente del nodo se le asigna la dirección de actual y al campo siguiente de anterior se le asigna la dirección del nodo creado. Por último independientemente del caso que ocurra se incrementa el valor de tamaño.

En cambio, cuando se va a eliminar un elemento de la lista, si este es el primero de la misma, se saca del nodo que está en el inicio la información y se almacena en una variable dato. Luego al inicio se le asigna la dirección almacenada en el campo siguiente del puntero inicio –este es el caso más simple-. En el caso de no ser el primero se recorre la lista en busca del elemento a eliminar –es decir el que concuerde con la clave–, con dos punteros anterior y actual al igual que en la inserción. Si el elemento que se desea quitar es encontrado –es decir si actual es distinto de *None*– se saca el valor del campo información del nodo apuntado por actual y se almacena en una variable dato, luego se le asigna al atributo siguiente del nodo apuntado por anterior la dirección del campo siguiente del nodo apuntado por actual para poder reconstruir el enlace. En ambos casos, si se encuentra un elemento que coincide con la clave se decrementa el tamaño. Y por último se devuelve el valor de la variable dato –por defecto si no se encuentra el valor de dato será *None*–.

Por su parte, para buscar un elemento en la lista se utiliza un puntero auxiliar al que se le asigna la dirección del inicio de la lista, luego mientras el puntero auxiliar sea distinto de *None* y el campo información de auxiliar sea distinto de clave que se está buscando, a auxiliar se le asigna la dirección de campo siguiente del puntero auxiliar. Finalmente se devuelve el puntero auxiliar. Si este es distinto de *None*, la clave fue encontrada y auxiliar tiene la dirección del nodo que la contiene, caso contrario la clave no fue encontrada.

Continuando ahora en la figura 6 se observa un ejemplo del uso del TDA lista para resolver un problema. En esta ocasión se tiene una lista de palabras no repetidas y se pretende determinar si una palabra dada está en dicha lista. De ser así habría que eliminarla y finalmente realizar un barrido de la lista:

```
from tda_lista import Lista, insertar, buscar, eliminar, barrido

lista = Lista()
dato = input('Ingrese una palabra ')

while(dato != ''):
    insertar(lista, dato)
    dato = input('Ingrese una palabra ')

buscado = input('Ingrese la palabra a buscar ')
posicion = buscar(lista, buscado)

if(posicion is not None):
    dato = eliminar(lista, posicion.info)
    print('Elemento eliminado: ', dato)
else:
    print('No se econtró el elemento a eliminar')

barrido(lista)
```

Figura 6. Ejemplo de uso del TDA Lista

Siguiendo con el ejemplo, vamos a realizar una descripción detallada de las actividades llevadas a cabo para la resolución del ejercicio anterior, para interpretar la dinámica del uso del TDA lista usando los eventos definidos en la etapa de diseño.

Para comenzar importamos del TDA lista las funciones que necesitamos utilizar, después se crea una variable de tipo lista que vamos a usar para proceder a la carga de datos. Luego ingresamos el dato a buscar en la lista y se busca dicho valor, seguidamente se determina si el dato buscado está en la lista. De ser así, se pasa a eliminarlo y mostrarlo, para finalmente realizar un barrido para mostrar el contenido de la lista.

Ahora pensemos en lo siguiente ¿Qué pasa cuando el dominio del dato utilizado no es un dato simple? es decir es un registro representado con una clase, al momento de realizar las operaciones de inserción, eliminación y búsqueda deben hacerse por uno de estos campos, pero el criterio para cada

una de estas operaciones pueden ser distintos. Por ejemplo, si se tiene una lista de persona es probable que la inserción se haga por apellido para que queden ordenados alfabéticamente, mientras que para la eliminación esto puede hacerse por el código de identificación no repetido –podría ser número de pasaporte o DNI–. Entonces, dichas funciones pueden ser redefinidas agregando un parámetro que denominaremos “campo” y una función extra que se llamará criterio; esto permitirá en función del parámetro ingresado como campo determinar por qué campo o criterio se realizarán las actividades mencionadas –sin tener que modificar el código de dichas funciones cuando cambia el tipo de dato–.

En este punto ustedes se preguntarán *¿Cómo se aplica entonces esta función criterio en una lista? ¿Qué debemos cambiar en nuestro TDA para que funcione?* En realidad la forma de aplicar esta función de criterio es muy sencilla y solo deberemos hacer unos pequeños y sutiles cambios en las funciones *insertar*, *eliminar* y *buscar* del TDA lista.

Por lo cual, continuando con el ejemplo anterior, cuando se cargan personas a la lista vamos a quedar ordenadas por apellido –este será nuestro criterio de inserción–.

Pero, *¿Qué pasa cuando vamos a eliminar una persona de la lista?*, no se puede hacer por el mismo campo que se cargó, ya que podríamos tener más de una persona con el mismo apellido y se eliminaría la primera que encuentra. Entonces para eliminar a una persona de la lista se debería hacerlo por el campo que identifica a cada persona únicamente –es decir por el número de pasaporte o DNI–, este será el criterio de eliminación.

¿Y cuándo tenemos que buscar una persona? En este caso cuando se realiza una búsqueda podría querer hacerlo por cualquiera de los criterios antes mencionados. Además si el dominio del problema cambia también cambian los criterios y esto implicaría que debería alterar todo el código del TDA lista para cada problema a resolver. Pero tranquilo no es tan dramático como parece, se puede resolver de una manera sencilla.

Primero hay que ver en qué consiste la función criterio, esta se puede observar en la figura 7, que recibirá dos parámetros dato y campo, y retornará el campo por el cual se debe realizar la comparación en las funciones antes mencionadas.

```
def criterio(dato, campo=None):
    """Determina el campo por el cual se debe comparar el dato."""
    dic = {}
    if hasattr(dato, '__dict__'):
        dic = dato.__dict__
    if campo is None or campo not in dic:
        return dato
    else:
        return dic[campo]
```

Figura 7. Función Criterio

Desglosada analíticamente la figura 7, primero necesitaremos determinar si el tipo de dato ingresado es un tipo de dato primitivo o una clase –utilizada como registro en este libro–. Una vez determinado esto si el dato ingresado es una clase, se debe determinar si el campo ingresado es un atributo de la misma –que indicaría el criterio con el cual se quiere realizar la comparación para los eventos de la lista–. Para esto se obtienen los atributos de la clase con una propiedad llamada “`_dict_`”, que devuelve un diccionario donde el nombre de los atributos son las claves, y los valores asociados a dichas claves son los valores almacenados en la variable.

A partir de esto se debe determinar si el campo ingresado está en el diccionario, de ser así se devuelve el valor del atributo ingresado como campo de criterio, en cualquiera de los casos contrarios. Es decir, siempre que el campo ingresado no sea un atributo del dato o que el tipo de dato sea primitivo, se devuelve el dato como entró a la función. De esta manera, se obtiene una función genérica que puede recibir datos primitivos o clases y seguir funcionando sin problemas ni necesidad de alterar el código cada vez que cambie el dominio de tipo de dato.

Entonces, ¿Dónde se aplica la función criterio en el TDA lista? Como ya se mencionó previamente se debe utilizar en los eventos insertar, eliminar, buscar y cualquier otra nueva actividad que lo requiera. Cuando se realiza una comparación del campo información, en su lugar se debe llamar a la función criterio pasándole como parámetros la variable información y el campo por el cual queremos comparar –que por defecto puede tener valor `None` para que de esta manera sea un parámetro opcional–. Esto es útil por ejemplo si es un dato primitivo, ya que ingresar el campo por el cual compara sería innecesario.

En resumen las funciones quedan redefinidas dentro del TDA de la siguiente manera como se muestra en la figura 8, 9 y 10.

```
def insertar(lista, dato, campo=None):
    """Inserta el dato pasado en la lista."""
    nodo = nodoLista()
    nodo.info = dato
    if (lista.inicio is None) or (criterio(lista.inicio.info, campo) >
        criterio(dato, campo)):
        nodo.sig = lista.inicio
        lista.inicio = nodo
    else:
        ant = lista.inicio
        act = lista.inicio.sig
        while(act is not None and criterio(act.info, campo) <
            criterio(dato, campo)):
            ant = ant.sig
            act = act.sig
        nodo.sig = act
        ant.sig = nodo
    lista.tamano += 1
```

Figura 8. Uso de la función criterio en el TDA lista parte 1

```

def buscar(lista, buscado, campo=None):
    """Devuelve la dirección del elemento buscado."""
    aux = lista.inicio
    while(aux is not None and criterio(aux.info, campo) != criterio(buscado, campo)):
        aux = aux.sig
    return aux

```

Figura 9. Uso de la función criterio en el TDA lista parte 2

```

def eliminar(lista, clave, campo=None):
    """Elimina un elemento de la lista y lo devuelve si lo encuentra."""
    dato = None
    if(criterio(lista.inicio.info, campo) == criterio(clave, campo)):
        dato = lista.inicio.info
        lista.inicio = lista.inicio.sig
        lista.tamano -= 1
    else:
        anterior = lista.inicio
        actual = lista.inicio.sig
        while(actual is not None and criterio(actual.info, campo) != criterio(clave, campo)):
            anterior = anterior.sig
            actual = actual.sig
        if (actual is not None):
            dato = actual.info
            anterior.sig = actual.sig
            lista.tamano -= 1
    return dato

```

Figura 10. Uso de la función criterio en el TDA lista parte 3

Ahora que ya conocemos los conceptos fundamentales de las listas y cómo funcionan, veamos otros tipos, uno de ellos es lista doblemente enlazada, en la cual la lista tiene dos enlaces anterior y siguiente. Esto es muy útil para poder barrer fácilmente una lista de manera ascendente o descendente, lo cual no es posible en el tipo de lista visto anteriormente. Además, como cada nodo tiene un enlace al anterior y al siguiente es posible realizar las operaciones de inserción y eliminación utilizando un solo puntero, sin necesidad de recurrir a un puntero auxiliar como vimos anteriormente –quedará a cargo del lector demostrarlo–; y por lo general también se agrega un atributo más en el TDA lista llamado fin, que es un puntero para acceder de manera rápida al último elemento de la lista. En otras palabras, para realizar operaciones con el campo anterior se usa el puntero final mientras que para operar con el campo siguiente se utiliza el puntero fin, como se observa en la figura 11.

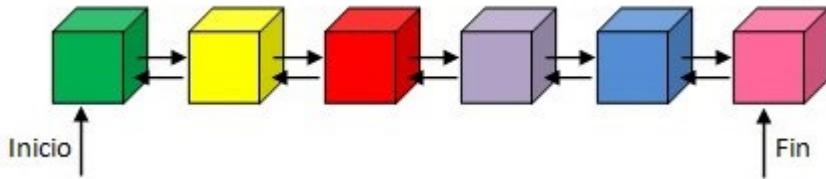


Figura 11. Esquema lista doblemente enlazada con puntero inicio y fin

Otro tipo es la lista circular. Es un caso particular de lista simplemente enlazada en la cual el atributo siguiente del último nodo de la lista en lugar de tener valor *None* tiene asignada la dirección del primer nodo de la lista, como se observa en la figura 12. Se debe tener en cuenta a la hora de barrer la lista y determinar el fin de la misma, que el ultimo nodo de la lista no tendrá valor *None* en su campo siguiente, sino que se debe preguntar si el valor siguiente del puntero auxiliar (utilizado para barrer la lista) es igual que el del inicio. De ser así habrá terminado el barrido.

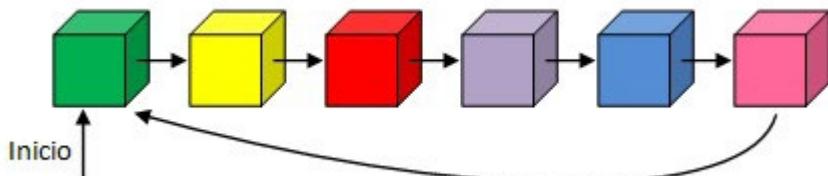


Figura 12. Esquema de lista circular

Por último tenemos el tipo lista de lista que utilizamos cuando tenemos un conjunto de elementos como veníamos viendo anteriormente, pero además cada uno de estos elementos tiene un conjunto de subelementos, es decir cada elemento de la lista tiene una lista. Por ejemplo suponga el siguiente caso donde se tienen las estaciones meteorológicas de una empresa en distintos lugares del país y cada una de esta registra los datos tiempo, incluso estas estaciones podrían registrar diferentes cantidades de veces al día –o sea que cada sublista no necesariamente tiene la misma cantidad de elementos que las otras–, como se puede observar en la figura 13.

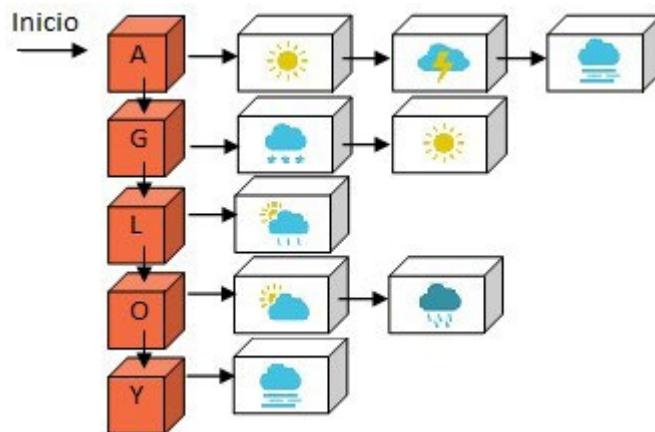


Figura 13. Estructura lista de lista dinámica

Pero ¿Cómo se implementa el TDA lista de lista? En realidad lo único que debemos hacer es definir el tipo de dato lista de lista –que será igual que la tipo de dato lista– y agregar un nuevo atributo que se denomina sublista que será de tipo lista (simplemente enlazada). Entonces gracias a la flexibilidad

del lenguaje se seguirán utilizando todas las funciones del TDA lista simplemente enlazada, solamente cambiaremos la lista que pasamos como parámetro a las funciones. En resumen podemos pasar la lista principal –o sea del tipo de lista de lista siguiendo el ejemplo la lista de las estaciones meteorológicas– o el de una de las sublistas de la lista principal –es decir la lista de los datos del clima de una de las estaciones–, como se ve en la figura 14.

```
from tda_lista import Lista, insertar, buscar, barrido

estaciones = Lista()

dato = input('Ingrese nombre de la estación: ')
insertar(estaciones, dato)

estacion = buscar(estaciones, dato)
if(estacion is not None):
    estado_clima = input('Cargar estado del clima: ')
    insertar(estacion.sublista, estado_clima)

buscado = input('Ingrese nombre de la estación a listar: ')
pos = buscar(estaciones, buscado)
if(pos is not None):
    barrido(pos.sublista)
```

Figura 14. Uso de eventos del TDA lista para la lista y sublista

Guía de ejercicios prácticos

A continuación se plantean una serie de problemas, que se deberán resolver utilizando los distintos tipos de TDA lista.

1. Diseñar un algoritmo que permita contar la cantidad de nodos de una lista.
2. Diseñar un algoritmo que elimine todas las vocales que se encuentren en una lista de caracteres.
3. Dada una lista de números enteros, implementar un algoritmo para dividir dicha lista en dos, una que contenga los números pares y otra para los números impares.
4. Implementar un algoritmo que inserte un nodo en la i -ésima posición de una lista.
5. Dada una lista de números enteros eliminar de estas los números primos.
6. Dada una lista de superhéroes de comics, de los cuales se conoce su nombre, año aparición, casa de comic a la que pertenece (Marvel o DC) y biografía, implementar la funciones necesarias para poder realizar las siguientes actividades:
 - a. eliminar el nodo que contiene la información de Linterna Verde;
 - b. mostrar el año de aparición de Wolverine;
 - c. cambiar la casa de Dr. Strange a Marvel;
 - d. mostrar el nombre de aquellos superhéroes que en su biografía menciona la palabra “traje” o “armadura”;
 - e. mostrar el nombre y la casa de los superhéroes cuya fecha de aparición sea anterior a 1963;
 - f. mostrar la casa a la que pertenece Capitana Marvel y Mujer Maravilla;
 - g. mostrar toda la información de Flash y Star-Lord;
 - h. listar los superhéroes que comienzan con la letra B , M y S ;
 - i. determinar cuántos superhéroes hay de cada casa de comic.
7. Implementar los algoritmos necesarios para resolver las siguientes tareas:
 - a. concatenar dos listas, una atrás de la otra;
 - b. concatenar dos listas en una sola omitiendo los datos repetidos y manteniendo su orden;
 - c. contar cuántos elementos repetidos hay entre dos listas, es decir la intersección de ambas;

- d. eliminar todos los nodos de una lista de a uno a la vez mostrando su contenido.
- 8. Utilizando una lista doblemente enlazada, cargar una palabra carácter a carácter, y determinar si la misma es un palíndromo, sin utilizar ninguna estructura auxiliar.
- 9. Se tiene una lista de los alumnos de un curso, de los que se sabe nombre, apellido y legajo. Por otro lado se tienen las notas de los diferentes parciales que rindió cada uno de ellos con la siguiente información: materia que rindió, nota obtenida y fecha de parcial. Desarrollar un algoritmo que permita realizar la siguientes actividades:
 - a. mostrar los alumnos ordenados alfabéticamente por apellido;
 - b. indicar los alumnos que no desaprobaron ningún parcial;
 - c. determinar los alumnos que tienen promedio mayor a 8,89;
 - d. mostrar toda la información de los alumnos cuyos apellidos comienzan con L;
 - e. mostrar el promedio de cada uno de los alumnos;
 - f. mostrar todos los alumnos que rindieron la cátedra “Algoritmos y estructuras de datos”;
 - g. indicar el porcentaje de parciales aprobados de un alumno indicado por el usuario;
 - h. indicar cuantos alumnos aprobaron y desaprobaron parciales de la cátedra “Base de datos”;
 - i. mostrar todos los alumnos que rindieron en el año 2020;
 - j. debe modificar el TDA para implementar lista de lista.
- 10. Se dispone de una lista de canciones de Spotify, de las cuales se sabe su nombre, banda o artista, duración y cantidad de reproducciones durante el último mes. Desarrollar un algoritmo que permita realizar las siguientes actividades:
 - a. obtener la información de la canción más larga;
 - b. obtener el TOP 5, TOP 10 y TOP 40 de canciones más escuchadas;
 - c. obtener todas las canciones de la banda Arctic Monkeys;
 - d. mostrar los nombres de las bandas o artistas que solo son de una palabra.
- II. Dada una lista que contiene información de los personajes de la saga de Star Wars con la siguiente información nombre, altura, edad, género, especie, planeta natal y episodios en los que apareció, desarrollar los algoritmos que permitan realizar las siguientes actividades:
 - a. listar todos los personajes de género femenino;

- b. listar todos los personajes de especie Droide que aparecieron en los primeros seis episodios de la saga;
 - c. mostrar toda la información de Darth Vader y Han Solo;
 - d. listar los personajes que aparecen en el episodio VII y en los tres anteriores;
 - e. mostrar los personajes con edad mayor a 850 años y de ellos el mayor;
 - f. eliminar todos los personajes que solamente aparecieron en los episodios IV, V y VI;
 - g. listar los personajes de especie humana cuyo planeta de origen es Alderaan;
 - h. mostrar toda la información de los personajes cuya altura es menor a 70 centímetros;
 - i. determinar en qué episodios aparece Chewbacca y mostrar además toda su información.
12. Desarrollar un algoritmo que elimine el anteúltimo nodo de una lista independientemente de la información del mismo, utilizando lista simplemente enlazada y después con lista doblemente enlazada.
13. Desarrollar un algoritmo que permita visualizar el contenido de una lista de manera ascendente y descendente de sus elementos, debe modificar el TDA para implementar lista doblemente enlazada.
14. Un grupo de amigos se reúnen a jugar un juego de dados, suponga que dichos jugadores están cargados en una lista de acuerdo a un número asignado de manera aleatoria y su nombre. Desarrollar un algoritmo que contemple las siguientes condiciones:
- a. simular la tirada de un dado –de seis lados D6– en cada turno del jugador;
 - b. el orden de turno de los jugadores es el mismo en el que están cargados en la lista;
 - c. después de que tira el último jugador de la lista debe seguir el primero;
 - d. el juego termina cuando uno de los jugadores saca un 5, en ese caso mostrar su nombre;
 - e. Debe modificar el TDA para implementar lista circular.
15. Se cuenta con una lista de entrenadores Pokémon. De cada uno de estos se conoce: nombre, cantidad de torneos ganados, cantidad de batallas perdidas y cantidad de batallas ganadas. Y además la lista de sus Pokémons, de los cuales se sabe: nombre, nivel, tipo y subtipo. Se pide resolver las siguientes actividades utilizando lista de lista implementando las funciones necesarias:
- a. obtener la cantidad de Pokémons de un determinado entrenador;
 - b. listar los entrenadores que hayan ganado más de tres torneos;

- c. el Pokémon de mayor nivel del entrenador con mayor cantidad de torneos ganados;
 - d. mostrar todos los datos de un entrenador y sus Pokémos;
 - e. mostrar los entrenadores cuyo porcentaje de batallas ganadas sea mayor al 79 %;
 - f. los entrenadores que tengan Pokémons de tipo fuego y planta o agua/volador (tipo y subtipo);
 - g. el promedio de nivel de los Pokémons de un determinado entrenador;
 - h. determinar cuántos entrenadores tienen a un determinado Pokémon;
 - i. mostrar los entrenadores que tienen Pokémons repetidos;
 - j. determinar los entrenadores que tengan uno de los siguientes Pokémons: Tyrantrum, Terrakion o Wingull;
 - k. determinar si un entrenador “X” tiene al Pokémon “Y”, tanto el nombre del entrenador como del Pokémon deben ser ingresados; además si el entrenador tiene al Pokémon se deberán mostrar los datos de ambos;
16. Se deben administrar las actividades de un proyecto de software, de estas se conoce su costo, tiempo de ejecución, fecha de inicio, fecha de fin estimada, fecha de fin efectiva y persona a cargo. Desarrollar un algoritmo que realice las siguientes actividades:
- a. tiempo promedio de tareas;
 - b. costo total del proyecto;
 - c. actividades realizadas por una determinada persona;
 - d. mostrar la información de las tareas a realizar entre dos fechas dadas;
 - e. mostrar las tareas finalizadas en tiempo y las finalizadas fuera de tiempo;
 - f. indicar cuántas tareas le quedan pendientes a una determinada persona, indicada por el usuario.
17. Se cuenta con los vuelos del aeropuerto de Heraklion en Creta, de estos se sabe la siguiente información: empresa, número del vuelo, cantidad de asientos del avión, fecha de salida, destino, kms del vuelo. Y además se conoce los datos de cantidades de asientos totales y ocupados por clase (primera y turista). Implemente las funciones necesarias que permitan realizar las siguientes actividades:
- a. mostrar los vuelos con destino a Atenas, Miconos y Rodas;
 - b. mostrar los vuelos con asientos clase turista disponible;

- c. mostrar el total recaudado por cada vuelo, considerando clase turista (\$75 por kilómetro) y primera clase (\$203 por kilómetro);
 - d. mostrar los vuelos programados para una determinada fecha;
 - e. vender un asiento (o pasaje) para un determinado vuelo;
 - f. eliminar un vuelo. Tener en cuenta que si tiene pasajes vendidos, se debe indicar la cantidad de dinero a devolver;
 - g. mostrar las empresas y los kilómetros de vuelos con destino a Tailandia.
18. Se tienen los usuarios colaboradores de un repositorio de GitHub y de cada uno de estos se tiene una lista de los *commit* realizados, de los cuales se cuenta con su *timestamp* (en formato fecha y hora), mensaje de *commit*, nombre de archivo modificado, cantidad de líneas agregadas/eliminadas (puede ser positivo o negativo) –suponga que solo puede modificar un archivo en cada *commit* que se haga–. Desarrollar un algoritmo que permita realizar las siguientes actividades:
- a. obtener el usuario con mayor cantidad de *commits* –podría llegar a ser más de uno–;
 - b. obtener el usuario que haya agregado en total mayor cantidad de líneas y el que haya eliminado menor cantidad de líneas;
 - c. mostrar los usuarios que realizaron cambios sobre el archivo *test.py* después de las 19:45 sin importar la fecha;
 - d. indicar los usuarios que hayan realizado al menos un *commit* con cero líneas agregadas o eliminadas;
 - e. determinar el nombre del usuario que realizó el último *commit* sobre el archivo *app.py* indicando toda la información de dicho *commit*;
 - f. deberá utilizar el TDA lista de lista.
19. Los astilleros de propulsores Kuat, son la mayor corporación de construcción de naves militares que provee al imperio galáctico –dentro de sus productos más destacados están los cazas TIE, destructores estelares, transporte acorazado todo terreno (AT-AT), transporte de exploración todo terreno (AT-ST), ejecutor táctico todo terreno (AT-TE), entre otros– y nos solicita desarrollar las funciones necesarias para resolver las siguientes necesidades:
- a. debe procesar los datos de las ventas de naves que están almacenados en un rudimentario archivo de texto, en el cual cada línea tiene los siguientes datos: código del astillero que lo produjo, producto (los mencionados previamente), precio en créditos galácticos, si fue construido con partes recicladas o no (booleano), quien realizó la compra (en algunos casos se desconoce quién realizó la compra y este campo tiene valor desconocido), todos estos datos están separados por “;” en cada línea del archivo;

- b. cargar los datos procesados en el punto anterior en dos listas, en la primera las ventas de las que se conocen el cliente y la segunda las que no;
 - c. el código del astillero son tres caracteres el primero en una letra mayúscula de la “A” hasta la “K” seguido de dos dígitos;
 - d. obtener el total de ingresos de créditos galácticos y cuantas unidades se vendieron;
 - e. listar los nombres de todos los clientes, los repetidos deberán mostrarse una sola vez, puede utilizar una estructura auxiliar para resolverlo;
 - f. realizar un informe de las compras realizadas por Darth Vader;
 - g. se le debe realizar un descuento del 15% a los clientes que compraron naves que fueron fabricadas con partes recicladas, mostrar los clientes y los montos a devolver a cada uno;
 - h. determinar cuánto ingreso generó la producción de naves cuyos modelos contengan la sigla “AT”.
20. Una empresa meteorológica necesita registrar los datos de sus distintas estaciones en las cuales recolecta la siguiente información proveniente de sus distintas estaciones de adquisición de datos diariamente, implemente las funciones para satisfacer los siguientes requerimientos:
- a. se deben poder cargar estaciones meteorológicas, de cada una de estas se sabe su país de ubicación, coordenadas de latitud, longitud y altitud;
 - b. estas estaciones registran mediciones de temperatura, presión, humedad y estado del clima –como por ejemplo soleado, nublado, lloviendo, nevando, etcétera– en distintos lapsos temporales, estos datos deberán guardarse en la lista junto con la fecha y la hora de la medición;
 - c. mostrar el promedio de temperatura y humedad de todas las estaciones durante el mes de mayo;
 - d. indicar la ubicación de las estaciones meteorológicas en las que en el día actual está lloviendo o nevando;
 - e. mostrar los datos de las estaciones meteorológicas que hayan registrado estado del clima tormenta eléctrica o huracanes;
 - f. debe implementar el TDA lista de lista.
21. Se cuenta con una lista de películas de cada una de estas se dispone de los siguientes datos: nombre, valoración del público –es un valor comprendido entre 0-10–, año de estreno y recaudación. Desarrolle los algoritmos necesarios para realizar las siguientes tareas:
- a. permitir filtrar las películas por año –es decir mostrar todas las películas de un determinado año–;

- b. mostrar los datos de la película que más recaudo;
 - c. indicar las películas con mayor valoración del público, puede ser más de una;
 - d. mostrar el contenido de la lista en los siguientes criterios de orden –solo podrá utilizar una lista auxiliar–:
 - I. por nombre,
 - II. por recaudación,
 - III. por año de estreno,
 - IV. por valoración del público.
22. Se dispone de una lista de todos los Jedi, de cada uno de estos se conoce su nombre, maestros, colores de sable de luz usados y especie. implementar las funciones necesarias para resolver las actividades enumeradas a continuación:
- a. listado ordenado por nombre y por especie;
 - b. mostrar toda la información de Ahsoka Tano y Kit Fisto;
 - c. mostrar todos los padawan de Yoda y Luke Skywalker, es decir sus aprendices;
 - d. mostrar los Jedi de especie humana y twi'lek;
 - e. listar todos los Jedi que comienzan con *A*;
 - f. mostrar los Jedi que usaron sable de luz de más de un color;
 - g. indicar los Jedi que utilizaron sable de luz amarillo o violeta;
 - h. indicar los nombre de los padawans de Qui-Gon Jin y Mace Windu, si los tuvieron.

Utilizando mapas para acceder rápidamente a los datos con tablas *hash*

Hasta el momento las estrategias de búsqueda que se han visto consisten en recorrer las estructuras comparando cada uno de sus elementos con lo que se está buscando –es decir se está haciendo una búsqueda por fuerza bruta–, por lo que la eficiencia de esta operación es del orden de $O(n)$. Ahora imagine una estrategia de búsqueda o aproximación a la búsqueda totalmente diferente a las vistas anteriormente, en la cual se puede evitar realizar estas sucesivas comparaciones, y que en su lugar utilice una función de mapeo $hash(k)$ que nos permita determinar la posición de un elemento k de manera directa en nuestra estructura de datos. Es decir que las operaciones de acceso para agregar o buscar elementos en la estructura son de tiempo constante del orden de $O(1)$. Pero *¿Esto es demasiado bueno para ser cierto, verdad?* De hecho todo parece muy simple y en verdad es una estructura muy sencilla de manejar, pero ya veremos más adelante que no todo es bueno y existen algunos factores que nos complicarán un poco. Peero tranquilos, encontraremos las manera de sortear estas dificultades.

En resumen, una tabla *hash*, tabla asociativa o tabla de dispersión es una colección estática de n elementos a loes que podemos acceder de manera directa utilizando una función denominada *hash* que transforma una clave (dato o parte de él) en una posición, como podemos observar en la figura 1.

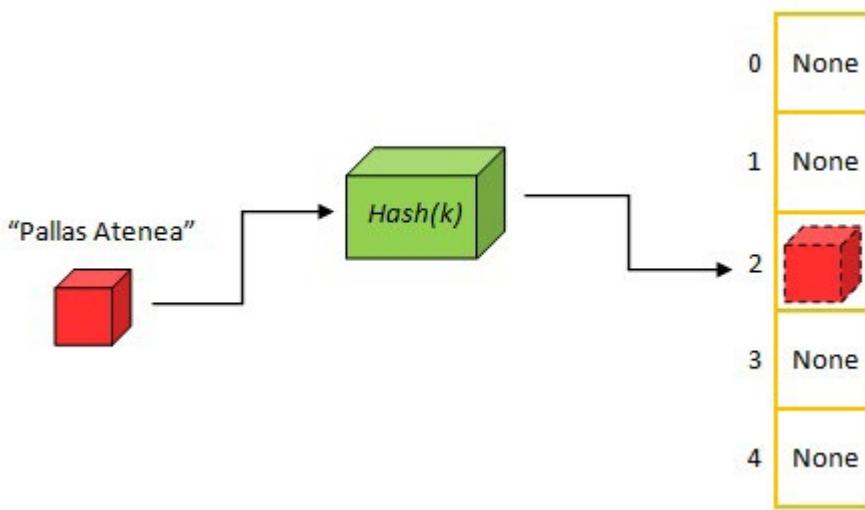


Figura 1. Esquema de una tabla *hash*

Como podemos apreciar en la figura anterior, la función *hash* no permite realizar el mapeo de un elemento transformándolo en una posición de la tabla (representada con un vector), mediante esta función podemos acceder de manera directa. Ahora observemos el ejemplo de la figura 2 en el cual

tenemos una tabla *hash* de nueve elementos: *¿Pero que hace exactamente la función hash para determinar la posición de cada nombre?* Seguramente ya lo deben haber deducido, nuestro algoritmo utiliza el operador modulo (%) para obtener el resto de la división entera entre la cantidad de caracteres de la palabra –sin contar espacios– y el tamaño de la tabla.

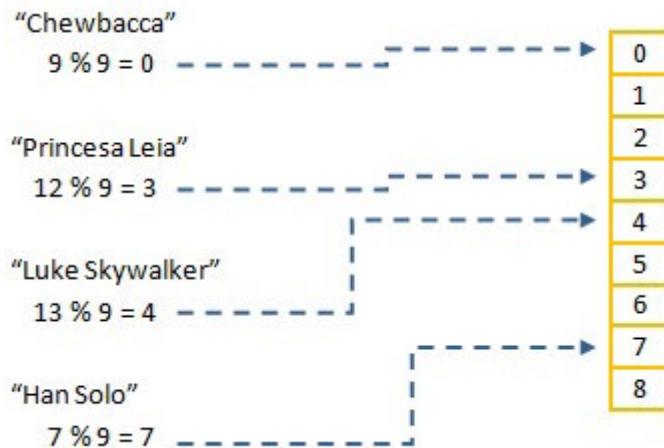


Figura 2. Funcionamiento de una tabla *hash*

Quizás el desafío más importante que enfrentaremos para utilizar esta estructura es poder encontrar esta función que transforme los datos en posiciones. Continuando con el ejemplo anterior, suponga que ahora tenemos que agregar en la tabla a "Obi-Wan Kenobi". Al aplicar la función *hash* la posición obtenida será cuatro y esto no es bueno porque ya hay un valor en esa posición –les dije que esto se complicaría un poco–, esto se conoce colisión y es muy normal que ocurra en una tabla *hash*; pero no te desesperes ya veremos cómo solucionarlo.

Si bien no parece una estructura que estemos acostumbrado a ver, está presente en nuestra vida cotidiana: por ejemplo, los diccionarios –con marcas de inicio de cada letra– que para buscar una palabra accedemos rápidamente a la sección donde está la primera letra de la palabra que buscamos y nos facilita encontrarla, archiveros de fichas de personas donde se almacenan por rango de letras, las guías de teléfonos donde accedemos inicialmente por localidad y luego accedemos por la letra inicial de la persona. Si lo vemos desde un enfoque informático, los códigos de seguimiento de envíos postales, los códigos de cifrado para el envío seguro de información, etcétera. Por su parte los sistemas operativos y las bases de datos relacionales las utilizan para generar índices de acceso a los archivos, por algoritmos de encriptación que generan sus códigos *hash* a partir de estas tablas. También las bases de datos NoSQL de tipo clave-valor están basadas en el concepto de *hash*.

Así que podemos definir una tabla *hash* considerando su estructura y funcionamiento: es una estructura lineal estática de datos –también llamadas de tipo diccionario– a la que accedemos directamente mediante una posición determinada por una función *hash*, la cual nos permite mapear los datos en la tabla.

Analicemos en detalle las fortalezas y debilidades de las funciones *hash*, ésta se encargará de transformar una clave –ya sea un número entero o una cadena de caracteres– en un entero (posición de la tabla). Algunos factores importantes a tener en cuenta al momento de elegir –o desarrollar– la función a utilizar: debe ser fácil de calcular –y no requerir gran cantidad de cómputo–, que distribuya los valores uniformemente en la tabla, debe minimizar el número de colisiones –recordemos que

una colisión ocurre cuando dos valores distintos comparten la misma posición– y además obviamente tiene que retornar siempre la misma posición ante un mismo elemento ingresado. Por lo general, estas funciones suelen utilizar el operador módulo entre el valor obtenido por la función *hash* y el tamaño de la tabla para garantizar que la posición obtenida no exceda el tamaño de dicha tabla.

Existen distintos tipos de funciones *hash* que detallaremos a continuación, comenzemos con el *hash* por división el cual consiste en tomar el resto de la división entre la clave y el número de elementos de la tabla. Es recomendable que sea un número primo no cercano a potencias de 2 o de 10 para que no dependa de los bits menos representativos de la clave. De esta manera la fórmula general de la función *hash* de división queda determinada de la siguiente manera:

$$h(k) = k \% m$$

Donde % es el operador módulo que representa el resto de la división entera entre dos números y el valor de k se puede obtener de distintas maneras: puede ser la cantidad de caracteres, o la suma de sus caracteres en código ASCII, etcétera.

Por su parte el *hash* por multiplicación requiere algunas operaciones más para hacer el cálculo de la posición, primero se debe multiplicar la clave por una constante A , que está en el rango de $0 < A < 1$, para luego extraer la parte fraccionaria del resultado y seguido de esto es multiplicarlo por m que representa un número significativo en función del número de elementos de la tabla –una buena opción es 2^n , donde n es la cantidad de elementos de la tabla o bien 2^{32} –, finalmente del resultado obtenido será un número mayor que el tamaño de la tabla por lo que tendremos que aplicar el operador modulo entre dicho valor y el tamaño de la tabla, dicho valor será la posición donde deberá localizarse el elemento en la tabla. Por lo que la fórmula general de la función *hash* por multiplicación queda de la siguiente manera:

$$h(k) = m * \text{parte_fraccionaria}(A * k)$$

Por lo que solo falta determinar el valor de A , para esto normalmente se utiliza la fórmula propuesta por Donald Knuth¹ (1973, p.516). En esta, el valor de A se obtiene de la siguiente manera:

$$A = \frac{\alpha}{\omega}$$

Donde el valor de α proviene del siguiente cálculo:

$$\alpha = 2^{32} * \left(\frac{-1 + \sqrt{5}}{2} \right)$$

Cuyo resultado aproximado es $\alpha \approx 2\ 654\ 435\ 769,497\ 2305$, dependiendo de las especificaciones del lenguaje en que se programe, y $\omega = 2^{32}$, cuyo resultado es $\omega = 4\ 294\ 967\ 296$. De esta forma, finalmente se obtiene que el valor sea aproximadamente $A \approx 0,6180339887498949$. Se ha demostrado que el uso de este valor reduce notablemente el número de colisiones de la función *hash*.

¹ Knuth, D. E. (1973). The Art of Computer Programming. Volume III. Searching and Sorting. Addison-Wesley.

Finalmente, presentaremos la función de Bernstein, que se suele utilizar cuando se requiere que para dos claves alfanuméricas similares la función *hash* obtenga posiciones alejadas entre sí. Para lograr esto, Bernstein propone multiplicar el valor de *h* por 33 y luego sumarle el valor en código ASCII del primer carácter guardando el nuevo resultado en *h*. Luego repite el mismo procedimiento para cada carácter de la cadena –la variable *h* representara el valor de la posición asociada a la cadena–. El número utilizado por Bernstein para multiplicar (el 33), es conocido como el número mágico, dado que se ha probado que da muy buenos resultados cuando se lo utiliza, pero aún no se ha demostrado el porqué. Probablemente el valor de *h* obtenido por esta función sea mayor que el tamaño de la tabla, por lo que se deberá obtener el resto de la división entera entre *h* y el tamaño de la tabla como se mencionó antes. En la figura 3 se observa la implementación de dicha función:

```
def bernstein(cadena):
    """Función hash de Bernstein para cadenas."""
    h = 0
    for caracter in cadena:
        h = h * 33 + ord(caracter)
    return h
```

Figura 3. Función *hash* de Bernstein para cadenas de texto

Ahora volvamos un poco para atrás para entender los tipos de tablas *hash*, para esto comencemos con el caso más sencillo llamado tabla *hash* cerrada o de dirección directa, este se presentó previamente en la figura 2. Se usa cuando el universo de claves es relativamente pequeño, de tal forma que cada clave o elemento es almacenado en una posición de la tabla y cuando ocurren colisiones –y sabemos que ocurrirán–, es necesario hacer un sondeo sobre la tabla para determinar la nueva posición del elemento y desplazar, de este modo, dicho elemento a una posición libre de la tabla.

Una manera sencilla de manejar las colisiones es buscar otra posición libre utilizando una función de sondeo que determinará la nueva ubicación del elemento. Veamos las funciones de sondeo más utilizadas para la resolución de colisiones:

Sondeo lineal: en este caso se busca la siguiente posición contigua libre en la tabla, es el método de mejor rendimiento de caché pero es el más propenso a generar aglomeramientos. El intervalo entre cada intento es constante generalmente 1, pero es de esperar que el tiempo de acceso total a la estructura esté en el orden de $O(1/(1-\alpha))$ donde α representa el factor de carga de la tabla y se obtiene con la siguiente fórmula $\alpha = m/n$ donde m es el número de claves ingresadas y n el tamaño de la tabla.

Doble *hash*: cuando ocurre una colisión se ejecuta una segunda función de *hash*, para determinar la nueva localización del elemento en la tabla. Al requerir realizar más cálculos el rendimiento es menor, pero elimina el aglomeramiento.

Sondeo cuadrático: para este caso particular los índices se incrementan de manera cuadrática. En cuanto a rendimiento este sondeo se ubica entre los dos mencionados previamente, y disminuyen el aglomeramiento que se genera en el sondeo lineal.

Dejemos de lado las funciones de sondeo para poder entender *¿Qué es un aglomeramiento?* El aglomeramiento es un problema grave para las tablas *hash* –les dije anteriormente que aparecerían algunas complicaciones–, y ocurre cuando todos los datos están concentrados en una parte de la tabla, en cambio si los datos están distribuidos por toda la tabla podemos decir que nuestros datos están bien espaciados, estas dos situaciones se pueden ver en la figura 4. Si se produce un aglomeramiento en la tabla *hash* es un claro indicador de que nuestra función *hash* tiene un problema –puede que la función utilizada no sea la indicada para el tipo de dato de los elementos–, por eso es esencial para el funcionamiento de la estructura la función *hash* que desarrollemos.



Figura 4. Distribuciones de datos en una tabla *hash*

Además, este tipo de tabla tiene buen rendimiento cuando se encuentra por debajo del 75% de su capacidad, dado que cuando supera dicho valor el número de saltos requeridos por la función sondeo para determinar la nueva posición aumenta considerablemente y por ende su rendimiento decae rápidamente.

Pero *¿Qué sucede cuando se excede el umbral de elementos de la tabla?*, es decir se supera el 75% de su capacidad y necesitamos seguir agregando elementos. En este caso, debemos hacer un *rehashing* de ella para seguir manteniendo la eficiencia de acceso. Esto implica crear una nueva tabla de mayor tamaño y volver a ingresar todos los elementos que estaban en la tabla anterior, dado que, al cambiar el tamaño de la tabla, las posiciones determinadas por la función *hash* variarán y todos los elementos deberán ser reubicados –obviamente esto es una operación costosa en cuanto al tiempo de ejecución pero necesaria para mejorar la eficiencia de la tabla–.

Cuando quitamos un elemento se deben contemplar dos situaciones: la primera cuando se elimine el elemento se debe corroborar con la función de sondeo si existen otros elementos en la tabla que coincida con la clave del eliminado –en este caso, se deben desplazar a la izquierda los elementos que colisionaron en la posición del elemento eliminado y fueron reubicados–. La segunda ocurre cuando en la posición determinada por la función *hash* no se encuentre el dato que se pretende eliminar, esto puede suceder si al momento de insertarlo ocurrió una colisión y fue necesario utilizar una función de sondeo. Entonces, debemos realizar la misma acción para localizarlo en la tabla para posteriormente quitarlo –o en su defecto si no lo encontramos es porque no está–.

Por ejemplo, supongamos que tenemos la siguiente tabla “tabla A”, en la cual hemos almacenado las claves en la posición que nos determina una función *hash*, como puede observarse en la figura 5. Nótese que las claves *F*, *H* y *W* comparten el mismo valor de la función *hash* (índice 4), y se realizó un sondeo lineal para determinar las nuevas posiciones. Si eliminamos la clave *H* y simplemente dejamos el espacio vacío “tabla B”, cuando busquemos *W* la función *hash* devolverá 4, pero ahí se encuentra *F*. Entonces, buscará en la posición siguiente y encontrará vacío y determinara que no se encuentra *W*, lo cual es un error. Como mencionamos previamente la solución a este problema es que luego de eliminar *H*, se debe seguir recorriendo la tabla mientras la clave coincida con la

función *hash* y desplazar los elementos por la tabla para quitar los espacios vacíos “tabla C”; ahora, si buscamos la clave *W*, la podremos encontrar.

Clave	funcion_hash()	A	B	C
A	0	0 A	0 A	0 A
D	8	1 P	1 P	1 P
F	4	2	2	2
H	4	3 F	3 F	3 F
P	1	4 H	4	4 W
W	4	5 W	5 W	5
Z	9	6	6	6 D
		7	7	7 Z
		8	8	8
		9	9	9

Figura 5. Eliminar elemento en tabla *hash* cerrada

Otra alternativa para manejar las colisiones –y evitar el problema del aglomeramiento generado por las funciones de sondeo–, sería almacenar varios elementos en una misma posición de la tabla y esto lo lograremos encadenando elementos. Las tablas *hash* abiertas o encadenadas utilizan una técnica de encadenamiento para solucionar las colisiones, es decir, se utiliza una estructura de datos auxiliar para agrupar todos los elementos que colisionan en la misma posición de la tabla, como se muestra en la figura 6. Por lo general, se utilizan listas enlazadas o árboles binarios de búsqueda si el tiempo de acceso es crítico (estos últimos los veremos en el próximo capítulo).

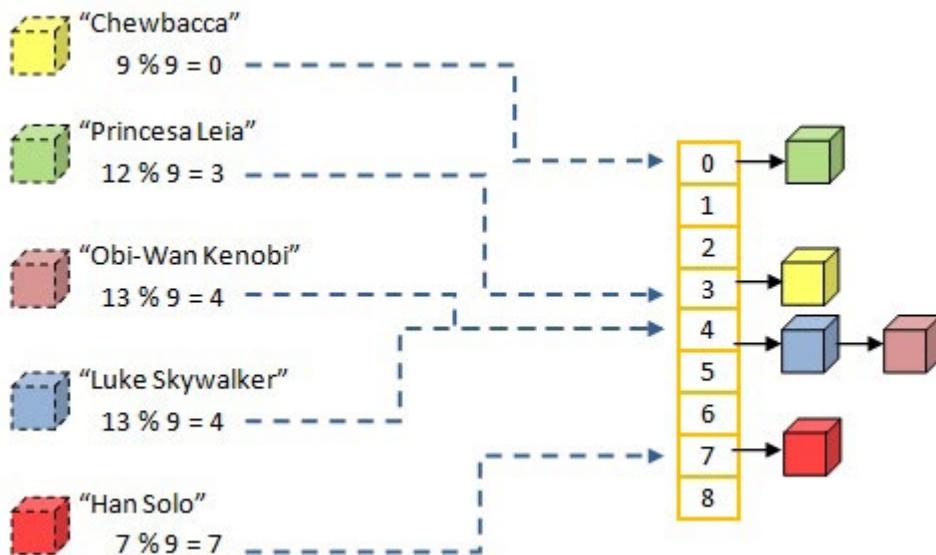


Figura 6. Tabla *hash* abierta o encadenada

Para que estas tablas funcionen eficientemente depende exclusivamente de la función *hash* utilizada. Esta debe distribuir los elementos de manera uniforme por la tabla, como mencionamos previamente, para que cada lista enlazada mantenga un tamaño similar y el orden de acceso a cada una de estas sea semejante. Por ejemplo si tenemos una tabla de m posiciones e insertamos n elementos, el coste de acceso a cada posición debería estar en el orden de $O(1 + \alpha)$ –si los elementos están distribuidos uniformemente–, donde α representan el factor de carga de los casilleros de la tabla y se calcula $\alpha = n/m$. En cambio, si los elementos quedaron concentrados en un par de casilleros de la

tabla el coste de acceso podría llegar a estar en el orden de $O(n)$. Una gran ventaja que tiene este tipo de tabla es que al momento de eliminar un elemento es mucho más simple dado que tanto las listas enlazadas como los árboles tienen operaciones para quitar elementos sin necesidad de recurrir a desplazamientos que afecten el rendimiento. Además el crecimiento de la tabla *hash* encadenada puede ser pospuesto por mucho más tiempo dado que el degradamiento es lineal si se utilizan listas enlazadas, o logarítmico si se utilizan árboles.

Analicemos el siguiente ejemplo: el DLE tiene 93 000 palabras, si utilizaremos una lista enlazada para almacenar todas las palabras y luego si tenemos que buscar una palabra en dicha estructura el costo es del orden de $O(n)$, es decir 93 000, pero si en su lugar utilizamos una tabla *hash* encadenada de 28 posiciones –una para cada letra del abecedario– y supongamos que la cantidad de palabras que empieza con cada letra es aproximadamente similar –más allá que no lo sea– el costo de buscar una palabra en este caso es del orden de $O(1 + \alpha)$, esto es igual a $(1 + 93\ 000/28)$ aproximadamente 3 322, lo cual implica una diferencia significativa respecto a la eficiencia de ambas estructuras.

Ahora que ya tenemos un panorama general respecto de las tablas *hash* es momento de adentrarnos en la implementación. El TDA tabla *hash* estará compuesto básicamente de un vector de n elementos al que denominaremos “tabla”. Y además contará con un conjunto de funciones que describen su comportamiento, las cuales se enumeran a continuación:

1. `crear_tabla(tamaño)`. Crea y devuelve una tabla *hash* vacía con la cantidad de elementos, determinado por el tamaño ingresado;
2. `agregar(tabla, dato)`. Agrega el elemento a la tabla en la posición determinada por la función *hash*, si se produce una colisión se deberá reubicar el elemento con una función de sondeo y si es una tabla encadenada deberá utilizar una función extra para insertarla en la estructura auxiliar;
3. `quitar(tabla, dato)`. Elimina y devuelve el elemento de la tabla en la posición indicada por la función *hash*, puede requerir además aplicar la función utilizada en la inserción para resolver colisiones para encontrar dicho elemento o una función extra si es una tabla encadenada. Si devuelve *None* significa que no se encontró el dato en la tabla –y por ende no se elimina ningún elemento–;
4. `buscar(tabla, dato)`. Devuelve la posición de la tabla en la que se encuentra el elemento buscado, puede requerir utilizar la función utilizada en la inserción para resolver las colisiones –si es una tabla *hash* cerrada– o una función extra si es una tabla *hash* encadenada. Si devuelve *None* significa que no se encontró la clave en la lista;
5. `funcion_hash(dato, tamaño_tabla)`. Devuelve la posición que le corresponde al dato en el vector. Es probable que se disponga de más de una función *hash* dado que no se utilizará siempre la misma y dependerá del dominio de los datos.
6. `sondeo(posicion, tamaño_tabla)`. Devuelve la nueva posición que le corresponde al dato en la tabla, para poder resolver las colisiones, o estructura auxiliar que permita resolver las colisiones que ocurran.

7. cantidad_elementos(tabla). Devuelve la cantidad de elementos en la tabla, puede requerir utilizar alguna función extra si es una tabla encadenada para contar los elementos de la estructura auxiliar.

A diferencia de los TDA anteriores no es necesario definir una estructura de datos –dado que solo se utilizará un vector–, por lo que en las figuras 7, 8, 9 y 10 se puede observar la implementación de las funciones mencionadas previamente, nótese que varias funciones están dos veces dado que se presentan las alternativas para tabla *hash* cerrada y encadenada –incluso algunas están incompletas–; además solo se presenta una función *hash* y quedará a cargo del lector desarrollar las otras como también las funciones de sondeo para completar las que no están completas. Estos serán los eventos con los que contaremos para trabajar con las tablas *hash*.

```

def crear_tabla(tamano):
    """Crea una tabla hash vacía."""
    tabla = [None] * tamano
    return tabla

def cantidad_elementos(tabla):
    """Devuelve la cantidad de elementos en la tabla."""
    return len(tabla) - tabla.count(None)

def cantidad_elementos(tabla):
    """Devuelve la cantidad de elementos en la tabla."""
    return sum(tamano(lista) for lista in tabla if lista is not None)

def funcion_hash(dato, tamano_tabla):
    """Determina la posición del dato en la tabla."""
    # hash por división para este caso
    return len(str(dato).strip()) % tamano_tabla

```

Figura 7. Interfaz o eventos del TDA tabla *hash* parte I

```

def agregar(tabla, dato):
    """Agrega un elemento a la tabla encadenada."""
    posicion = funcion_hash(dato, len(tabla))
    if(tabla[posicion] is None):
        tabla[posicion] = Lista()
    insertar(tabla[posicion], dato)

def agregar(tabla, dato):
    """Agrega un elemento a la tabla cerrada."""
    posicion = funcion_hash(dato, len(tabla))
    if(tabla[posicion] is None):
        tabla[posicion] = dato
    else:
        print('se produjo una colisión')
        # ejecutar función de sondeo para reubicar elemento

```

Figura 8. Interfaz o eventos del TDA tabla hash parte 2

```

def buscar(tabla, buscado):
    """Determina si un elemento existe en la tabla y determina su posición."""
    pos = None
    posicion = funcion_hash(buscado, len(tabla))
    if(tabla[posicion] is not None):
        pos = busqueda(tabla[posicion], buscado)
    return pos

def buscar(tabla, buscado):
    """Determina si un elemento existe en la tabla y determina su posición."""
    pos = None
    posicion = funcion_hash(buscado, len(tabla))
    if(tabla[posicion] is not None):
        if(buscado == tabla[posicion]):
            pos = posicion
        else:
            print('aplicar función de sondeo')
            # para determinar si esta en otra posición
    return pos

```

Figura 9. Interfaz o eventos del TDA tabla hash parte 3

```

def quitar(tabla, dato):
    """Quita un elemento de la tabla encadenada si existe."""
    dato = None
    posicion = funcion_hash(dato, len(tabla))
    if(tabla[posicion] is not None):
        dato = eliminar(tabla[posicion], dato)
        if(lista_vacia(tabla[posicion])):
            tabla[posicion] = None
    return dato

def quitar(tabla, dato):
    """Quita un elemento de la tabla cerrada si existe."""
    dato = None
    posicion = funcion_hash(dato, len(tabla))
    if(tabla[posicion] is not None):
        if(dato == tabla[posicion]):
            dato = tabla[posicion]
            tabla[posicion] = None
        else:
            print('aplicar función de sondeo')
            # para determinar si esta en otra posición y quitarlo
    return dato

```

Figura 10. Interfaz o eventos del TDA tabla *hash* parte 4

En el momento que un nuevo elemento se va cargar a la tabla se utiliza la función *hash* para determinar la posición donde debe ser agregado, luego se evalúa si dicha posición tiene valor *None* lo que implica que no hay elemento y puede ser cargado; en el caso de que ya haya un elemento cargado dependiendo del tipo de tabla, se debe realizar una de las siguientes actividades: si es una tabla cerrada, se debe aplicar la función de sondeo correspondiente para determinar la nueva posición, en algunos casos puede ser necesario repetir esta acción varias veces si ocurren muchas colisiones en una misma posición. En cambio, si la tabla utilizada es encadenada, se debe utilizar una función extra para insertar dicho elemento en la estructura auxiliar que se esté utilizando –ya sea insertar en la lista enlazada o en un árbol binario de búsqueda–, por lo que no se requiere función extra para resolver las colisiones.

En cambio, para quitar un elemento de la tabla, se calcula la posición de la tabla donde debería estar con la función *hash*, si dicha posición es distinta de *None* significa que el elemento podría estar y se pueden dar algunos de los siguientes casos dependiendo del tipo de tabla: si es una tabla cerrada, se debe verificar si el elemento en la posición determinada por la función *hash* es el que se quiere eliminar. En ese caso, se quita el elemento y se debe aplicar la función sondeo para determinar si no hay más elementos con la misma clave y reubicarlos para evitar el problema mencionado anteriormente. En cambio, si en la posición no esté el elemento que se desea eliminar se debe aplicar la función de sondeo para buscarlo en otra posición, por si ocurrieron colisiones, de igual manera que

para el otro caso. Mientras que, si es una tabla encadenada se debe utilizar una función extra para eliminar el elemento de la estructura auxiliar, pero con este tipo de tabla no existe el problema de tener que reubicar los elementos cuando han ocurrido colisiones. Cualquiera fuera el caso y tipo de tabla puede ocurrir que al intentar eliminar un elemento no lo encontremos en la tabla por lo que la función devolverá valor *None*.

Ahora, nos resta entender cómo utilizar el TDA tabla *hash*, para esto resolveremos un problema a modo de ejemplo, el cual se presenta en la figura II. En este caso particular, vamos a implementar una tabla *hash* encadenada para almacenar nombres de personajes –el cual venimos trabajando desde el principio del capítulo–, dicha tabla tendrá 9 posiciones. Sobre esta se requiere hacer una búsqueda para determinar si un personaje está o no cargado. Para este caso, usaremos una tabla encadenada, por lo que no importa si ocurren colisiones dado que los elementos se almacenan en una estructura auxiliar. La función *hash* utilizada para este caso particular es la que hemos mencionado previamente, que suma los caracteres del nombre meno los espacios para luego aplicar un *hash* por división.

```
from tda_tabla_hash import crear_tabla, agregar, buscar

tabla = crear_tabla(9)

nombre = input('ingrese nombre ')

while(nombre != ''):
    agregar(tabla, nombre)
    nombre = input('ingrese nombre ')

buscado = input('ingrese el nombre a buscar ')
posicion = buscar(tabla, buscado)
if (posicion is not None):
    print('elemento encontrado', posicion.info)
else:
    print('no se encontro el elemoneto buscado')
```

Figura II. Ejemplo de uso del TDA tabla *hash*

Ahora se desglosa analíticamente las actividades realizadas para la resolución del ejercicio anterior, y entender la manera de usar el TDA tabla *hash* usando los eventos definidos en el bloque de interfaz.

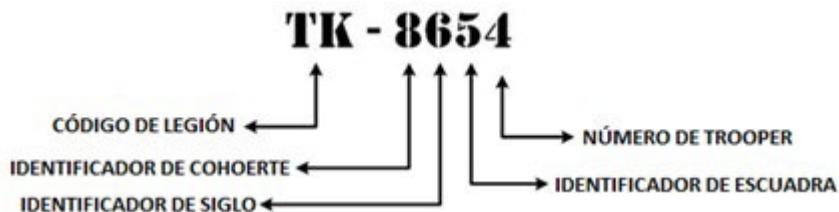
Para arrancar, se importan del TDA tabla *hash* las funciones que vamos a utilizar para dar solución al problema, después creamos una variable de tipo tabla *hash* de la dimensión indicada. Luego se cargan personajes en la tabla, una vez finalizada la carga se ingresa el nombre del personaje a buscar y se realiza la búsqueda para determinar si el personaje buscado está cargado en la tabla o no.

Guía de ejercicios prácticos

A continuación se plantean una serie de problemas, se deberá utilizar las funciones definidas en el TDA tabla *hash* para dar solución a estos.

1. Desarrollar un algoritmo que permita implementar una tabla *hash* para representar un diccionario que permita resolver las siguientes actividades:
 - a. agregar una palabra y su significado al diccionario;
 - b. determinar si una palabra existe y mostrar su significado;
 - c. borrar una palabra del diccionario;
 - d. la tabla debe tener 28 posiciones y manejar las colisiones con lista enlazadas;
 - e. mejorar el rendimiento de la tabla utilizando árboles binarios de búsqueda.
2. Desarrollar un algoritmo que implemente una tabla *hash* para una guía de teléfono, los datos que se conocen son número de teléfono, apellido, nombre y dirección de la persona. El campo clave debe ser el número de teléfono.
3. Implementar un tabla *hash* cerrada para guardar las cátedras de una carrera universitaria de acuerdo a su código, que permita resolver las siguientes actividades:
 - a. cargar cátedras de una carrera de las cuales se conoce nombre, modalidad (anual o cuatrimestral), cantidad de horas;
 - b. además se deben poder agregar los docentes vinculados con las cátedras;
 - c. debe ser una tabla cerrada;
 - d. debe poder solucionar las colisiones;
 - e. no podrán estar cargadas de manera correlativa de acuerdo a un número.
4. Desarrollar un algoritmo que implemente una tabla *hash* para cargar personajes de Star Wars de los que solo se conoce su nombre, contemplando las siguientes actividades:
 - a. la tabla inicialmente será de 20 posiciones;
 - b. deberá permitir el manejo de colisiones;
 - c. cuando el factor de carga de la tabla exceda el 75%, se deberá incrementar el tamaño de la tabla al doble y hacer un *rehashing* de las claves cargadas.

5. Desarrollar un algoritmo que implemente una tabla *hash* cerrada para administrar los contactos de personas de las cuales se conoce nombre, apellido y correo electrónico, contemplando las siguientes pautas:
 - a. El campo clave para generar las posiciones son el apellido y nombre.
 - b. Deberá contemplar una función de sondeo para resolver las colisiones.
6. Los *Stormtrooper* del imperio galáctico se identifican de la siguiente manera:



Darth Vader le encarga desarrollar los algoritmos para organizar los *Stormtrooper* cumpliendo con las siguientes demandas:

- a. Deberá generar 2000 *Stormtrooper* siguiendo el formato de la imagen anterior contemplando las siguientes legiones FL, TF, TK, CT, FN, FO y los dígitos generados de manera aleatoria;
 - b. deberá cargar los *Stormtrooper* generados en dos tablas *hash* encadenadas, en la primera se deberá agrupar de acuerdo a los tres últimos dígitos del código y en la segunda a partir de las iniciales de la legión;
 - c. determinar si el *Stormtrooper* FN-2187 está cargado para poder quitarlo porque es un traidor desertor.
 - d. ahora obtenga todos los *Stormtrooper* terminados en 781 para asignarlos a una misión de asalto y a los terminados en 537 para una misión de exploración;
 - e. ahora obtenga los *Stormtrooper* de la legión CT para que custodien a Darth Vader a una misión de exploración al planeta Hoth y los de la legión TF para una misión de exterminación a Endor.
7. Escribir un algoritmo que permita utilizar tres tablas *hash* para guardar los datos de Pokémons, que contemple las siguientes actividades:
 - a. en la primera tabla *hash* la función *hash* debe ser sobre el tipo de Pokémon, en la segunda tabla la función *hash* deberá utilizar el número del Pokémon como clave, mientras que en la tercera el campo clave de la función hash será por el nombre del Pokémon.
 - b. el tamaño de la primera tabla debe ser lo suficientemente grande como para que pueda almacenar todos los distintos tipos de Pokémon, debe manejar las colisiones con alguna función de sondeo;

- c. el tamaño de cada una de las segundas tablas debe ser 15;
 - d. el algoritmo debe permitir cargar tipos de Pokémon en la primera tabla y crear su respectiva segunda tabla, –en el caso de que no exista–;
 - e. si el Pokémon es de más de un tipo deberá cargarlo en cada uno de las tabla que indiquen estos tipos;
 - f. deberá permitir cargar Pokémons de los cuales se dispone de su número, nombre, tipo, nivel.
8. La alianza rebelde necesita comunicarse de manera segura pero el imperio galáctico interviene todas la comunicaciones, por lo que la princesa Leia nos encarga el desarrollo de un algoritmo de encriptación para las comunicaciones rebeldes, que contemple los siguientes requerimientos:
- a. cada carácter deberá ser encriptado a ocho caracteres;
 - b. se deberá generar dos tablas *hash* para encriptar y desencriptar, para los caracteres desde el “” hasta el “}” –es decir desde el 32 al 125 de la tabla ASCII.
9. Desarrollar un algoritmo que permita cifrar y descifrar un mensaje carácter a carácter, contemplando las siguientes pautas:
- a. Se debe utilizar una tabla *hash* para guardar los valores de codificación y decodificación respectivamente que se vayan utilizando.
 - b. Se deberá cifrar de la siguiente manera: primero, convertir al valor numérico correspondiente de la tabla ASCII cada carácter y luego, cada número de dicho valor se deberá remplazar por su valor correspondiente según los siguientes valores: 1 – “abd”, 2 – “def”, 3 – “ghi”, 4 – “jkl”, 5 – “mnñ”, 6 – “opq”, 7 – “rst”, 8 – “uvw”, 9 – “xyz”, 0 – “#?&”, y se debe agregar al final el carácter %. Por ejemplo D = 68 debería quedar de la siguiente manera “opquvw%”.
10. Implementar una tabla *hash* para almacenar la información de todos los Jedi, de los cuales se conoce su nombre y quien fue su maestro –este último puede ser más de uno o desconocido– contemplando las siguientes requerimientos:
- a. la tabla debe ser de 15 posiciones;
 - b. debe poder manejar las colisiones que se produzcan dependiendo del tipo de tabla utilizado;
 - c. cargar al menos 30 Jedi;
 - d. determinar si están cargados los Jedi: Yoda, Luke Skywalker y Ahsoka Tano. Si no están, agregarlos;
 - e. crear una función que permita determinar el factor de carga de la tabla, dependiendo del tipo utilizado;

- f. mostrar toda la información de Ahsoka Tano, Obi-Wan Kenobi y Qui-Gon Jin;
- g. mostrar los maestros y aprendices (padawan) de Yoda y Luke Skywalker; los aprendices no son parte de la información, debe determinarlos a partir del campo maestro de cada Jedi.
- II. Nick Fury director de la agencia S.H.I.E.L.D. intenta detener a la organización Hydra y a su líder Red Skull, los agentes de la agencia pueden interceptar los mensajes de Hydra pero están cifrados, por tanto no pueden hacer nada con estos; afortunadamente el Capitán América en una misión encubierta logró determinar las pautas del método de codificación. Ahora Fury nos solicita desarrollar el algoritmo que permita decodificar los mensajes, contemplando las siguientes pautas:
- Las codificación se realiza de la siguiente manera:
 - primero se convierte el carácter a su valor en la tabla ASCII y se lo multiplica por 37 para transformarlo en un número de cuatro dígitos;
 - segundo se calcula un complemento en base al valor del carácter:
$$\text{complemento}(\text{caracter}) = \begin{cases} 79 + \text{caracter} - 32, & \text{caracter} \leq 78 \\ 32 + \text{caracter} - 79, & \text{caracter} > 78 \end{cases}$$
 - luego a cada digito obtenido en el punto uno se lo eleva al cuadrado y se le suma un complemento obtenido en el punto anterior y se transforma a carácter;
 - por último se juntan los cuatro caracteres y se le agrega al final el carácter correspondiente al complemento.

Por ejemplo el carácter R se codifica de la siguiente manera:
 $R = 82, 82 * 37 = 3034$, complemento $= 32 + 82 - 79 = 35 = "#"$
 $3^2 + 35 = 44 = ", 0^2 + 35 = 35 = "#, 3^2 + 35 = 44 = ", 4^2 + 35 = 51 = "3"$
 El resultado final son estos cinco caracteres ",#,3#";

 - deberá utilizar una tabla hash cerrada para almacenar cada una de las cadenas de caracteres –de cinco caracteres– asociados a cada clave, una buena alternativa para la función hash podría ser la función de Bernstein;
 - no se debe decodificar todas las cadenas de caracteres, esto debe hacerse a medida que se necesitan y no están en la tabla;
 - ayuda al Capitán América descifrando los siguientes tres mensajes para poder conocer cuáles serán los próximos movimientos de Hydra (los mensajes están almacenados en archivos de texto, que deberá leerlos previamente desde cada archivo, en el siguiente link: https://github.com/belwalter/mensajes_codificados).

Desde la raíz hasta las hojas, entendiendo los árboles desde otro punto de vista

Ahora es el momento de comenzar a estudiar las estructuras ramificadas, cuya principal diferencia con las lineales –donde cada elemento de la estructura tiene un anterior y un siguiente–, en las estructuras ramificadas cada elemento puede relacionarse con muchos elementos. En este caso, nos centraremos en la estructura de datos árbol y los distintos tipos que existen. Cuando el tamaño de la entrada de nuestro problema es muy grande, y el tiempo de acceso lineal de orden $O(n)$ que tiene con el uso de listas enlazadas no es una opción viable, es necesario recurrir a estructuras arbolesas que nos otorgan en promedio un orden de $O(\log n)$. De estas, haremos énfasis al modelado de un caso particular que nos garantice –en el peor caso– un orden de $O(\log n)$ para las operaciones de *inserción, eliminación y búsqueda*. Esta estructura es el árbol binario de búsqueda balanceado.

Un árbol es una colección de datos jerárquica, en el que cada nodo tiene un parente a excepción del nodo raíz, que representa la jerarquía máxima sobre el resto de los elementos. Los árboles son una estructura de datos recursiva ya que está compuesta por árboles más pequeños llamados subárboles, es decir los hijos de un nodo son la raíz de los subárboles de dicho nodo. Los hijos de un nodo están conectados mediante una rama dirigida padre-hijo, es decir que desde los hijos no se puede acceder al parente. Los árboles en la vida real sin importar su tipo crecen a partir de sus raíces, en las estructuras de datos árboles pasa lo mismo, siempre deben tener una raíz, de hecho un solo nodo es un árbol. Los árboles se dibujan al revés con la raíz en la parte superior como se observa en la figura 1.

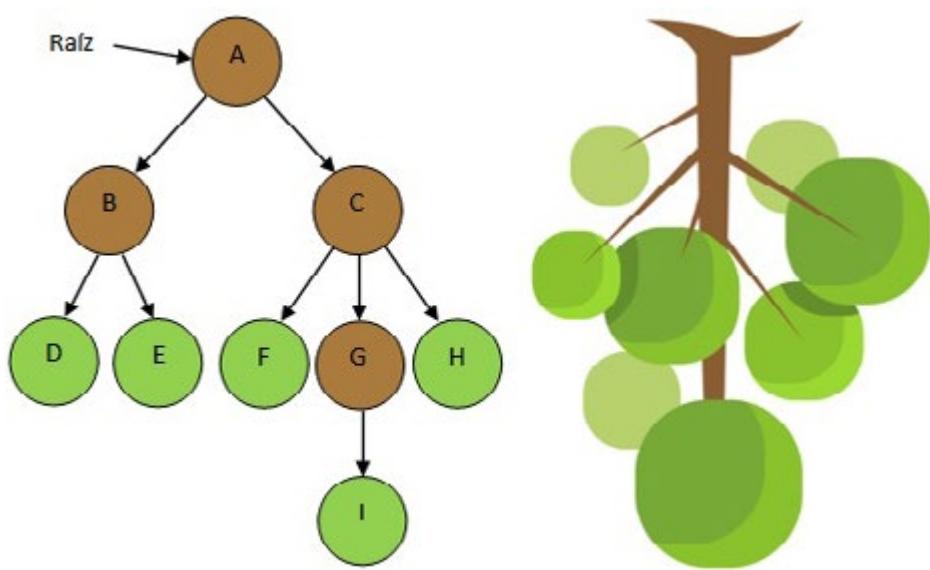


Figura 1. Estructura de un árbol

Tomando como ejemplo el árbol de la figura 1 detallaremos la terminología básica referida a árboles para poder comprender los conceptos que se desarrollarán a lo largo del capítulo:

Raíz: es el nodo que se encuentra en la parte superior del árbol, mediante el cual se accede al árbol. En este caso el nodo A.

Hijo: es un nodo que depende directamente de otro, es decir está conectado mediante una flecha que llega. Todos, menos el nodo raíz.

Padre: es aquel nodo del que depende al menos un nodo hijo, es decir que los hijos son aquellos nodos a los que apuntan las flechas que salen de este. En este caso A, B, C, G.

Hermanos: son el conjunto de nodos que tienen el mismo padre, para este caso se puede observar los siguientes grupos de hermanos “B-C”, “D-E”, “F-G-H”.

Descendientes: los descendientes de un nodo son todos aquellos nodos que pertenecen a los subárboles de los hijos de dicho nodo. Por ejemplo los descendientes de B son D y E, mientras que los de C son F, G, H, I.

Ancestros: los ancestros de un nodo, son todos aquellos nodos en el camino desde la raíz a dicho nodo. Para este caso por ejemplos los ancestros de E son A y B, mientras que los de I son A, C, G.

Hoja: son todos los nodos que no tienen hijos, se ubican en la parte superior del árbol, pero pueden estar en distintos niveles, en nuestro ejemplo D, E, F, H, I.

Nodo interno: son todos aquellos nodos que tienen al menos un hijo, a excepción de la raíz, es decir B, C, G.

Grado de un nodo: es el número de subárboles de dicho nodo, por ejemplo el grado de A es dos y el de G es uno.

Rama: es la conexión entre dos nodos, también es llamado enlace.

Grado de un árbol: es el grado (o aridad) máximo de todos los nodos del árbol, es decir la cantidad máxima de hijos que tiene alguno de los nodos del árbol. En este caso, el grado del árbol es tres.

Camino: es una secuencia de nodos desde un nodoj a un nodok –distintos entre sí– de modo que exista una rama que conecte cada par de nodos –siempre en sentido descendente–, su longitud es el número de ramas que contiene. Para este caso por ejemplo el camino de C hasta I es “C-G-I” y su longitud es dos. Además cada nodo del árbol a excepción del nodo raíz puede ser accedido desde este último mediante un camino único a través de una secuencia de ramas. Si existe un camino entre un nodoj y un nodok, entonces nodoj es un ancestro de nodok y nodok es un descendiente de nodoj.

Nivel de un nodo: se define por el número de ramas entre dicho nodo y la raíz más uno, es decir para este caso el nivel de A es uno y el de F es tres.

Altura de un nodo: es el número de ramas del camino más largo entre ese nodo y una hoja, por ejemplo la altura de *C* es dos y la de *B* es uno.

Altura de un árbol: es la altura de su nodo raíz, para este caso particular la altura del árbol es tres.

Profundidad de un nodo: es el número de ramas desde la raíz del árbol hasta dicho nodo, por ejemplo la profundidad de *I* es tres y la de *B* es uno.

Subárbol: es un subconjunto de nodos del árbol aunque un solo nodo es considerado un árbol también. Por ejemplo el subárbol *B* está formado por su raíz *B* y sus dos hijos *D* y *E*, pero *D* también es un subárbol compuesto solo por él mismo.

Bosque: es un conjunto de dos o más árboles –si solo se elimina el nodo raíz de un árbol se puede generar un bosque formado por los subárboles de la raíz–, para este ejemplo al eliminar *A* se puede generar el bosque formado por los árboles *B* y *C*.

Si bien los árboles a simple vista no parecen una estructura con la que estemos familiarizados también está presente en nuestra vida cotidiana y podemos mencionar varios ejemplos: para representar el árbol genealógico de una familia, en los organigramas de una organización u empresa, para representar el índice de un libro. Visto desde un enfoque informático, se usan para analizar circuitos eléctricos, representar expresiones matemáticas, para analizar la estructura sintáctica del código fuente de un algoritmo, por algoritmos de minería de datos, por las bases de datos para manejar los índices de acceso a la información y además por los sistemas operativos para manejar los directorios de una computadora y rutas de accesos a archivos –conocidos como *paths*–.

Por lo cual podemos definir un árbol de esta forma considerando su estructura y mecánica de funcionamiento: es una estructura ramificada dinámica de datos que están ordenados de manera general o por nivel dependiendo del tipo.

Los árboles son muy eficientes para realizar operaciones de búsqueda aun cuando el volumen de datos es muy grande por lo que suelen ser utilizados como índices para acceder a la información que se almacena en el disco –memoria secundaria– por lo que normalmente no se almacena toda la información en cada nodo del árbol sino que se almacena el campo clave –sobre el que se realizará la búsqueda– y la posición donde se encuentra el resto de la información en el archivo, también llamado número relativo de registro (*nrr*). Esto también reduce significativamente el tamaño de la representación del árbol en memoria mejorando su rendimiento y permitiendo utilizar esos recursos para construir más de un árbol con distintos índices de acceso a un mismo archivo.

Recién hemos comenzado a trabajar con estructuras ramificadas y seguro nos surge la pregunta: *¿cómo realizar un barrido en este tipo de estructura?* Para el caso particular de árboles que es una estructura jerárquica, existen tres maneras útiles de mostrar el contenido de un árbol, por *preorden* o de orden previo, *inorden* o en orden y *postorden* o de orden posterior. Estos barridos son recursivos y se definen de forma general de la siguiente manera: en el primero caso *preorden* en el cual primero se trata el nodo raíz y luego se llama recursivamente el barrido con sus nodos hijos, primero el izquierdo y luego el derecho. El segundo es *inorden* para este primero se trata el hijo izquierdo recursivamente, luego el nodo raíz y después el hijo derecho recursivamente. Y el último caso es *postorden* en el que primero se trata el hijo derecho recursivamente, luego el nodo raíz y luego el hijo izquierdo recursivamente.

Un árbol compuesto de dos árboles, siempre pensando en binario

El árbol binario de búsqueda (abb) es un caso particular de árbol en el que cada nodo puede tener como máximo dos hijos, los cuales se denominan hijo izquierdo e hijo derecho, por ende el grado del árbol como máximo será dos. Estos están ordenados de manera general, es decir que todos los nodos del subárbol izquierdo son menores que el padre y todos los del subárbol derecho mayores, a partir de esto se puede establecer la siguiente regla: para cada nodo del árbol se establece que el nodo izquierdo es menor y el nodo derecho es mayor, como se puede ver en la figura 2. Además los árboles binarios son una estructura ramificada regular que puede ser representada utilizando simplemente un vector, aunque en nuestro caso trabajaremos con la implementación dinámica.

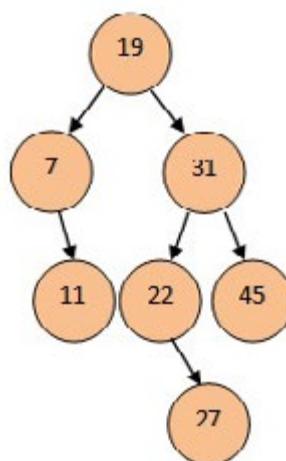


Figura 2. Árbol binario de búsqueda

Además estos árboles se utilizan para representar expresiones matemáticas que pueden ser descompuestas en base dos, también se los utiliza para construir árboles de decisión, que nos permite representar la estructura de razonamiento de un sistema experto a partir de reglas simples y también en el área de minería de datos. Estos casos particulares de árboles binarios no están ordenados de manera general, sino que son construidos de una manera específica, determinada por las reglas necesarias para resolver el problema sobre el que se trabaja, a continuación en la figura 3 se observa un ejemplo de ambos casos de izquierda a derecha respectivamente.

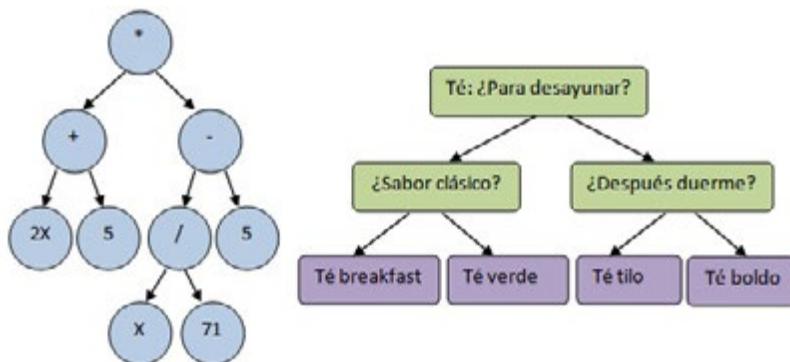


Figura 3. Árbol binario de expresión y de decisión

A partir de las reglas definidas previamente se pueden observar las siguientes propiedades: el número máximo de nodos en el nivel i de un árbol binario es 2^{i-1} , y el número máximo de nodos en un árbol binario de altura k es $2^k - 1$. Se dice que un árbol binario de altura k está lleno si tiene $2^k - 1$ nodos, se dice que un árbol binario de altura k está completo si está lleno hasta la altura $k-1$ y su último nivel está ocupado de izquierda a derecha. Además debemos tener en cuenta el desequilibrio producido durante la carga de un árbol, para que este no pierda su eficiencia de búsqueda y acceso en todos los subárboles. Por ejemplo, en la figura 4 a la izquierda se puede observar un árbol desequilibrado dado que en el subárbol izquierdo en el peor solo se hará una comparación $O(1)$, mientras que en el subárbol derecho se harán cuatro comparaciones $O(4)$ –es decir que la eficiencia en los subárboles no es la misma–. Por su parte a la derecha de la figura podemos ver el peor caso de un árbol –en el cual los elementos han sido cargados en orden– esto se lo conoce como árbol degenerado y el orden de acceso es de $O(n)$ como si fuera una lista. Esta situación representa un problema crítico que debemos solucionar para que los árboles realmente puedan ser eficientes, más adelante veremos cómo podemos lograr equilibrar los árboles.

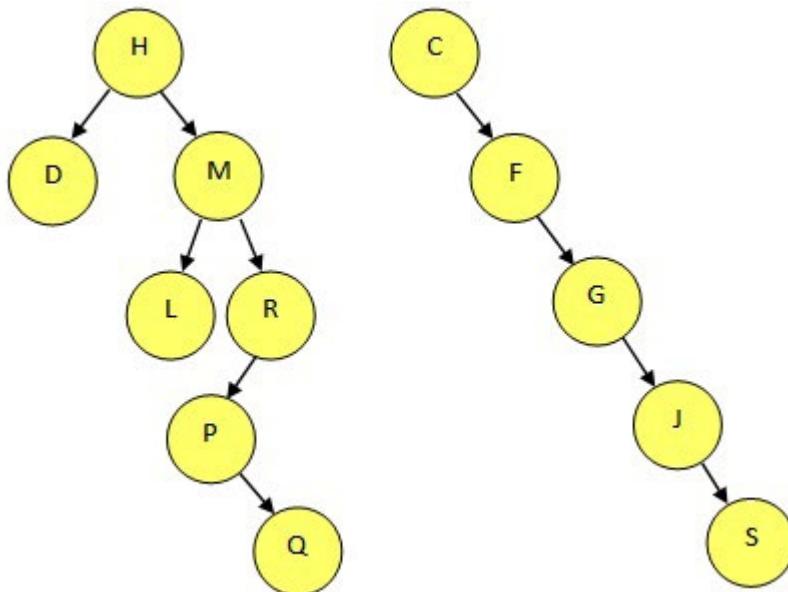


Figura 4. Árbol desequilibrado (a la izquierda) y degenerado (a la derecha)

Ya conocemos las bases fundamentales de árboles para pasar al diseño del TDA árbol binario de búsqueda, a diferencia de los anteriores solo necesitaremos una variable puntero –que será la raíz del árbol– que apuntará a un nodoArbol. Este último está compuesto de tres elementos, información, hijo izquierdo e hijo derecho, además de las funciones que detallan su manera de operar, las cuales se enumeran a continuación:

1. `insertar_nodo(raíz, elemento)`. Agrega el elemento al árbol;
2. `eliminar_nodo(raíz, clave)`. Elimina y devuelve del árbol si encuentra un elemento que coincide con la clave dada –el primero que encuentre–, si devuelve `None` significa que no se encontró la clave en el árbol, y por ende no se elimina ningún elemento;
3. `reemplazar(raíz)`. Determina el nodo que reemplazará al que se va a eliminar, esta es una función interna que solo es utilizada por la función `eliminar`;

4. arbol_vacio(raíz). Devuelve verdadero (*true*) si el árbol no contiene elementos;
5. buscar(raíz, clave). Devuelve un puntero que apunta al nodo que contiene un elemento que coincida con la clave –el primero que encuentra–, si devuelve *None* significa que no se encontró la clave en el árbol;
6. preorden(raíz). Realiza un recorrido de orden previo del árbol mostrando la información de los elementos almacenados en el árbol;
7. inorder(raíz). Realiza un recorrido en orden del árbol mostrando la información de los elementos almacenados en el árbol;
8. postorden(raíz). Realiza un recorrido de orden posterior del árbol mostrando la información de los elementos almacenados en el árbol.
9. por_nivel(raíz). Realiza un recorrido del árbol por nivel mostrando la información de los elementos almacenados.

Es momento de comenzar con la implementación del TDA abb, para lo cual definiremos primero la estructura del nodoArbol que se presenta en la figura 5. El cual tiene tres campos –como ya mencionamos– información, izquierdo y derecho, el primero tendrá el valor de dato a cargar en el nodo y los dos últimos *None*. Dado que solo necesitamos un puntero que apunte al nodo raíz no será necesario definir una clase árbol como lo hicimos en los TDA de las estructuras anteriores. Luego en las figuras 6, 7, 8, 9 y 10 se pueden encontrar la definición de las funciones mencionadas anteriormente, estos serán los eventos que nos permitirán administrar el funcionamiento del árbol.

```
class nodoArbol(object):
    """Clase nodo árbol.

    def __init__(self, info):
        """Crea un nodo con la información cargada."""
        self.izq = None
        self.der = None
        self.info = info
```

Figura 5. Definición de la estructura del nodoArbol

```

def eliminar_nodo(raiz, clave):
    """Elimina un elemento del árbol y lo devuelve si lo encuentra."""
    x = None
    if(raiz is not None):
        if(clave < raiz.info):
            raiz.izq, x = eliminar_nodo(raiz.izq, clave)
        elif(clave > raiz.info):
            raiz.der, x = eliminar_nodo(raiz.der, clave)
        else:
            x = raiz.info
            if(raiz.izq is None):
                raiz = raiz.der
            elif(raiz.der is None):
                raiz = raiz.izq
            else:
                raiz.izq, aux = remplazar(raiz.izq)
                raiz.info = aux.info
    return raiz, x

```

Figura 6. Interfaz o eventos del TDA árbol parte 1

```

def insertar_nodo(raiz, dato):
    """Inserta un dato al árbol."""
    if(raiz is None):
        raiz = nodoArbol(dato)
    elif(dato < raiz.info):
        raiz.izq = insertar_nodo(raiz.izq, dato)
    else:
        raiz.der = insertar_nodo(raiz.der, dato)
    return raiz

def arbolvacio(raiz):
    """Devuelve true si el árbol esta vacio."""
    return raiz is None

```

Figura 7. Interfaz o eventos del TDA árbol parte 2

```

def remplazar(raiz):
    """Determina el nodo que remplazará al que se elimina."""
    aux = None
    if(raiz.der is None):
        aux = raiz
        raiz = raiz.izq
    else:
        raiz.der, aux = remplazar(raiz.der)
    return raiz, aux

def por_nivel(raiz):
    """Realiza el barrido postorden del árbol."""
    pendientes = Cola()
    arribo(pendientes, raiz)
    while(not cola_vacia(pendientes)):
        nodo = atención(pendientes)
        print(nodo.info)
        if(nodo.izq is not None):
            arribo(pendientes, nodo.izq)
        if(nodo.der is not None):
            arribo(pendientes, nodo.der)

```

Figura 8. Interfaz o eventos del TDA árbol parte 3

```

def buscar(raiz, clave):
    """Devuelve la dirección del elemento buscado."""
    pos = None
    if(raiz is not None):
        if(raiz.info == clave):
            pos = raiz
        elif clave < raiz.info:
            pos = buscar(raiz.izq, clave)
        else:
            pos = buscar(raiz.der, clave)
    return pos

```

Figura 9. Interfaz o eventos del TDA árbol parte 4

```

def inorder(raiz):
    """Realiza el barrido inorder del árbol."""
    if(raiz is not None):
        inorder(raiz.izq)
        print(raiz.info)
        inorder(raiz.der)

def preorden(raiz):
    """Realiza el barrido preorden del árbol."""
    if(raiz is not None):
        print(raiz.info)
        preorden(raiz.izq)
        preorden(raiz.der)

def postorden(raiz):
    """Realiza el barrido postorden del árbol."""
    if(raiz is not None):
        postorden(raiz.der)
        print(raiz.info)
        postorden(raiz.izq)

```

Figura 10. Interfaz o eventos del TDA árbol parte 5

Para entender cómo se realizan las actividades de *insertar*, *eliminar* y *buscar* elementos, desglosemos analíticamente las acciones realizadas por dichas funciones, denotando lo siguiente: cuando se inserta un elemento, primero verifica si la raíz del árbol es *None*, de ser así crea un nodoArbol al cual se le carga el campo información con el dato ingresado; caso contrario se chequea si el dato es menor que el almacenado en el nodo raíz, si es menor llamamos recursivamente a la función con el subárbol izquierdo y sino con el subárbol derecho, hasta encontrar el lugar donde se insertará el nuevo nodo (tenga en cuenta que siempre que se agrega un nodo a un árbol, este se agrega como una hoja del mismo, nunca como nodo intermedio). Por su parte, para eliminar un nodo del árbol primero se debe buscar el nodo que se desea quitar. Para hacer, esto evaluamos si el elemento a eliminar es menor o mayor que el que está en el nodo raíz y se llama recursivamente a la función con el subárbol izquierdo o derecho respectivamente hasta encontrar el elemento o llegar a una hoja del árbol. Si el elemento fue encontrado en el árbol, entonces ocurrirá uno de los siguientes tres casos:

El primer caso es que el nodo a eliminar no tenga hijo derecho, en este caso al nodo a eliminar se le asigna el hijo izquierdo de dicho nodo. El segundo caso ocurre cuando no tiene hijo izquierdo, al igual que el anterior al nodo a eliminar se le asigna el hijo derecho. El último caso es que presente ambos hijos, este es un poco más complicado que los anteriores, para resolverlo requeriremos realizar algunas operaciones –dado que hay que enlazar los dos subárboles del nodo que se quita manteniendo el orden de los mismos–. Entonces, tenemos que buscar un nodo hoja cuyo valor reemplace al del nodo que se quiere eliminar para luego quitar dicha hoja. Para lo cual hay dos

alternativas, buscar la hoja mayor del subárbol izquierdo o la menor del subárbol derecho, esto lo resolveremos con una función auxiliar denominada “reemplazar” cuya tarea es encontrar la hoja con la cual se intercambiará la información. En el ejemplo de la figura II se quiere eliminar el nodo con valor K , y se opta por buscar la hoja mayor del subárbol izquierdo para este caso particular es el nodo con valor J , después se copia la información y se procede a eliminar dicha hoja. Sin importar cuál de estos casos ocurra se puede quitar previamente la información de nodo a eliminar antes de remplazarlo.

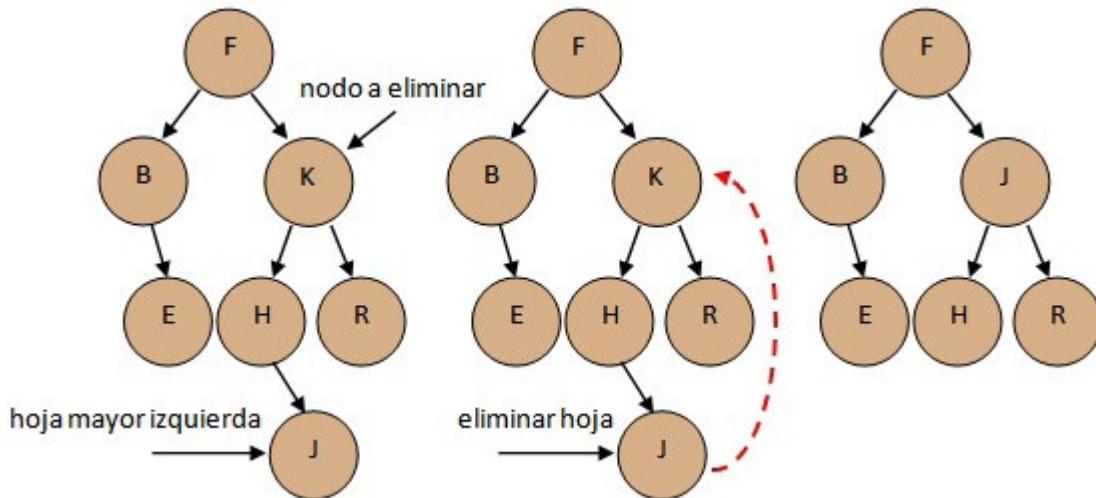


Figura II. Eliminación de nodo con ambos hijos

Nos resta las búsquedas. Para esto mientras la raíz sea distinto de *None*, comparamos si el valor almacenado en la raíz es igual al buscado, en dicho caso se guarda la dirección del nodo en la variable auxiliar posición –es decir hemos encontrado el elemento buscado– caso contrario al igual que en la inserción determinamos si el valor buscado es menor o mayor que el del nodo raíz, para llamar recursivamente a la función con el subárbol izquierdo o derecho respectivamente. Finalmente, se devuelve el puntero auxiliar posición. Si este es distinto de *None*, el dato fue encontrado y la variable posición tiene la dirección del nodo que la contiene, sino no fue encontrada. Los árboles binarios nos permiten realizar una búsqueda binaria incluso en la implementación dinámica, algo que no podíamos hacer en las estructuras lineales.

Finalmente nos queda examinar las funciones para recorrer un árbol inorden, preorden y postorden, las que solamente cambian el orden en que se trata el nodo y hacen las llamadas recursivas. Siguiendo con el ejemplo del primer árbol a la izquierda en la figura II al ejecutar los distintos recorridos obtendremos las siguientes salidas: con el recorrido inorden la salida es B, E, F, H, J, K, R , si lo hacemos con preorden obtendremos F, B, E, K, H, J, R y finalmente con postorden el resultado es R, K, J, H, F, E, B . Mientras que el barrido “por nivel” utiliza internamente una cola, primero trata el nodo y luego hace un arribo de los hijos de dichos nodos que quedaran pendientes de ser tratados y pasa al siguiente nodo de la cola, cuando la cola está vacía termina el barrido la salida que se obtiene es F, B, K, E, H, R, J .

Comprimiendo información con árboles binarios gracias a los códigos de Huffman

Los códigos Huffman se utilizan para compresión de datos, para esto se crea una tabla de códigos de longitud variable para codificar un determinado símbolo –como un carácter en un archivo o trama a transmitir–, dicha tabla se genera de una manera específica basada en la frecuencia y aparición de dicho símbolo, también llamada peso. Este algoritmo de compresión fue desarrollado por David A. Huffman¹ en 1952, como trabajo final de la asignatura Teoría de la Información del profesor Robert Fano, quien trabajaba con Claude Shannon, el fundador del campo de la Teoría de la Información² en 1948.

Esta codificación se genera a partir de un método específico para determinar la representación de cada símbolo, generando un código prefijo, es decir, que el código que representa a un símbolo en particular nunca es el prefijo de otro código de un símbolo distinto. Dicho código representa los caracteres que aparecen más frecuentemente usando cadenas de bits cortas, los menos frecuentes con cadenas más largas.

Otra característica muy útil de los códigos de Huffman es que son de decodificación inmediata, esto quiere decir, que no es necesario utilizar algún carácter especial para separar o delimitar las distintas partes de la cadena codificada, gracias a la propiedad de los códigos de ser prefijos. Entonces, cuando decodificamos una cadena seguimos su recorrido desde la raíz del árbol hasta llegar una hoja lo que implica que se ha decodificado una parte de la cadena y luego comienza la otra. Además se puede garantizar que ante un número C de caracteres de entradas, el árbol tendrá como máximo $(2^*C) - 1$ nodos.

¿Pero cómo se generan los códigos? ¿Cómo construimos el árbol binario?

Bueno vamos a contestar esto a través de un ejemplo del proceso de generación de códigos de Huffman y de cómo construir el árbol necesario para esto. Primero partiremos de un bosque de nodos raíces en el cual cada nodo contará con el peso que se observa en la siguiente tabla de frecuencias:

Carácter	Frecuencia
I	0.28
N	0.16
T	0.08
E	0.16
L	0.08
G	0.08
C	0.08
A	0.08

¹ David A. Huffman: "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., sept 1952, pp 1098-1102.

² Claude E. Shannon: A Mathematical Theory of Communication, Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, 1948.

Los nodos del bosque estarán ordenados de menor a mayor, primero por peso y luego por orden alfabético del campo información, entonces nuestro bosque quedará de la siguiente manera:

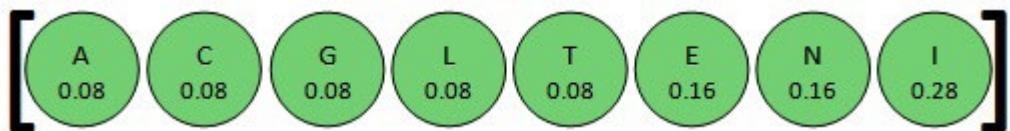


Figura 12. Bosque inicial de nodos

Luego se eligen los dos primeros nodos del bosque *A* y *C* en este caso y se genera un nuevo nodo cuyo peso es la suma de ambos, en el cual el hijo izquierdo de este nuevo nodo será el primero de los dos elegidos y el derecho el segundo. Después se inserta el nuevo nodo por el final del bosque y se adelanta mientras su peso sea menor que el del nodo anterior, por lo cual nuestro bosque quedará de la siguiente manera –nótese que los nuevos nodos que solo tienen peso son insertados al bosque como una cola de prioridad ordenados por el dicho campo–:

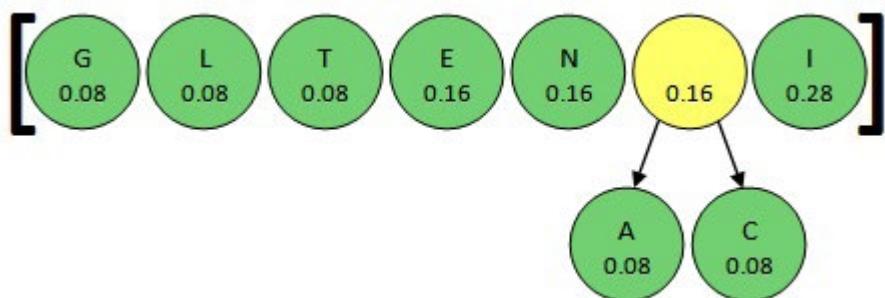


Figura 13. Árbol Huffman paso 1

Nuevamente se vuelve a repetir el mismo proceso con los siguientes nodos de menor peso, en este caso *G* y *L* obtendremos el siguiente bosque:

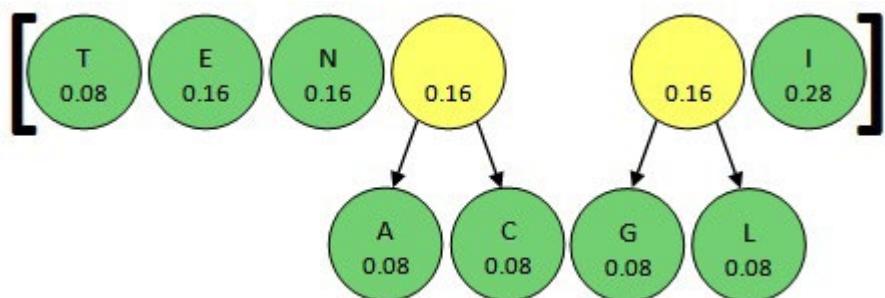


Figura 14. Árbol Huffman paso 2

Seguimos repitiendo este procedimiento hasta que el bosque se transforme en un árbol como se ve a continuación:

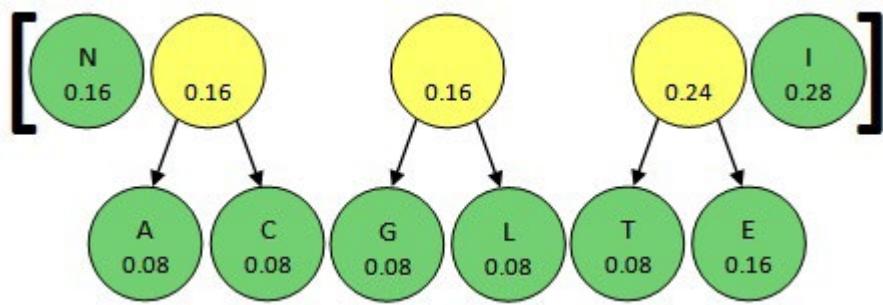


Figura 15. Árbol Huffman paso 3

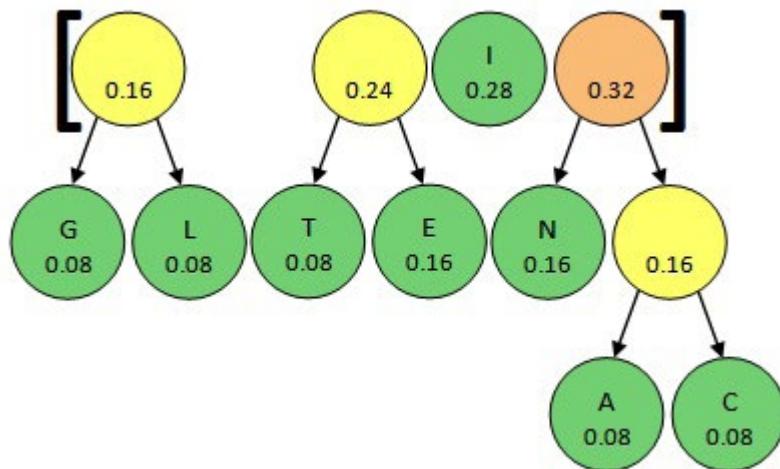


Figura 16. Árbol Huffman paso 4

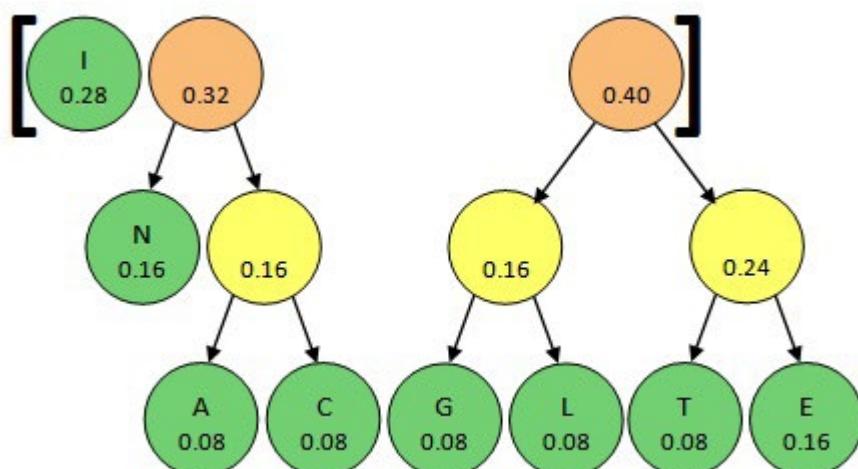


Figura 17. Árbol Huffman paso 5

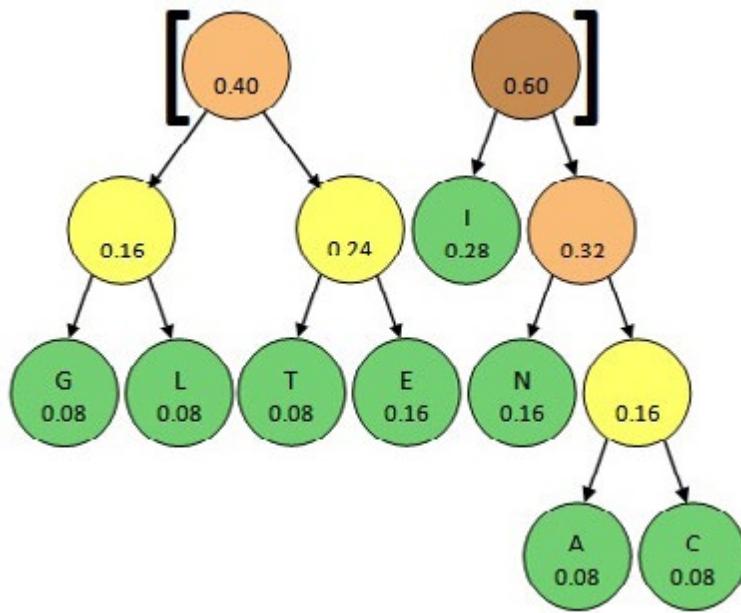


Figura 18. Árbol Huffman paso 6

Finalmente, obtendremos el árbol que se observa a en la figura 19, en el cual las ramas que apuntan a la izquierda son ceros y las que apuntan a la derecha unos, mediante el cual podremos generar la tabla de código –que se observa después del árbol–, además también utilizaremos dicho árbol para decodificar o descomprimir la cadena codificada, nótese que el nodo raíz debería tener valor uno o aproximado a este dependiendo de los decimales tomados en el cálculo de las frecuencias:

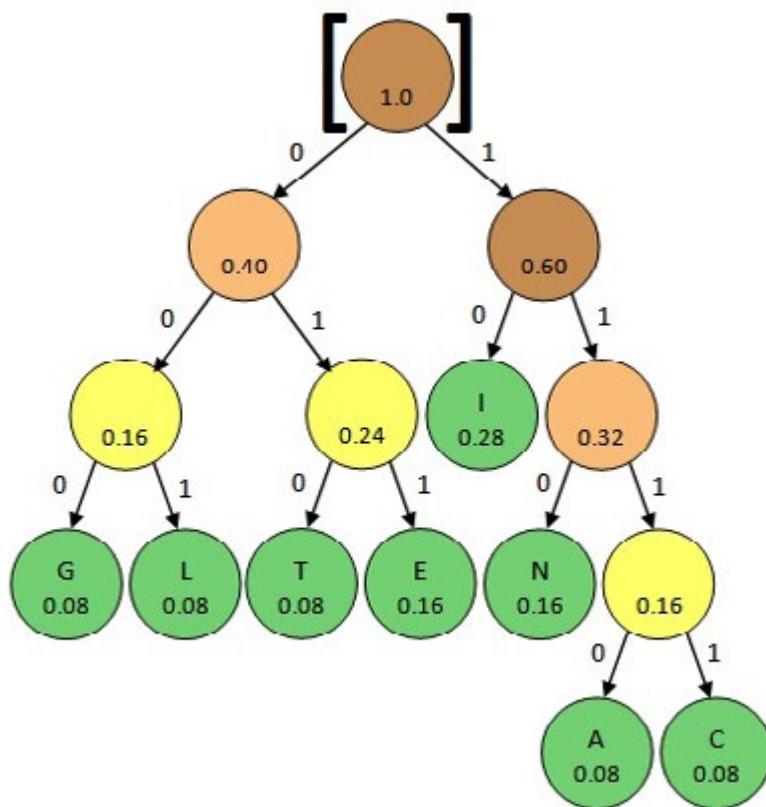


Figura 19. Árbol Huffman final

Carácter	Código
I	IO
N	IIO
T	OIO
E	OII
L	OOI
G	OOO
C	III
A	IIIO

Para finalizar con el ejemplo, veamos cómo se decodifica una cadena. Supongamos que dispone de la siguiente cadena comprimida “`IOIIOOIOOOIOOOOOOOIIIIOIIIIOIIIO`”, como se puede aplicar descodificación instantánea –por lo que no hay caracteres delimitadores de las partes de la cadena– se toma el primer carácter y partiendo de la raíz del árbol, si es un cero, pasamos al subárbol izquierdo y, si es un uno, al subárbol derecho. Luego se toma el siguiente carácter de la cadena y se vuelve a hacer lo mismo, hasta que el nodo accedido sea una hoja del árbol. Cuando llegamos a una hoja hemos decodificado un carácter y se vuelve a iniciar desde la raíz con el resto de la cadena. Siguiendo con el ejemplo, tomamos el primer carácter que es un uno, entonces se accede al subárbol derecho luego sigue un cero por lo que se accede al derecho, y en este caso ya es una hoja por lo que el primer carácter de la cadena decodificada es una *I*, luego siguen dos uno y un cero, el carácter es una *N*. Obsérvese que como los códigos son prefijos solo hay una combinación posible a cada hoja. Ahora queda a cargo del lector terminar de decodificar la cadena para descubrir cuál es el mensaje y, luego, diseñar un algoritmo que permita realizar la decodificación de manera automática generando el árbol binario obtenido previamente.

Árboles que pueden equilibrarse un poco de ayuda de AVL

Este tipo de árbol binario fue desarrollado por los matemáticos rusos Adelson-Velskii y Landis en 1962³ (de ahí proviene su nombre). Fue el primer árbol de búsqueda binario auto balanceado. Un árbol AVL está siempre equilibrado, de tal modo que para todos los nodos la diferencia de altura entre la rama izquierda y la altura de la rama derecha no difiere en más de una unidad. Gracias a esta propiedad de mantener su equilibrio, la complejidad de realizar búsqueda en uno de estos árboles está siempre en orden de $O(\log n)$. El factor de equilibrio, es decir la altura de cada nodo, se puede almacenar directamente en cada nodo o ser calculado a partir de las alturas de los subárboles. Para poder realizar el cálculo del factor de equilibrio es necesario conocer la altura de cada nodo, por lo que debemos redefinir el nodoArbol agregándole el campo altura, como se observa en la figura 20. Dicho campo se actualiza en cada operación de inserción, eliminación o rotación simple que se

³ Adelson-Velsky, Georgy; Landis, Evgenii (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences (in Russian). 146: 263–266. English translation by Myron J. Ricci in Soviet Mathematics - Doklady, 3:1259–1263, 1962.

realiza en el árbol, como dichas operación son recursivas se actualizara la altura de todos los nodos que forman parte del camino desde el nodo donde se realizo dicha operación hasta la raíz, para garantizar que todo el árbol quede balanceado.

```
class nodoArbol(object):
    """Clase nodo árbol."""

    def __init__(self, info):
        """Crea un nodo con la información cargada."""
        self.izq = None
        self.der = None
        self.info = info
        self.altura = 0
```

Figura 20. Nodo árbol AVL

Primero necesitaremos determinar la altura de cada nodo para lo cual utilizaremos dos funciones una que se encargue calcular la altura y la otra de actualizarla cuando el nodo tiene hijos –como mencionamos anteriormente la altura de un nodo es la longitud del camino más largo desde dicho nodo a una hoja–, podemos ver ambas funciones en la figura 21.

```
def altura(raiz):
    """Devuelve la altura de un nodo."""
    if(raiz is None):
        return -1
    else:
        return raiz.altura

def actualizaraltura(raiz):
    """Actualiza la altura de un nodo."""
    if(raiz is not None):
        alt_izq = altura(raiz.izq)
        alt_der = altura(raiz.der)
        raiz.altura = (alt_izq if alt_izq > alt_der else alt_der) + 1
```

Figura 21. Cálculo de altura de un nodo

Veamos ahora cómo hacer para que los árboles se auto balanceen, para conseguir esto luego de las operaciones de inserción y eliminación de los nodos debemos revisar si el árbol rompió su condición de equilibrio, es decir si la altura de sus subárboles izquierdo y derecho difieren en más de una unidad, en dicho caso hay que realizar una serie de acciones entre nos los nodos llamadas rotaciones para volver a balancear el árbol. Existen dos tipos de rotaciones “rotación simple” y “rotación doble”, ambas pueden hacerce haciaa la izquierda como a la derecha, por lo cual tendremos cuatro casos posibles para recuperar el equilibrio del árbol y los detallaremos a continuación: comencemos

con el primer caso es rotación simple a la derecha, como se nota en la figura 22 se rompe el factor de equilibrio –el subárbol izquierdo tiene altura 2 y el derecho 0–, por lo cual se debe rotar el hijo izquierdo de la raíz de árbol a la derecha. El hijo izquierdo se transforma en la raíz y la raíz pasa a ser el hijo derecho, finalmente se actualizan las alturas de los nodos involucrados y de esta manera ambos subárboles quedan con altura 1.

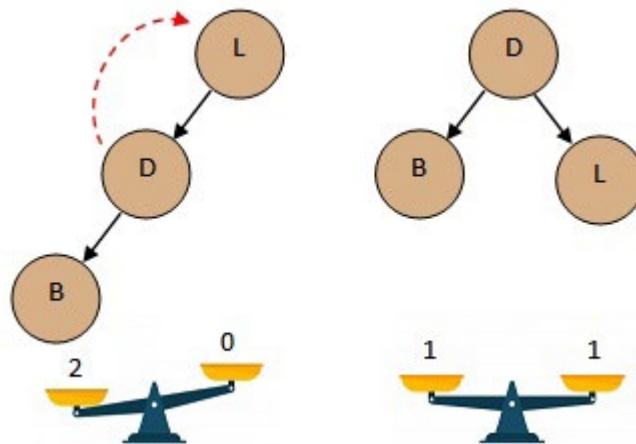


Figura 22. Rotación simple a la derecha

El segundo caso es el simétrico opuesto al anterior –rotación simple a la izquierda–, es decir rotamos el hijo derecho de la raíz del árbol a la izquierda como podemos apreciar en la figura 23, entonces el hijo derecho se transforma en la raíz del árbol para equilibrar la altura de los dos subárboles.

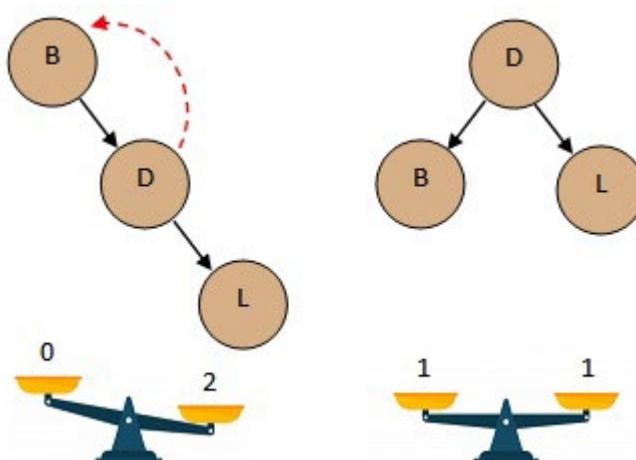


Figura 23. Rotación simple a la izquierda

Seguimos con el tercer caso es rotación doble a la derecha, lo cual implica hacer dos rotaciones simples para poder recuperar el equilibrio del árbol primero se rota el hijo derecho del hijo izquierdo de la raíz a la izquierda y luego se rota el hijo izquierdo de la raíz a la derecha como se muestra en la figura 24 para restablecer finalmente el factor de equilibrio.

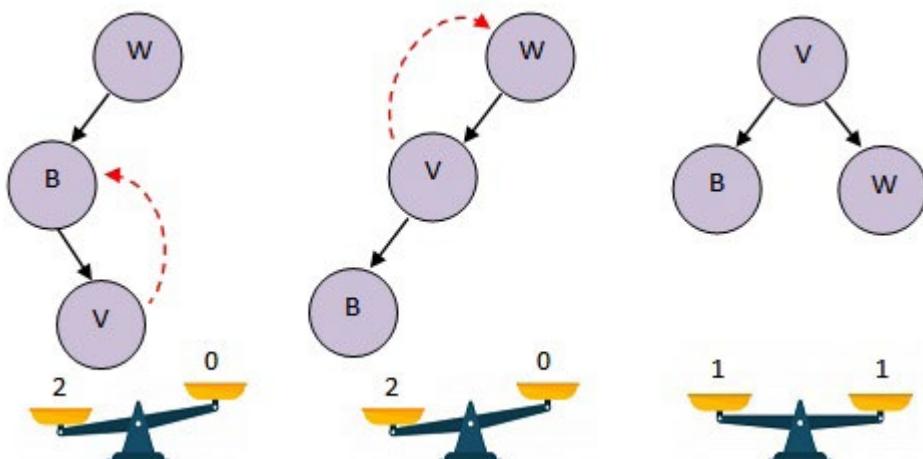


Figura 24. Rotación doble a la derecha

Por último, el cuarto caso es el opuesto simétrico al anterior –rotación doble a la izquierda-. En esta ocasión, primero rotamos el hijo izquierdo del hijo derecho de la raíz hacia la derecha y luego, rotamos el hijo derecho de la raíz hacia la izquierda logrando balancear el árbol, como se puede apreciar en la figura 25.

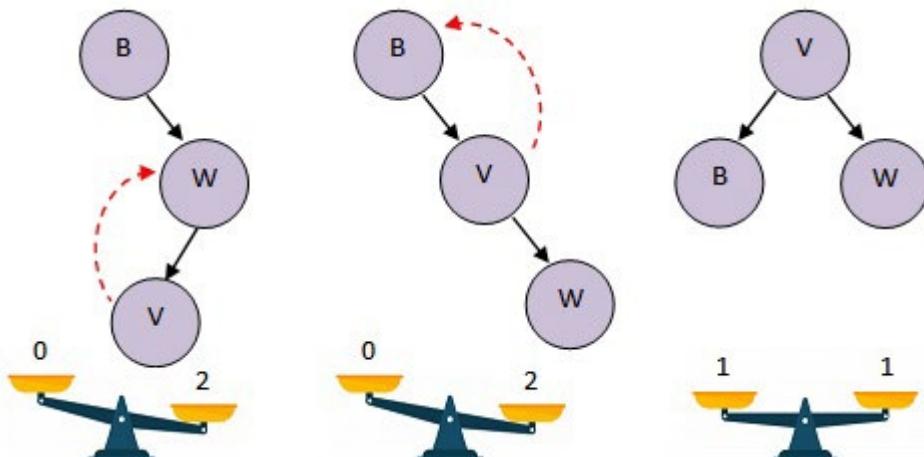


Figura 25. Rotación doble a la izquierda

Nos resta ver de qué manera implementamos estas funciones para lograr que el árbol pueda auto balancearse cuando el mismo crece o decrece en cantidad de elementos, en las figuras 26 y 27 se presentan las funciones de rotación simple y rotación doble respectivamente, las cuales reciben como parámetros de entrada la raíz del subárbol desequilibrado y una variable de control booleana que indica si la rotación debe hacerse a la derecha (*True*) o a la izquierda (*False*).

```

def rotar_simple(raiz, control):
    """Realiza una rotación simple de nodos a la derecha o a la izquierda."""
    if control:
        aux = raiz.izq
        raiz.izq = aux.der
        aux.der = raiz
    else:
        aux = raiz.der
        raiz.der = aux.izq
        aux.izq = raiz
    actualizaraltura(raiz)
    actualizaraltura(aux)
    raiz = aux
    return raiz

```

Figura 26. Funciones rotación simple

```

def rotar_doble(raiz, control):
    """Realiza una rotación doble de nodos a la derecha o a la izquierda."""
    if control:
        raiz.izq = rotar_simple(raiz.izq, False)
        raiz = rotar_simple(raiz, True)
    else:
        raiz.der = rotar_simple(raiz.der, True)
        raiz = rotar_simple(raiz, False)
    return raiz

```

Figura 27. Funciones rotación doble

En este momento ustedes se preguntarán *¿De qué manera determina el árbol qué tipo de rotación debe hacer?* Para esto desarrollaremos una función que se denominará “balancear” que se encargará de verificar si se rompió el factor de equilibrio. Si así lo fuera, será necesario determinar qué caso de rotación debe aplicar para restaurar el equilibrio. Dicha función se presenta en la figura 28, nótese que las acciones de rotación se ejecutan cuando el factor de equilibrio es igual a dos.

```

def balancear(raiz):
    """Determina que rotación hay que hacer para balancear el árbol."""
    if(raiz is not None):
        if(altura(raiz.izq)-altura(raiz.der) == 2):
            if(altura(raiz.izq.izq) >= altura(raiz.izq.der)):
                raiz = rotar_simple(raiz, True)
            else:
                raiz = rotar_doble(raiz, True)
        elif(altura(raiz.der)-altura(raiz.izq) == 2):
            if(altura(raiz.der.der) >= altura(raiz.der.izq)):
                raiz = rotar_simple(raiz, False)
            else:
                raiz = rotar_doble(raiz, False)
    return raiz

```

Figura 28. Función balancear

Finalmente para poder aplicar el proceso de balanceo de árboles en las figuras 29 y 30 podemos observar cómo se aplica la función para balancear el árbol en las operaciones de inserción y eliminación respectivamente.

```
def insertar_nodo(raiz, dato, pos):
    """Inserta un dato al árbol."""
    if(raiz is None):
        raiz = nodoArbol(dato, pos)
    elif(dato < raiz.info):
        raiz.izq = insertar_nodo(raiz.izq, dato, pos)
    else:
        raiz.der = insertar_nodo(raiz.der, dato, pos)
    raiz = balancear(raiz)
    actualizaraltura(raiz)
    return raiz
```

Figura 29. Llamadas a la función balancear al insertar

```
def eliminar_nodo(raiz, clave):
    """Elimina un elemento del árbol y lo devuelve si lo encuentra."""
    x = None
    if(raiz is not None):
        if(clave < raiz.info):
            raiz.izq, x = eliminar_nodo(raiz.izq, clave)
        elif(clave > raiz.info):
            raiz.der, x = eliminar_nodo(raiz.der, clave)
        else:
            x = raiz.info
            if(raiz.izq is None):
                raiz = raiz.der
            elif(raiz.der is None):
                raiz = raiz.izq
            else:
                raiz.izq, aux = remplazar(raiz.izq)
                raiz.info, raiz.nrr = aux.info, aux.nrr
    raiz = balancear(raiz)
    actualizaraltura(raiz)
    return raiz, x
```

Figura 30. Llamadas a la función balancear al eliminar

Ahora dejemos un momento de lado los árboles binarios y centrémonos en los árboles n-arios también llamados de multicamino, en los que cada nodo del árbol puede tener n hijos, por lo que la representación y recorrido de los mismos no es tan sencilla como en los árboles binarios que hemos visto hasta acá. Como cada nodo puede tener n hijos, para poder representar esto cada nodo deberá

tener un vector o una lista de hijos ordenados de izquierda a derecha. Respecto del tratamiento, es un poco más complejo que en los árboles binarios, por lo que vamos a necesitar de otras funciones como `hijo_mas_izquierdo(raíz)` y `hermano_derecho(raíz)`, las cuales veremos cómo funcionan a continuación y quedarán a cargo del lector desarrollarlas.

Entonces ¿Qué debemos modificar en nuestro TDA para trabajar con estos árboles? Por suerte existe una técnica que permite generar un árbol binario a partir de un árbol multicamino –un árbol binario no balanceado dado que si no se perdería la relación padre-hijos del árbol n-ario–. Esta técnica se denomina “transformada de Knuth”⁴ o árbol binario hijo- izquierdo hermano-derecho⁵. Esta técnica nos permitirá transformar un árbol n-ario a binario y utilizar el TDA abb para operar sobre el mismo, teniendo algunas consideraciones particulares al momento de *insertar* y *eliminar*.

Los pasos que se deben seguir son los siguientes: primero se debe enlazar en forma vertical (en el subárbol izquierdo) del nodo padre con el hijo que se encuentra más a la izquierda, además se debe eliminar el vínculo de ese padre con el resto de sus hijos. Segundo se debe enlazar el resto de los hijos de manera horizontal (en el subárbol derecho del nodo hijo cargado en el subárbol izquierdo) es decir sus hermanos. Luego repetir este proceso para cada nodo del árbol.

A continuación en la figura 31 se observa un árbol multicamino –de parte del árbol genealógico de deidades griegas– el cual se transformará paso a paso en un árbol binario siguiendo los pasos antes mencionados.

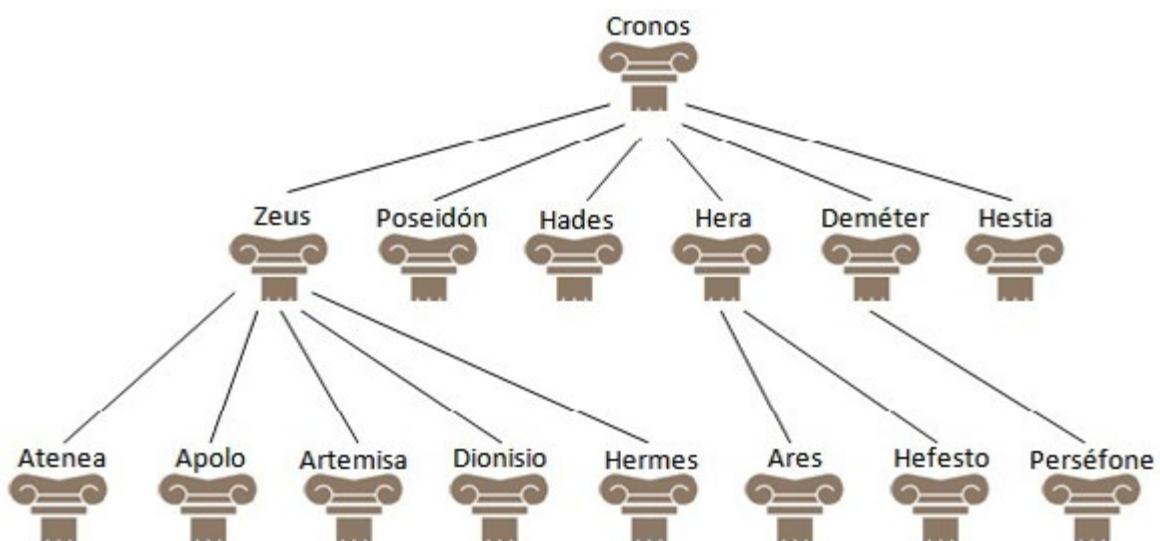


Figura 31. Árbol n-ario

Primero, se carga en la raíz del árbol binario el titán Cronos, luego, en su subárbol izquierdo se coloca el hijo que se ubica más a la izquierda –en este caso el dios Zeus– y se deja en espera al resto de sus hermanos, y vuelve a repetir el mismo proceso, es decir en el subárbol izquierdo del Zeus se inserta el hijo más a la izquierda, en este caso la diosa Atenea. Como esta no tiene hijos se procede a cargar del siguiente hermano que quedó pendiente. Entonces se procede a cargar los hermanos

⁴ Computer Data Structures. John L. Pfaltz. McGraw-Hill, 1977.

⁵ Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. pp. 214–217.

que quedaron pendientes, Apolo, Artemisa, Dionisio y Hermes dado que ninguno de estos tiene hijos –nótese que los hermanos siempre se cargan en el subárbol derecho y el subárbol izquierdo es el primer hijo–. En este punto ya se cargaron todos los descendientes de Zeus, pero aún quedó pendiente tratar a sus hermanos. De la misma manera que se procedió anteriormente se agregan al subárbol derecho de Zeus a Poseidón, Hades y Hera, como esta última tiene hijos se procede a la carga de estos, por lo que ahora insertamos a Ares y Hefesto. Luego, continuamos con los hermanos de Hera que quedaron pendientes aplicando el mismo procedimiento, entonces ahora se agrega a la diosa Deméter y su hija Perséfone para finalmente cargar a la diosa Hestia que había quedado pendiente. De esta manera, hemos logrado transformar un árbol n-ario en uno binario, el árbol resultante de aplicar la transformada de Knuth sobre él árbol de la figura 31 se puede observar a continuación, en la figura 32.

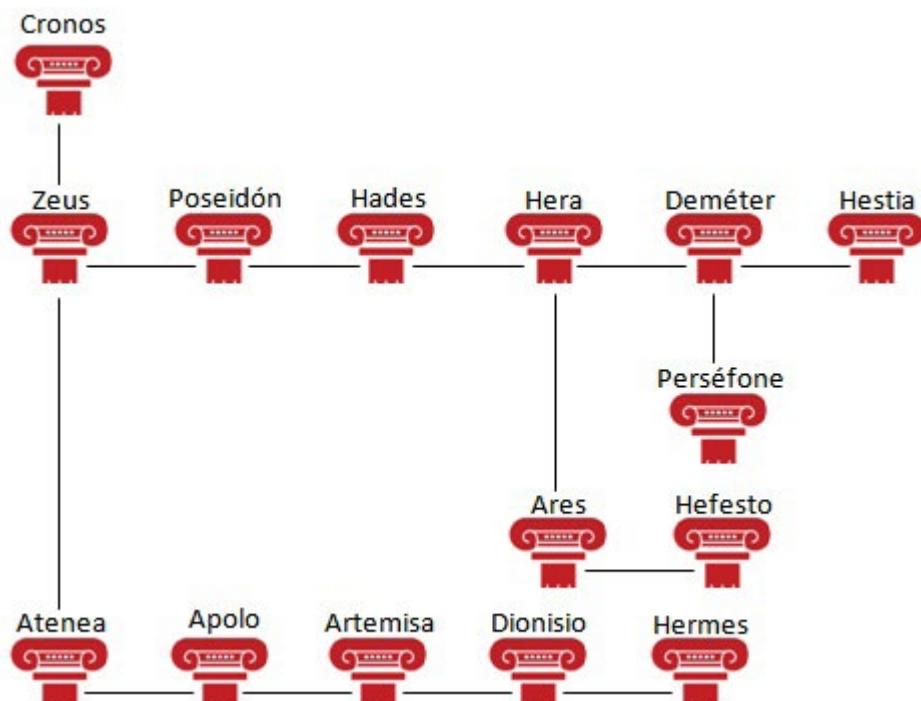


Figura 32. Árbol general transformado a binario

Retomando con árboles binarios en la figura 33 presentaremos un ejemplo de la implementación del TDA árbol AVL para dar solución a un problema. En este caso debemos cargar un árbol con nombres de países de manera desordenada (21 como máximo). Luego los tendremos que listar de manera ordenada y además eliminar uno de los países elegido por el usuario –si el mismo se encuentra en el árbol–. Finalmente hay que modificar el nodo con valor “Tailandia” –suponiendo que el mismo está– porque fue mal cargado por “Tailandia” y por último volver a listar el contenido del árbol para verificar que los cambios se hayan realizado de manera correcta.

```

from tda_arbolbb_avl import insertar_nodo, eliminar_nodo, buscar, inorden

raiz = None
i = 0
pais = input('ingrese nombre del pais a cargar: ')
while(pais != '' and i < 21):
    raiz = insertar_nodo(raiz, pais)
    i += 1
    pais = input('ingrese nombre del pais a cargar: ')

inorden(raiz)
pais = input('ingrese nombre del pais a eliminar: ')
raiz, dato = eliminar_nodo(raiz, pais)
if(dato is not None):
    print('País eliminado:', dato)

pos = buscar(raiz, 'Tailandia')
if(pos is not None):
    raiz, dato = eliminar_nodo(raiz, 'Tailandia')
    dato = 'Tailandia'
    raiz = insertar_nodo(raiz, dato)

inorden(raiz)

```

Figura 33. Ejemplo de uso del TDA árbol AVL

Ahora repasemos las actividades realizadas para la resolución del ejercicio anterior, para consolidar la mecánica del manejo del TDA árbol binario de búsqueda mediante los eventos definidos.

Para comenzar tenemos que importar del TDA árbol las funciones que vamos a utilizar, luego debemos crear la variable raíz para cargar los países al árbol para luego listarlos ordenados alfabéticamente utilizando la función inorden. Después ingresaremos el nombre del país que queremos eliminar, si dicho país está cargado se lo quitará del árbol. Seguidamente procedemos a modificar el país que está mal cargado, para esto se debe quitar dicho elemento del árbol modificarlo y volverlo a insertar, dado que al modificar el campo clave de un nodo alteraremos el orden del árbol –regla para que los abb funcionen–. Al quitarlo y volverlo a insertar quedará ordenado por lo cual el árbol continuará funcionando correctamente y finalmente volvemos a listar el contenido del árbol.

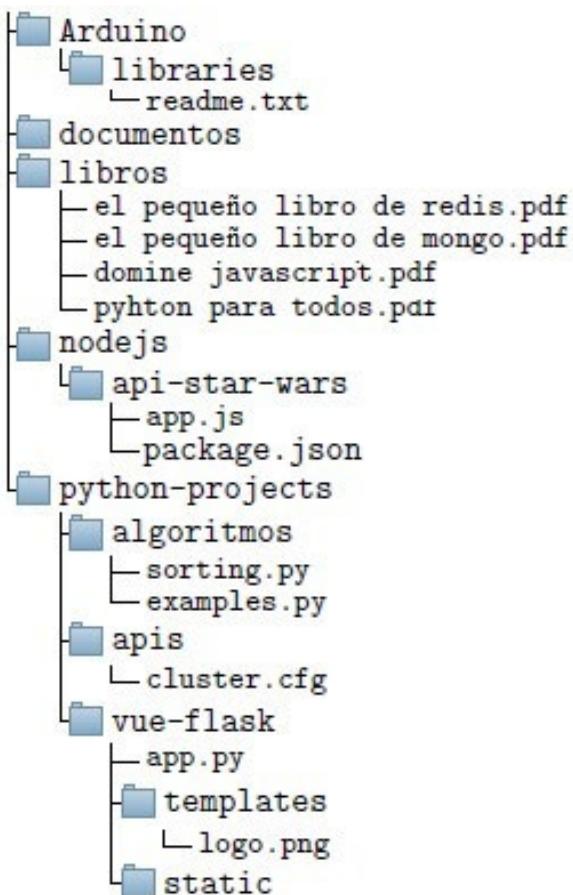
Guía de ejercicios prácticos

A continuación se plantean una serie de problemas, que se deberán resolver utilizar el TDA árbol binario de búsqueda AVL, salvo que el ejercicio pida utilizar otro tipo particular de árbol.

1. Desarrollar un algoritmo que permita cargar 1000 número enteros –generados de manera aleatoria– que resuelva las siguientes actividades:
 - a. realizar los barridos preorden, inorden, postorden y por nivel sobre el árbol generado;
 - b. determinar si un número está cargado en el árbol o no;
 - c. eliminar tres valores del árbol;
 - d. determinar la altura del subárbol izquierdo y del subárbol derecho;
 - e. determinar la cantidad de ocurrencias de un elemento en el árbol;
 - f. contar cuántos números pares e impares hay en el árbol.
2. Implementar una función que permita cargar una expresión matemática en un árbol binario (no balanceado), y resuelva lo siguiente:
 - a. determinar cuál de los barridos muestra la expresión en el orden correcto;
 - b. resolver la expresión matemática y muestre el resultado.
3. Desarrollar un algoritmo que permita cargar el índice del libro Ingeniería de Software de Ian Summerville de manera automática desde un archivo de texto, transformando el árbol n-ario del índice en un árbol binario no balanceado mediante el uso de la transformada de Knuth, para resolver las siguientes actividades:
 - a. listar el índice en su orden original;
 - b. mostrar la parte del índice correspondiente al subtítulo “Diseño de software de tiempo real”;
 - c. deberá almacenar además del texto de índice la página del libro donde está dicho tema;
 - d. determinar cuántos capítulos tiene;
 - e. determinar todos los temas que contengan las palabras modelo y métrica.
4. Implementar un algoritmo que contemple dos funciones, la primera que devuelva el hijo derecho de un nodo y la segunda que devuelva el hijo izquierdo.

5. Dado un árbol con los nombre de los superhéroes y villanos de la saga *Marvel Cinematic Universe* (MCU), desarrollar un algoritmo que contemple lo siguiente:
 - a. además del nombre del superhéroe, en cada nodo del árbol se almacenará un campo booleano que indica si es un héroe o un villano, *True* y *False* respectivamente;
 - b. listar los villanos ordenados alfabéticamente;
 - c. mostrar todos los superhéroes que empiezan con *C*;
 - d. determinar cuántos superhéroes hay el árbol;
 - e. Doctor Strange en realidad está mal cargado. Utilice una búsqueda por proximidad para encontrarlo en el árbol y modificar su nombre;
 - f. listar los superhéroes ordenados de manera descendente;
 - g. generar un bosque a partir de este árbol, un árbol debe contener a los superhéroes y otro a los villanos, luego resolver las siguientes tareas:
 - I. determinar cuántos nodos tiene cada árbol;
 - II. realizar un barrido ordenado alfabéticamente de cada árbol.
6. Dado un archivo con todos los Jedi, de los que se cuenta con: nombre, especie, año de nacimiento, color de sable de luz, ranking (Jedi Master, Jedi Knight, Padawan) y maestro, los últimos tres campos pueden tener más de un valor. Escribir las funciones necesarias para resolver las siguientes consignas:
 - a. crear tres árboles de acceso a los datos: por nombre, ranking y especie;
 - b. realizar un barrido inorden del árbol por nombre y ranking;
 - c. realizar un barrido por nivel de los árboles por ranking y especie;
 - d. mostrar toda la información de Yoda y Luke Skywalker;
 - e. mostrar todos los Jedi con ranking “Jedi Master”;
 - f. listar todos los Jedi que utilizaron sable de luz color verde;
 - g. listar todos los Jedi cuyos maestros están en el archivo;
 - h. mostrar todos los Jedi de especie “Togruta” o “Cerean”;
 - i. listar los Jedi que comienzan con la letra *A* y los que contienen un “-” en su nombre.

7. Partiendo del árbol n-ario del directorio, que se observa en la siguiente figura, implementar los algoritmos necesarios para poder transformarlo a un árbol binario no balanceado –utilizando la transformada de Knuth–. Tener en cuenta que los archivos serán nodos hojas –es decir que estos no pueden tener hijos– y además resolver las siguientes actividades:
- el nodo deberá tener un campo que indique si es un directorio o un archivo;
 - realizar un barrido inorden del árbol;
 - listar el contenido de la carpeta /Imágenes;
 - contar cuantos archivos hay en cada carpeta;
 - mostrar todos los archivos



- Desarrollar un algoritmo que implemente dos funciones, una para obtener el mínimo nodo del árbol y la segunda para obtener el máximo.
- Poe Dameron, líder del escuadrón negro de la Resistencia, tiene dificultades para transmitir los mensajes a la base de la Resistencia, dado que los mismos son muy largos y los satélites espías de la Primera Orden los intercepta, en un lapso muy corto desde que se transmiten. Por lo cual, nos solicita desarrollar un algoritmo que permita comprimir los mensajes para enviarlos más rápido y no puedan ser interceptados. Contemplando los siguientes requerimientos, implementar un algoritmo que los resuelva:

- a. crear un árbol de Huffman a partir de la siguiente tabla:

Símbolo	Frecuencia
A	0.2
F	0.17
I	0.13
3	0.21
O	0.05
M	0.09
T	0.15

- b. desarrollar las funciones para comprimir y descomprimir un mensaje.
10. Desarrollar dos algoritmos, el primero que permita calcular en el número de nodos de un nivel del árbol –a partir de un nivel ingresado–. La segunda que cuente los nodos que hay en dicho nivel –dado que podría no estar completo–, para responder las siguientes actividades:
- determinar si el nivel del árbol está completo;
 - ¿cuántos nodos faltan para completar dicho nivel?
11. Escribir un algoritmo que permita resolver las siguientes actividades:
- contar el número de nodos del árbol;
 - determinar el número de hojas del árbol;
 - mostrar la información de los nodos hojas;
 - determinar el padre de un nodo;
 - determinar la altura de un árbol.
12. Generar un árbol binario que tenga nueve niveles, luego diseñar los algoritmos necesarios para resolver las siguientes actividades:
- generar un bosque cortando los tres primeros niveles del árbol;
 - contar cuántos nodos tiene cada árbol del bosque;
 - realizar un barrido preorden de cada árbol del bosque;
 - determinar cuál es el árbol con mayor cantidad de nodos;
 - indicar que árboles del bosque están completos.

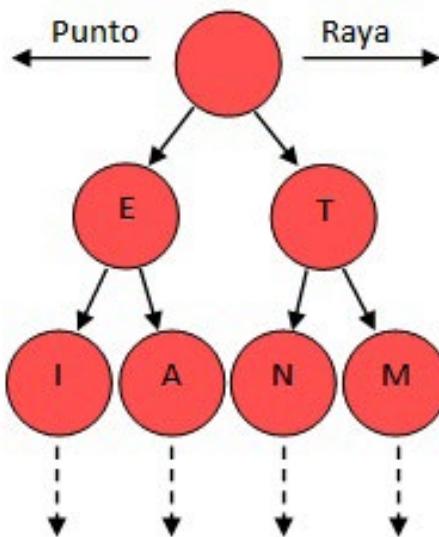
13. Nick Fury, líder de la agencia S.H.I.E.L.D., tiene la difícil tarea de decidir qué vengador asignará a cada nueva misión –por ahora considere que solo se asignará un superhéroe por cada misión–. Por lo que nos solicita desarrollar un árbol de decisión para resolver esta tarea con los siguientes requerimientos:
- a. cada nodo hoja del árbol debe ser un superhéroe y en cada nodo intermedio inclusive el raíz debe haber una pregunta;
 - b. si la respuesta es sí, se debe desplazar hacia el subárbol izquierdo, si es no al subárbol derecho;
 - c. desarrollar una función que determine el superhéroes para una misión;
 - d. los Guardianas de la Galaxia son ideales para misiones intergalácticas en equipo;
 - e. Ant-Man es excelente en misiones de recuperación donde sea necesario no se detectado;
 - f. para misiones de destrucción Hulk es una excelente opción;
 - g. el Capitán América es un supersoldado de ética incorruptible ideal para comandar misiones de defensa y de recuperación;
 - h. Capitana Marvel es muy poderosa y puede viajar por las distintas galaxias;
 - i. Spider-Man es muy hábil y puede ser útil para varias misiones;
 - j. para misiones de recuperación donde requiera infiltrarse con personas del lugar, Black Widow es la indicada;
 - k. Iron Man es un líder para planear misiones de defensa, además es un genio y domina el manejo de tecnología avanzada, cuenta con un traje muy poderoso;
 - l. cuando se requiere elegir cuál será la próxima acción a tomar y moverse rápidamente de un lugar a otro, Doctor Strange es la opción más lógica;
 - m. Thor tiene el poder para destruir ejércitos completos;
 - n. no se debe utilizar árbol balanceado.

14. Desarrollar un algoritmo que permita decodificar mensajes en código morse, para ello deberá resolver las siguientes consignas:

 - generar un árbol que contenga todo el alfabeto y los dígitos del 0 al 9, cuyos códigos morse están en la siguiente figura:

A ● -	G --- ●	M ---	S ● ● ●	Y - ● -	S ● ● ● ●
B - ● ● ●	H ● ● ● ●	N - ●	T -	Z - - ● ●	6 - ● ● ● ●
C - ● - ●	I ● ●	Q - - -	U ● ● -	1 ● - - -	7 - - ● ● ●
D - ● ●	J ● - - -	P ● - - ●	V ● ● ● -	2 ● ● - -	8 - - - ● ●
E ●	K - ● -	Q - - - ●	W ● - -	3 ● ● ● -	9 - - - - ●
F ● ● - ●	L ● - - ● ●	R ● - - ●	X - ● ● -	4 ● ● ● ● -	0 - - - - -

- b. cada nodo del árbol contendrá una letra o un dígito, el cual se debe construir de manera manual (en un árbol que no sea auto balanceado), cuya raíz es vacía y, a partir de esta, la izquierda significa punto y la derecha guion –y se cargaran según su codificación morse, como se observa en la siguiente figura:



- c. el espacio separa cada letra de una palabra y la “/” separa cada palabra;
 - d. descifrar los siguientes mensajes:

Mensaje 1 (Dr. Abraham Erskine): .-. - ... / ... --- / --. ... / .--. - ... / --. - - .- / -. - ---
--- - - .- / .- ... - - --- / --. - .. / - - - - .- / - - .- --- / ... - - - .- / - - .- / .. - / ... - - - - .-
. - - - / --. - - - - .- / ... - - - - / ... - - - - / ... - - - - / ... - - - -

Mensaje 3 (Natasha Romanoff): -.- --- / ... --- .-.. --- / .-.-. - .. --- / -. --- - - - / / .-. / ...- .-.. - .. / ... --- /-.-.-.- / - - - -.. --- ..-.

Mensaje 4 (Tony Stark): -.--.. --- ... / - - - -.- / .-.-. - .. --- / .-.. - / ..-..... -.- / - - - -.- / ..- .. - .- - .. - - -.

Mensaje 5 (Steve Rogers): -.-. --- -...-....- / - - - -.- / - - - / - - - .. --- / .-.. / - - - - -.-.

15. Desarrollar un algoritmo que permita implementar un árbol como índice para hacer consultas a un archivo que contiene personajes de la saga de Star Wars, de los cuales se sabe su nombre, altura y peso. Además deberá contemplar los siguientes requerimientos:
 - a. en el árbol se almacenara solo el nombre del personaje, además de la posición en la que se encuentra en el archivo (*nrr*);
 - b. se debe poder cargar un nuevo personaje, modificarlo (cualquiera de sus campos) y darlo de baja;
 - c. mostrar toda la información de Yoda y Boba Fett;
 - d. mostrar un listado ordenado alfabéticamente de los personajes que miden más de 1 metro;
 - e. mostrar un listado ordenado alfabéticamente de los personajes que pesan menos de 75 kilos;
 - f. deberá utilizar el TDA archivo desarrollado en el capítulo V;
16. Una empresa de nano satélites dedicada al monitoreo de lotes campo destinados al agro, tiene problemas para la transmisión de los datos recolectados, dado que la ventana de tiempo que dispone para enviar los datos antes de una nueva medición es muy corta, por lo que nos solicita desarrollar un algoritmo que permita comprimir la información para poder enviarla más rápido, para lo cual se debe tener en cuenta los siguientes requerimientos:
 - a. la información transmitida por el nano satélite son estado del tiempo, humedad del suelo, y tres dígitos que identifican el lote al cual pertenecen los datos;
 - b. desarrollar un árbol de Huffman que permita comprimir la información para transmitirla, la frecuencia de la información transmitida se observa en la siguiente tabla:

Variable	Símbolo	Frecuencia
Estado del clima	Despejado	0.22
	Nublado	0.15
	Lluvia	0.03
Humedad del suelo	Baja	0.26
	Alta	0.14
Código de identificación 1	1	0.05
	2	0.01
Código de identificación 2	3	0.035
	5	0.06
Código de identificación 2	7	0.02
	8	0.025

- c. comprimir un mensaje y descomprimirlo, para ver si no se pierde información durante el proceso de codificación, la trama enviada por el nano satélite tiene el siguiente formato (estado del clima-humedad del suelo-cod1-cod2-cod3), por ejemplo la siguiente trama es válida “Nublado-Baja-1-5-7”, –los guiones son a fines de comprender como está formada la trama pero no forman parte de la misma–;
- d. determinar la diferencia en tamaño de memoria utilizada por la trama original y la trama comprimida –puede utilizar la función getsizeof() de la librería sys–.
17. Se tiene un archivo con los Pokémons de las 8 generaciones cargados de manera desordenada (890 en total) de los cuales se conoce su nombre, número, tipo/tipos, debilidad frente a tipo/tipos, para el cual debemos construir tres árboles para acceder de manera eficiente a los datos almacenados en el archivo, contemplando lo siguiente:
- los índices de cada uno de los árboles deben ser nombre, número y tipo;
 - mostrar todos los datos de un Pokémon a partir de su número y nombre –para este último, la búsqueda debe ser por proximidad, es decir si busco “bul” se deben mostrar todos los Pokémons cuyos nombres comiencen o contengan dichos caracteres–;
 - mostrar todos los nombres de todos los Pokémons de un determinado tipo agua, fuego, planta y eléctrico;
 - realizar un listado en orden ascendente por número y nombre de Pokémon, y además un listado por nivel por nombre;
 - mostrar todos los Pokémons que son débiles frente a Jolteon, Lycanroc y Tyrantrum;
 - mostrar todos los tipos de Pokémons y cuántos hay de cada tipo.

18. La armería de la base Starkiller, central de la primera orden, almacena los registros de los reportes de fallos en armas de las tropas de sus principales generales –Kylo Ren, general Hux y capitana Phasma–. Para dicha labor, se solicita desarrollar un algoritmo que permita resolver las siguientes tareas:
- a. se debe registrar el nombre del general a cargo de la misión, fecha de la misión –a los fines del ejercicio considere como máximo 20 fechas de misiones–, código de blaster generado de manera aleatoria –de 8 dígitos y no puede estar repetido–, estado del blaster (si falló o no) y el tipo de soldado que portaba el blaster –Imperial Stromtrooper, Imperial Scout Trooper, Imperial Death Trooper, Sith Trooper o First Order Stromtrooper–;
 - b. debe generar y cargar al menos 10 000 registros;
 - c. determinar el total de armas que fallaron por general;
 - d. indicar la cantidad y tipo de soldado de las misiones de Kylo Ren;
 - e. determinar cuántos Sith Troopers salieron en misiones y a cuantos les fallaron los blasters;
 - f. listar los códigos de los blasters de las misiones de una determinada fecha, indicando además el porcentaje de armas que fallaron;
 - g. mostrar los datos del blaster código “75961380” si fue utilizado en alguna misión.
19. Desarrollar los algoritmos necesarios que permitan almacenar libros, de los cuales se conoce su título, ISBN, autores, editorial y cantidad páginas en un archivo, contemplando los siguientes requerimientos y tareas:
- a. utilizar el TDA archivo desarrollado en el capítulo V;
 - b. deberá cargar al menos 100 libros;
 - c. implementar tres árboles para manejar los índices de acceso, estos serán por título, ISBN y autores. En cada nodo del árbol se almacenará el campo clave correspondiente y la posición en el archivo donde está el resto de la información;
 - d. las búsquedas deberán ser de la siguiente manera:
 - I. por exactitud en el árbol de ISBN,
 - II. que esté contenido en el árbol de autores –es decir si son más de un autor y busco por uno debería encontrarlo–,
 - III. por proximidad en el inicio del nombre en el árbol de título –si busco “Alg” debería encontrar todos los libros cuyo nombres comienzan así–.

20. Ahora resuelva las siguientes consultas mostrando toda la información de los libros correspondiente:

- a. los libros de los autores Tanenbaum, Connolly, Rowling, Riordan, Morgan Kass;
- b. mostrar los libros de “minería de datos”, “algoritmos” y “bases de datos”;
- c. mostrar los libros de más de 873 páginas;
- d. mostrar los datos del libro ISBN 9789504967453;
- e. mostrar el autor del libro “los 100”.

21. Implementar un algoritmo que permita generar un árbol de decisión meteorológico para la predicción del estado del tiempo basado en las reglas de la siguiente figura, considerando los siguientes requerimientos:

- a. los 5 posibles estados del tiempo son despejado, parcialmente nublado, mayormente nublado, nublado y lluvia;
- b. los nodos hojas del árbol representan los estados del tiempo que predice el árbol –en la figura están con negrita–;
- c. en cada nodo deberá almacenar el nombre de la variable o campo que se utilizará en ese nodo para la decisión y el valor umbral: se avanza hacia la izquierda si el valor es menor o igual y se avanza a la derecha cuando el valor es mayor;
- d. dado un nuevo registro con datos meteorológicos del cual se conoce temperatura, presión, humedad, visibilidad, velocidad del viento, se debe predecir el estado del tiempo de manera automática.

```
visibilidad <= 15
|   humedad <= 70
|   |   viento <= 8.7
|   |   |   viento <= 5: Despejado
|   |   |   viento > 5: Nublado
|   |   viento > 8.7: Parcialmente nublado
humedad > 70
|   visibilidad <= 8
|   |   presión <= 1013
|   |   |   humedad <= 96: Nublado
|   |   |   humedad > 96: Mayormente nublado
|   |   presión > 1013
|   |   |   viento <= 7.2
|   |   |   |   presión <= 1018
|   |   |   |   |   visibilidad <= 1: Lluvia
|   |   |   |   |   visibilidad > 1: Mayormente nublado
|   |   |   |   presión > 1018: Nublado
|   |   |   viento > 7.2: Nublado
|   |   visibilidad > 8
|   |   |   humedad <= 92
|   |   |   |   visibilidad <= 12: Despejado
|   |   |   |   |   visibilidad > 12: Mayormente nublado
|   |   |   humedad > 92
|   |   |   |   viento <= 12.2: Lluvia
|   |   |   |   viento > 12.2: Nublado
|   |   visibilidad > 15: Despejado
```

22. Parta de la base del árbol genealógico “*greek-gods*” (n-arios) que se observa en el siguiente link: <https://drive.google.com/file/d/13lMB6A2k4zO2zq-wuTgdxZdgwUKLl4Tr/view?usp=sharing>, y utilice la transformada de Knuth para convertirlo en un árbol binario que permita realizar las siguiente actividades (no se deben utilizar árboles balanceados):
- a. la raíz del árbol debe ser Urano;
 - b. además del nombre de los dioses, deberá cargar una breve descripción de quien es o lo que representa, no más de 20 palabras;
 - c. listar el árbol por niveles, es decir, mostrando primero los hermanos. Para esto desarrolle una función barrido llamada “hermanos(raíz)” que devuelva todos los hijos derechos de un determinado nodo de un árbol general transformado a binario;
 - d. solo se representarán las relaciones padre-hijo, a excepción de los dioses que en la imagen no tengan padre, –en este caso se deberá cargar la relación madre-hijo–, en los demás la madre será almacenada en un campo del nodo;
 - e. dado el nombre de un dios mostrar los hijos de este;
 - f. dado el nombre de un dios mostrar su nombre, padre, madre, hermanos y sus hijos;
 - g. realizar un barrido inorden, preorden y por nivel de dicho árbol;
 - h. realizar un barrido inorden mostrando el nombre de cada dios y el de su madre;
 - i. mostrar todos los ancestros de un determinado dios;
 - j. generar un bosque eliminando el nodo Urano:
 - I. determinar cuántos árboles forman dicho bosque,
 - II. realizar un barrido inorden de cada árbol del bosque,
 - III. determinar cuántos nodos hay en cada árbol y cuál es el nombre del dios del nodo raíz del árbol más grande;
 - k. mostrar todos los hijos de Tea.
23. Implementar un algoritmo que permita generar un árbol con los datos de la siguiente tabla y resuelva las siguientes consultas:
- a. listado inorden de las criaturas y quienes la derrotaron;
 - b. se debe permitir cargar una breve descripción sobre cada criatura;
 - c. mostrar toda la información de la criatura Talos;

- d. determinar los 3 héroes o dioses que derrotaron mayor cantidad de criaturas;
- e. listar las criaturas derrotadas por Heracles;
- f. listar las criaturas que no han sido derrotadas;
- g. además cada nodo debe tener un campo “capturada” que almacenará el nombre del héroe o dios que la capturo;
- h. modifique los nodos de las criaturas Cerbero, Toro de Creta, Cierva Cerinea y Jabalí de Erimanto indicando que Heracles las atrapó;
- i. se debe permitir búsquedas por coincidencia;
- j. eliminar al Basilisco y a las Sirenas;
- k. modificar el nodo que contiene a las Aves del Estíñfalo, agregando que Heracles derroto a varias;
- l. modifique el nombre de la criatura Ladón por Dragón Ladón;
- m. realizar un listado por nivel del árbol;
- n. muestre las criaturas capturadas por Heracles.

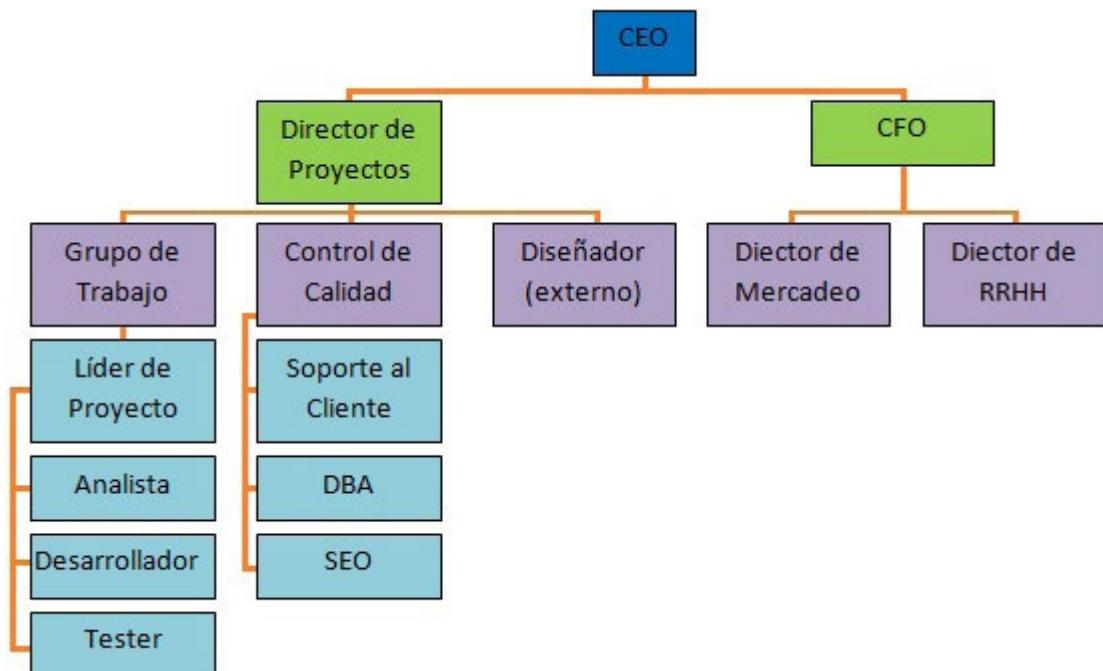
Criaturas	Derrotado por	Criaturas	Derrotado por
Ceto	-	Cerda de Cromión	Teseo
Tifón	Zeus	Ortro	Heracles
Equidna	Argos Panoptes	Toro de Creta	Teseo
Dino	-	Jabalí de Calidón	Atalanta
Pefredo	-	Carcinos	-
Enio	-	Gerión	Heracles
Escila	-	Cloto	-
Caribdis	-	Láquesis	-
Euríale	-	Átropos	-
Esteno	-	Minotauro de Creta	Teseo
Medusa	Perseo	Harpías	-
Ladón	Heracles	Argos Panoptes	Hermes
Águila del Cáucaso	-	Aves del Estíñfalo	-
Quimera	Belerofonte	Talos	Medea
Hidra de Lerna	Heracles	Sirenas	-
León de Nemea	Heracles	Pítón	Apolo
Esfinge	Edipo	Cierva de Cerinea	-
Dragón de la Cólquida	-	Basilisco	-
Cerbero	-	Jabalí de Erimanto	-

- c. finalmente, calcule el espacio de memoria requerido por el mensaje original y el comprimido.

Carácter	Cantidad	Frecuencia
A	II	
B	2	
C	4	
D	3	
E	I4	
G	3	
I	6	
L	6	
M	3	
N	6	
O	7	
P	4	
Q	I	
R	I0	
S	4	
T	3	

U	4	
V	2	
' espacio	17	
, coma	2	

25. A partir del organigrama de un empresa de desarrollo que se presenta en la siguiente figura (árbol n-ario) implementar las funciones necesarias para resolver los siguientes requerimientos:
- utilizar la transformada de Knuth para convertirlo en un árbol binario;
 - realizar un barrido inorden y peor nivel de dicho árbol;
 - agregue una persona a cada puesto de la empresa, salvo a los puesto de desarrollador, tester y soporte al cliente, en estos cargue cinco personas;
 - mostrar todos los empleados dependiente del líder de proyecto;
 - mostrar todo los empleados dependientes directamente del director de proyectos;
 - mostrar todos los empleados del nivel tres del organigrama (recuerde que la raíz es el nivel uno).



Pintando nodos de color rojo y negro, ¡conozcamos los árboles rojinegros!

Por un momento continuaremos estudiando más respecto de árboles, en particular de árboles binarios. Como la mayoría de las cosas en el área de desarrollo existen más de una manera de implementarlas, y las estructuras de datos no son la excepción. En esta ocasión veremos otro tipo de árbol binario de búsqueda que tiene la capacidad de auto balancearse, este tipo de árbol utiliza el color de sus nodos como factor de equilibrio. Pero en esencia su comportamiento es similar al árbol AVL visto en el capítulo anterior.

Un árbol rojo-negro es un árbol binario de búsqueda equilibrado (como el AVL visto en el capítulo anterior), esta estructura de datos fue creada originalmente por Rudolf Bayer¹ en 1972, que le dio el nombre de “árboles-B binarios simétricos”, pero la estructura de árbol rojo-negro fue creada por Leonidas Guibas y Robert Sedgewick en 1978² inspirados en el trabajo de Bayer. Este árbol sin embargo es una estructura que debe apegarse a un conjunto muy estricto de reglas para asegurar su auto equilibrio. Precisamente, estas reglas nos permiten garantizar su complejidad de tiempo logarítmico.

Básicamente estas son las cuatro reglas que deben seguirse pase lo que pase, cuando realizamos una operación de inserción y eliminación, de lo contrario no es considerado un árbol rojo-negro y además dejaría de estar equilibrado:

1. Cada nodo en el árbol debe ser rojo o negro.
2. El nodo raíz del árbol debe ser siempre negro.
3. Dos nodos rojos nunca pueden aparecer consecutivamente, uno tras otro. Es decir un nodo rojo siempre debe tener un nodo padre negro e hijos negros.
4. Cada ruta de bifurcación del árbol desde el nodo raíz a un nodo hoja vacío (o nulo) debe pasar por el mismo número exacto de nodos negros. Los nodos hojas vacíos también se deben considerar como un nodo negro del camino, ya que representa la ruta que se tomará si se busca un nodo que no existe dentro del árbol.

La figura I muestra un árbol rojo-negro que cumple las cuatro reglas. Observe que siempre las hojas del árbol serán vacías (*Null* o *None* en el caso de Python) y de color negro. Si bien no es necesario dibujarlas hay que saber que están ahí y cuentan como un nodo negro –esto nos ayudará a cumplir la tercera y cuarta regla–.

¹ Rudolf Bayer (1972). "Symmetric binary B-Trees: Data structure and maintenance algorithms". *Informatic Record*. 1 (4): 290–306.

² Leonidas J. Guibas and Robert Sedgewick (1978). "A Dichromatic Framework for Balanced Trees". *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. pp. 8–21.

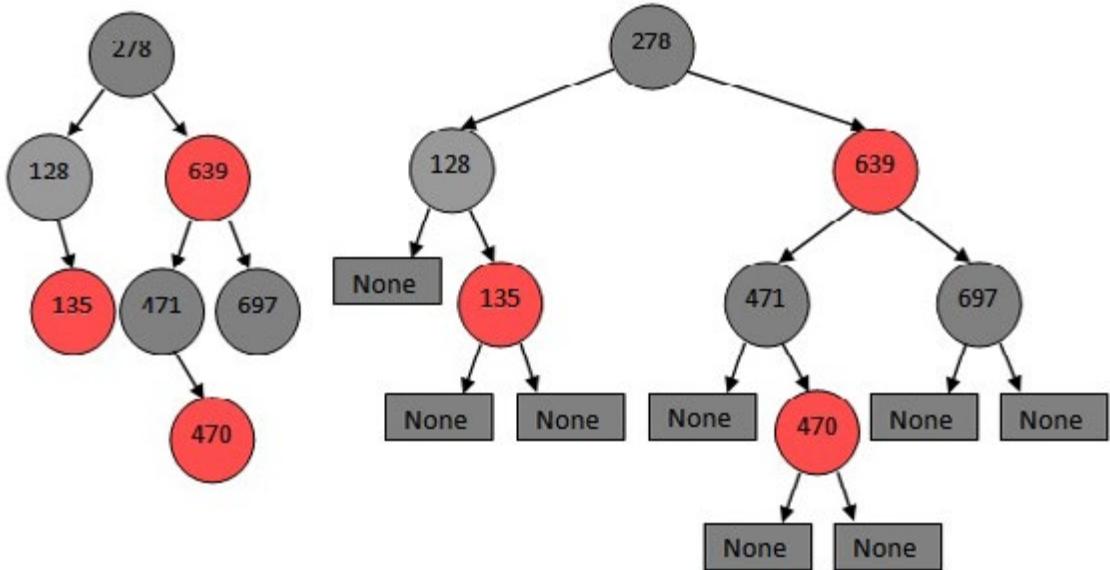


Figura 1. Árbol rojo-negro

Muchos de ustedes se estarán preguntado *¿Por qué los colores rojo y negro? ¿Por qué no otros colores?*, estas preguntas también se las hicieron a los inventores de este árbol y estas fueron sus respuestas:

“[...] fue debido a que los bolígrafos disponibles en aquel entonces para poder dibujar árboles eran rojos y negros [...]” (Leonidas Guibas, inédito).

Por su parte Robert Sedgewick respondió:

Bueno, inventamos esta estructura de datos, esta forma de ver árboles equilibrados, en una Xerox PARC, que fue la computadora personal y muchas otras innovaciones con las que vivimos hoy al ingresar a las interfaces gráficas de usuario, Ethernet y programaciones orientadas a objetos y muchas otras cosas. Pero una de las cosas que se inventó fue la impresión láser y estábamos muy entusiasmados de tener una impresora láser a color cerca que pudiera imprimir las cosas en color y de los colores el rojo se veía mejor. Entonces, es por eso que elegimos el color rojo para distinguir los enlaces rojos, los tipos de enlaces, en los nodos de árbol (Robert Sedgewick, inédito).

Ahora vamos a describir cómo es el procedimiento de inserción para comprender cómo funcionan estas reglas, primero debemos determinar la ubicación en el árbol de la misma manera que vimos en el capítulo anterior –recordemos que siempre insertamos hojas–. Este proceso al principio parece un poco más complejo de entender respecto de los árboles AVL, pero tranquilos iremos paso a paso para que sea más sencillo. El primer interrogante a resolver *¿De qué color debe ser el nodo que insertamos?* La estrategia más sencilla es siempre insertar un nodo rojo, esto permite que sea mucho más fácil identificar cualquier infracción a una de las reglas, para luego poder corregirlo. Entonces, si se rompió una de las reglas, *¿cómo lo corregimos?* Para esto utilizaremos dos técnicas: coloreo y rotación.

Para analizar los posibles casos de infracción a las reglas que pueden ocurrir durante la inserción, trabajaremos con el siguiente ejemplo: supongamos que tenemos un conjunto de Pokémon de los que conocemos su número y nombre, los cuales queremos insertar en un árbol rojo-negro ordenado por su número. Estos son los datos con los que trabajaremos: 278 Wingull, 128 Tauros, 470 Leafeon, 639 Terrakion, 126 Magmar, 127 Pinsir.

Comenzamos cargando el primer elemento 278 Wingull, como ya mencionamos, siempre insertamos nodos rojos. Como podemos ver en la figura 2 ya estamos rompiendo la segunda regla –la raíz del árbol debe ser de color negro–, para reparar este caso si el nodo no tiene padre, es decir es la raíz, utilizamos la técnica de coloreo y pintamos el nodo de color negro para cumplir con todas las regla.



Figura 2. Ruptura y reparación regla 2

Luego, procedemos a ingresar los elementos siguientes dos elementos del conjunto 128 Tauros y 471 Leafeon, como observamos en la figura 3 no se rompe ninguna de las reglas.

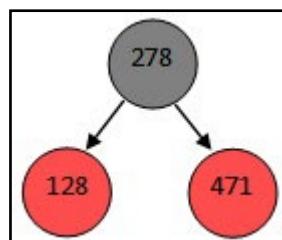


Figura 3. Inserción sin infringir reglas

Continuando con el ejemplo, al insertar el cuarto elemento 639 Terrakion se rompe la tercera regla, es decir no puede haber dos nodos rojos seguidos. Para reparar este caso, cuando tanto el padre como el tío del nodo insertado son rojos, aplicamos coloreo para cambiar el color del padre y el tío del nodo insertado a negro y el del abuelo a rojo. Al hacer esto, el nodo raíz queda infringiendo la segunda regla. Como la inserción es recursiva la reparación también, entonces esta se hace desde la hoja insertada hasta la raíz. Así que, por último, se aplica coloreo nuevamente para reparar la regla rota por la raíz del árbol, este proceso se presenta en la figura 4.

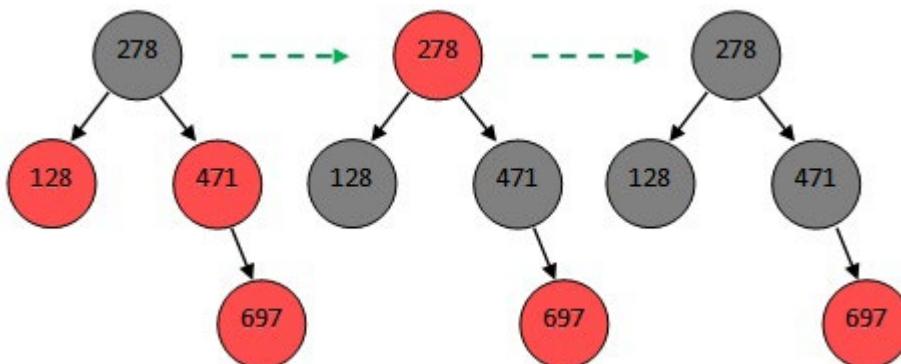


Figura 4. Ruptura y reparación con coloreo de regla 3

Ahora es momento de insertar el quinto elemento 745 Lycanroc, esto rompe nuevamente la tercera regla. Pero en este caso el padre es rojo pero el tío no, por lo cual no lo podremos corregir solo coloreando, por lo que se debe utilizar además la técnica de rotación. Entonces usamos coloreo para pintar el padre del nodo de color negro y el abuelo de rojo, luego aplicaremos rotación simple a la izquierda del abuelo como se detalla en la figura 5 para finalizar la reparación –nótese que tanto el nodo como el padre son hijos derechos-. Estas rotaciones son similares a las utilizadas en árboles AVL pero debemos contemplar algunas cuestiones que veremos más adelante.

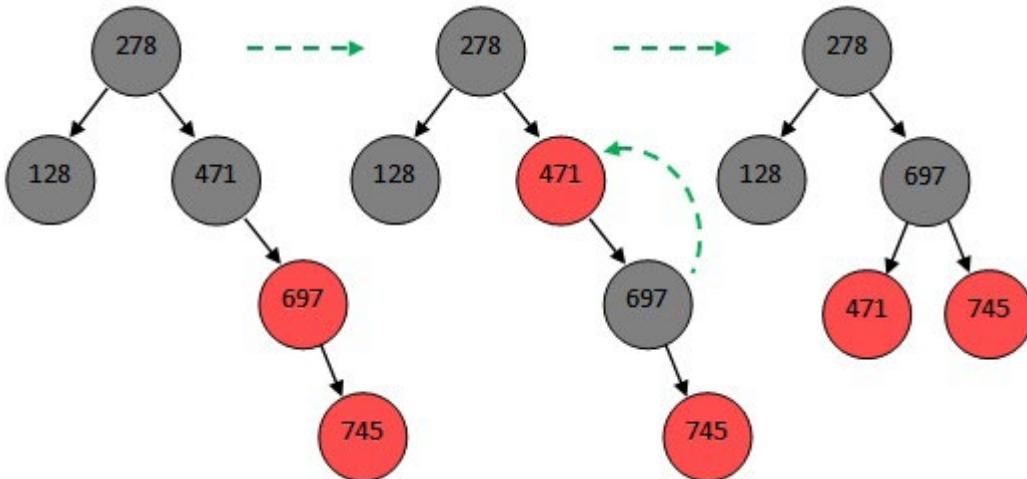


Figura 5. Ejemplo reparación usando coloreo y rotación

Sigamos cargando más elementos. Ahora es turno del 126 Magmar y 127 Pinsir. Después de agregar el segundo elemento volvemos a infringir la regla 3, como sucedió en el caso anterior el padre es rojo pero el tío no, así que solo colorear no repara el árbol. A diferencia del caso anterior el nodo es hijo derecho y el padre es hijo izquierdo por lo que una rotación simple tampoco nos alcanza, así que para repararlo debemos usar dos rotaciones simples y coloreo de la siguiente manera: primero rotamos al padre a la izquierda, luego coloreamos el padre de negro y al abuelo de rojo, y finalmente rotamos el abuelo a la derecha para terminar de reparar el árbol. Estas acciones se representan en la figura 6.

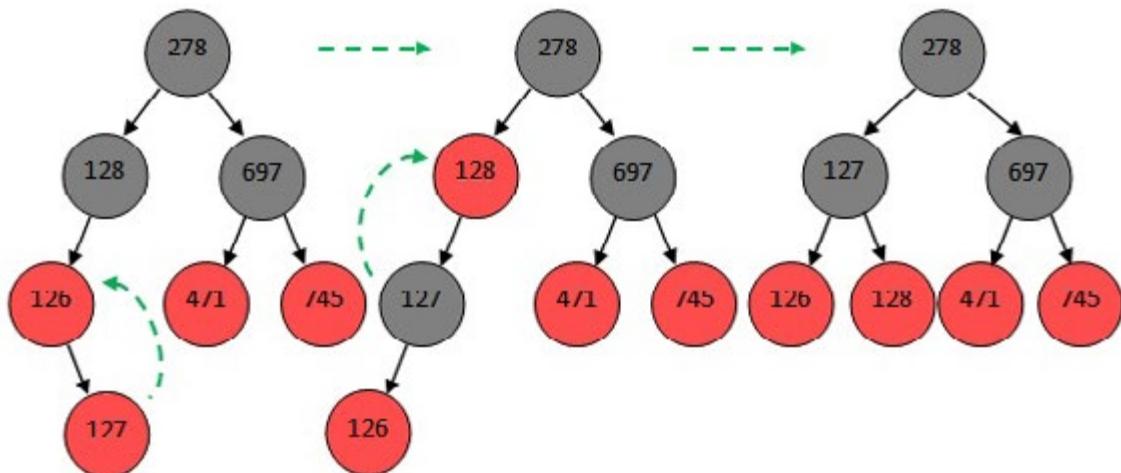


Figura 6. Ejemplo reparación usando coloreo y dos rotaciones

Además los casos vistos anteriormente que rompen las reglas –a excepción del primero–, tienen opuestos simétricos, es decir que se solucionan de la misma manera pero invirtiendo las rotaciones. Queda a cargo del lector hacer la representación para comprobar dichos casos.

Continuemos detallando lo que ocurre cuando se elimina un nodo del árbol. En primer lugar, se debe determinar si el valor que se pretende eliminar está en el árbol. Recordemos que si el nodo a eliminar tiene ambos hijos debemos buscar un nodo hoja para remplazar dicho valor y luego quitar dicha hoja como se explicó en el capítulo anterior. Por ello, debemos considerar el color del nodo que se quita, si el nodo a eliminar es rojo, no se rompen ninguna de las reglas del árbol rojo-negro, por lo cual queda equilibrado y simplemente se elimina el nodo al igual que en un abb y se acomodan los enlaces correspondientes. Pero en el caso contrario, es decir, que el color del nodo sea negro, al quitarlo se rompe la cuarta regla y debemos reparar el árbol de manera similar a como lo hicimos en la inserción, detallaremos este proceso a continuación describiendo cada caso con un ejemplo.

Cuando quitemos un nodo negro ocurrirá una de las siguientes situaciones: que el nodo a eliminar solo tenga un hijo izquierdo, en este caso se remplaza el nodo a eliminar por el nodo hijo y si este último es rojo tenemos que pintarlo de negro como se observa en la figura 7. Además esta situación tiene un opuesto simétrico que queda a cargo del lector verificarlo.

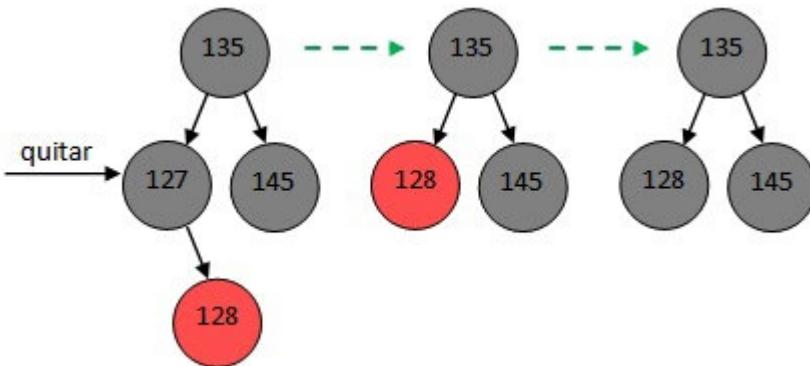


Figura 7. Eliminación nodo negro con un hijo

La segunda situación es que el nodo a eliminar no tenga hijo, lo que significa que es una hoja y por la propiedades del árbol sabemos que si o si tiene un hermano que puede ser rojo o negro –salvo que se al raíz que es un caso excepcional–, cuando ocurre esto se dice que estamos en presencia de un nodo “doble negro” y para poder reparar esto es necesario identificar cuál de los seis casos posibles ha ocurrido. Veamos en detalle cómo solucionar cada uno de estos casos de “doble negro” en algunas ocasiones deberemos aplicar más de un caso para lograr reparar el árbol, desde la hojas hasta la raíz.

El primer caso es que el doble negro sea la raíz. Cuando ocurre esto simplemente se descarta el doble negro porque es la raíz del árbol y se ha reparado el equilibrio, es decir hemos subido desde la hoja que rompió el equilibrio y ya no queda nada mas por reparar.

Luego el segundo caso es cuando el hermano del doble negro es rojo y su padre es negro. Lo primero que hacemos es una rotación a la derecha del padre, luego se colorean el padre de negro y el hermano de rojo, solucionando parcialmente el problema para terminar de repararlo con el cuarto caso. Esto lo podemos observar en la figura 8.

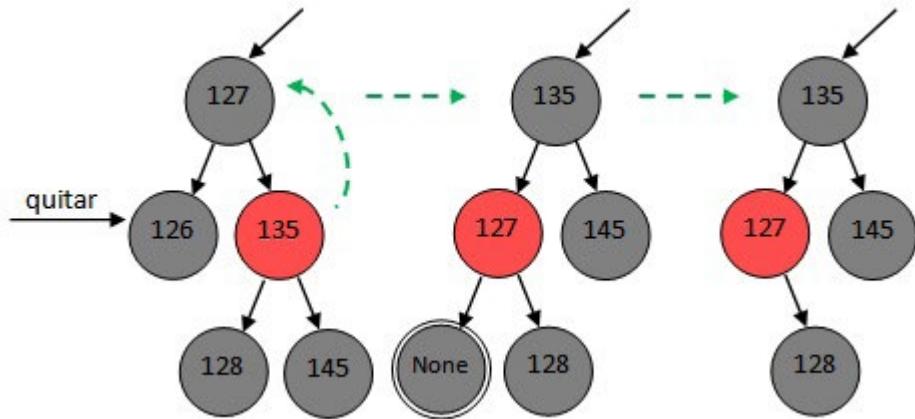


Figura 8. Ejemplo de eliminación caso 2

Por su parte en el tercer caso el hermano es negro y si tiene hijos son de color negro. Para solucionar esto se colorea el hermano de rojo con lo cual queda reparado como se aprecia en la figura 9.

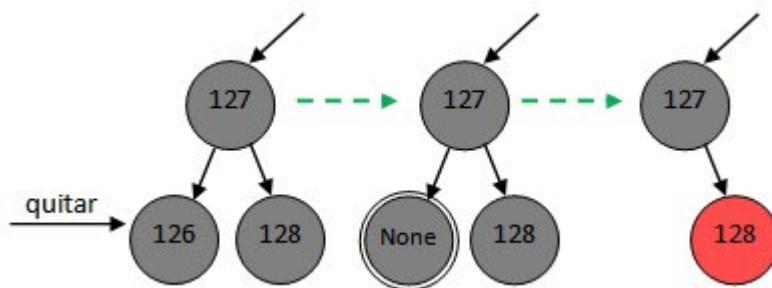


Figura 9. Ejemplo de eliminación caso 3

En cambio en el cuarto caso se da que el padre del doble negro es rojo. Esto se corrige coloreando el hermano de rojo y el padre de negro como podemos ver en la figura 10.

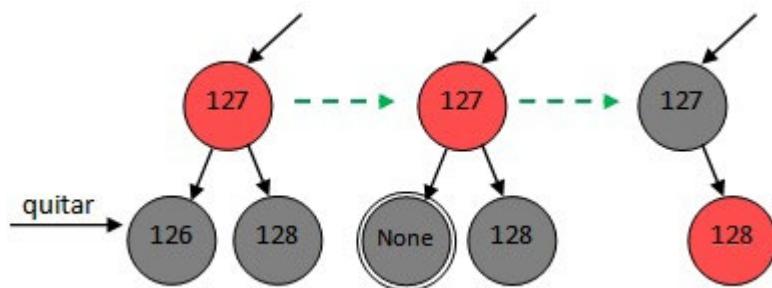


Figura 10. Ejemplo de eliminación caso 4

Seguimos con el quinto caso ocurre cuando el hijo izquierdo del hermano es rojo. Para reparar esto primero se realiza una rotación a la derecha del hermano, luego se colorea el hermano de rojo y el hijo izquierdo del hermano de negro, logrando una solución parcial como observa en la figura 11 que luego terminaremos de reparar con el caso 6.

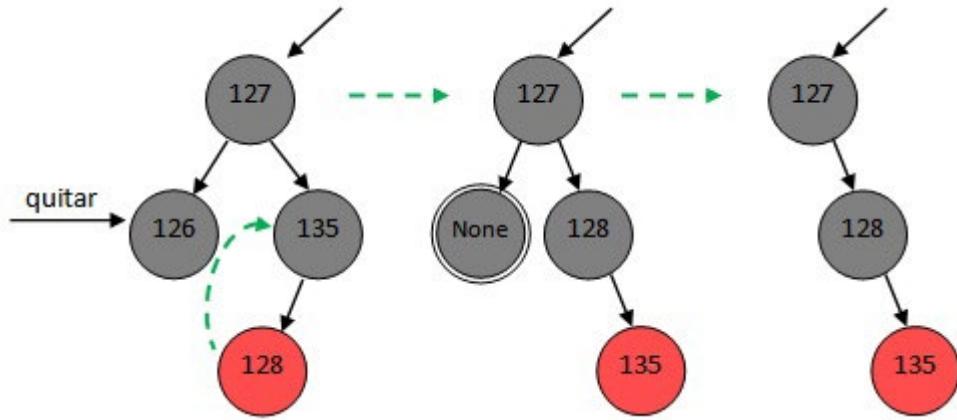


Figura II. Ejemplo de eliminación caso 5

Finalmente el sexto caso sucede cuando el hijo derecho del hermano es rojo. Sin importar el color que tenga el padre y el hijo izquierdo –por eso se observan con fondo rayado en la imagen–, dado que los nodos que ocupen su lugar mantendrán dicho color, para reparar esto se realiza una rotación a la izquierda del padre, luego se coloreamos: el padre de negro, el hermano del color que tenía el padre, el hijo derecho del hermano de negro, y el hijo izquierdo del hermano mantiene su color, como se ilustra en la figura 12.

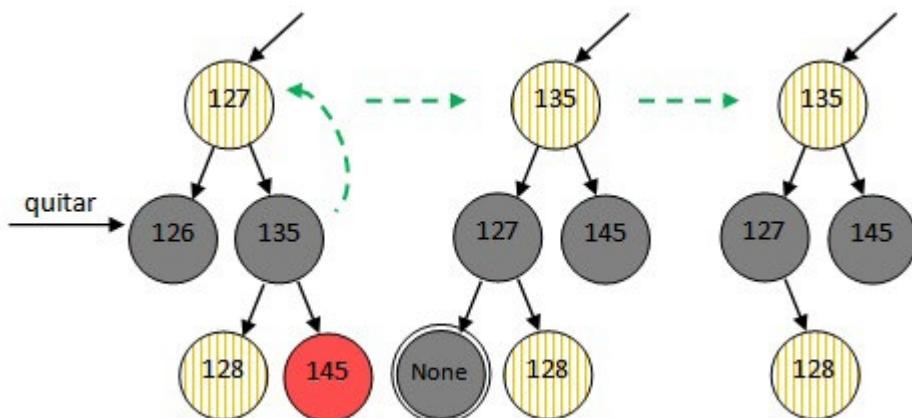


Figura 12. Ejemplo de eliminación caso 6

Además todos los casos vistos anteriormente –al igual que ocurrió en los caso de inserción– tienen sus casos opuestos simétricos a excepción del primero, quedará a cargo del lector hacer la representación correspondiente para comprobar dichos casos.

Nos resta hacer una evaluación de la estructura respecto de las actividades detalladas hasta este momento, si bien las operaciones de *inserción* y *eliminación* tienen cierta dificultad para comprender los casos que rompen las reglas y cómo corregirlo coloreando y rotando nodos, evaluemos desde la perspectiva del esfuerzo o costo de las actividades en un árbol rojo-negro:

Insertar, *eliminar* y *buscar* son operaciones en el orden de $O(\log n)$ al igual que en un árbol binario de búsqueda perfectamente balanceado, es decir tiempo logarítmico.

Colorear un nodo tiene un tiempo constante de orden $O(1)$, dado que solo implica cambiar el valor de una variable.

Rotar y colorear un nodo para acomodar la regla que rompa sigue manteniendo el orden $O(1)$, dado que solo cambiamos el valor de unas pocas variables –por lo que es despreciable–. Pero a veces, como vimos, es necesario corregir varios nodos incluso hasta llegar a la raíz lo que implicaría que el coste sería en el orden de $O(\log n)$. Igualmente sigue siendo muy bueno dado que es logarítmico en el peor de los casos.

La complejidad espacial de un árbol rojo-negro es de orden $O(n)$ de igual manera que para los árboles AVL.

Sin embargo, la representación o seguimiento del factor de equilibrio, en este caso el color de un nodo, solo requiere un bit de almacenamiento. Un bit es un solo dígito en sistema binario –que representa la menor unidad de almacenamiento en informática–, y solo eso necesitamos para representar el color de un nodo en el árbol, dado que solo puede tomar dos colores rojo (1) o negro (0), lo cual representa una ventaja significativa respecto a otros árboles balanceados.

Quizás el principal referente de uso de un árbol rojo-negro es en el “Planificador Completamente Justo” (CFS) del kernel de Linux, que fue introducido en 2007. Su objetivo es mantener el equilibrio al proporcionar tiempo a los procesos, donde cada proceso reciba la cantidad justa de tiempo de uso del procesador. El uso de árbol rojo-negro permite mejorar significativamente el rendimiento de la administración de los procesos para el uso del CPU frente a las colas que utilizaba el modelo anterior. En la figura 13 se puede ver un ejemplo de cómo se distribuyen los procesos en base al tiempo que necesita cada uno.

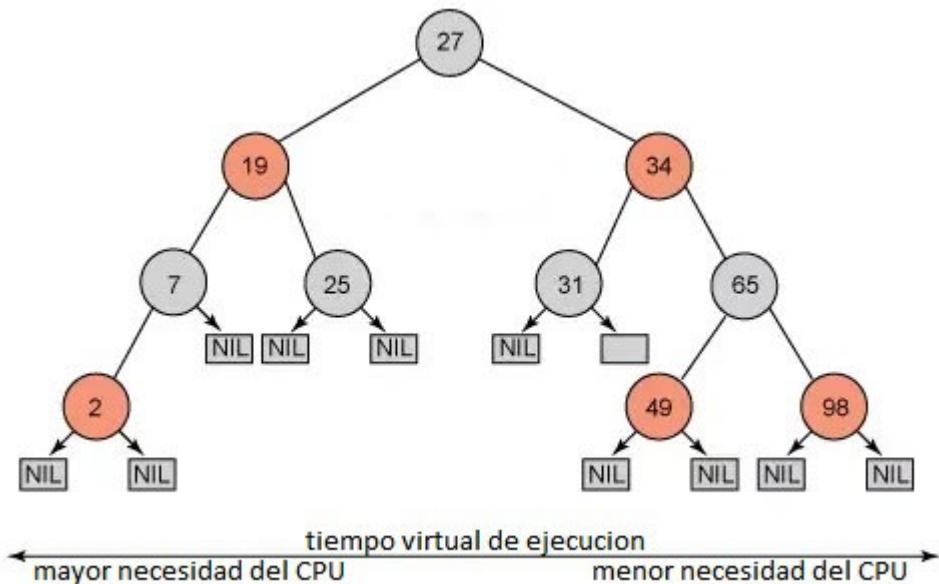


Figura 13. Árbol rojo-negro del CFS de Linux

Ya contamos con los conocimientos necesarios para empezar con la implementación del TDA árbol rojo-negro. Hemos visto las reglas que debe cumplir para mantener su equilibrio, los casos particulares que ocurren durante la *inserción* y *eliminación* que rompen dichas reglas, así como también

las técnicas de coloreo y rotación que nos permiten repararlo. Lo primero que haremos es definir el nodoArbolRN que a diferencia del nododArbol agrega los siguientes atributos “color” para poder representar el color del nodo y “padre” que hace referencia al padre de dicho nodo –este será esencial para poder reparar el árbol– como se puede ver en la figura 14. Para representar el color utilizaremos un byte, es decir valores 0 y 1, que representarán los colores negro y rojo respectivamente. Nótese en la figura que el valor por defecto de la variable color es 1 dado que como vimos anteriormente siempre insertaremos nodos rojos.

```
class nodoArbolRN(object):
    """Clase nodo árbol rojo-negro."""

    def __init__(self, info):
        """Crea un nodo con la información cargada."""
        self.padre = None
        self.izq = None
        self.der = None
        self.info = info
        self.color = 1
```

Figura 14. Definición de la estructura del nodoArbolRN

Como ya sabemos la *inserción* en un árbol binario se puede hacer de manera recursiva, pero en esta ocasión se presenta una alternativa iterativa. Básicamente determinamos la posición donde se debe insertar –siempre agregamos nodos hojas–, seguidamente se agrega incorpora el nuevo nodo y finalmente se llama a la función reparar *inserción* para chequear si se rompió alguna de las reglas del árbol rojo-negro y restaurar su equilibrio, como se presenta en la figura 15.

```
def insertar_nodo(raiz, dato):
    """Inserta un dato al árbol."""
    ant = None
    act = raiz
    nodo = nodoArbolRN(dato)
    while (act is not None):
        ant = act
        if (nodo.info < act.info):
            act = act.izq
        else:
            act = act.der
        nodo.padre = ant
        if (ant is None):
            raiz = nodo
        elif(nodo.info < ant.info):
            ant.izq = nodo
        else:
            ant.der = nodo
    raiz = reparar_insertar(nodo)
    return raiz
```

Figura 15. Función para insertar nodo en árbol rojo-negro

Por su parte la función que se encarga de reparar el equilibrio del árbol luego de una *inserción*, utiliza dos técnicas coloreo y rotación. Esta función contempla los casos mencionados anteriormente, en algunas ocasiones es necesario aplicar más de un caso para reparar el equilibrio, por eso se comienza a trabajar desde el nodo insertado y se va subiendo a través de su enlace padre –aplicando la técnica necesaria– hasta poder balancearlo o llegar a la raíz, como se detalla en la figura 16 y 17.

```
def reparar_insertar(nodo):
    """Repara el equilibrio del árbol luego de insertar un nodo."""
    aux = None
    while (nodo.padre is not None and nodo.padre.color == 1):
        abuelo = nodo.padre.padre
        if(abuelo is not None and nodo.padre == nodo.padre.izq):
            aux = nodo.padre.padre.der
            if (aux is not None and aux.color == 1): # Case 1
                nodo.padre.color = 0
                aux.color = 0
                nodo.padre.padre.color = 1
                nodo = nodo.padre.padre
            elif (nodo == nodo.padre.der): # Case 2
                nodo = nodo.padre
                rotar_izquierda(nodo)
            else: # Case 3
                nodo.padre.color = 0
                nodo.padre.padre.color = 1
                rotar_derecha(nodo.padre.padre)
```

Figura 16. Función para reparar equilibrio de árbol rojo-negro después de insertar parte 1

```
elif(nodo.padre.padre is not None):
    aux = nodo.padre.padre.izq
    if (aux is not None and aux.color == 1): # Case 1 2
        nodo.padre.color = 0
        aux.color = 0
        nodo.padre.padre.color = 1
        nodo = nodo.padre.padre
    elif (nodo == nodo.padre.izq): # Case 2 2
        nodo = nodo.padre
        rotar_derecha(nodo)
    else: # Case 3
        nodo.padre.color = 0
        nodo.padre.padre.color = 1
        rotar_izquierda(nodo.padre.padre)
    else:
        nodo = nodo.padre

if(nodo.padre is None):
    nodo.color = 0
else:
    while(nodo.padre is not None):
        nodo = nodo.padre
return nodo
```

Figura 17. Función para reparar equilibrio de árbol rojo-negro después de insertar parte 2

Las figuras 18 y 19 presentan la implementación de las funciones rotar a la derecha y a la izquierda, utilizada por las funciones para reparar la *inserción* y *eliminación*. Estas funciones son muy similares a las descriptas en árbol AVL, contemplando la salvedad que los nodos del árbol rojo-negro tiene un enlace extra, que lo conecta con su padre.

```
def rotar_derecha(nodo):
    """Rotar nodod a la derecha."""
    aux = nodo.izq
    nodo.izq = aux.der

    if (aux.der is not None):
        aux.der.padre = nodo
    aux.padre = nodo.padre

    if(nodo.padre is not None):
        if (nodo.padre.der == nodo):
            nodo.padre.der = aux
        else:
            nodo.padre.izq = aux
        aux.der = nodo
        nodo.padre = aux
```

Figura 18. Función rotar a la derecha

```
def rotar_izquierda(nodo):
    """Rotar nodod a la izquierda."""
    aux = nodo.der
    nodo.der = aux.izq

    if (aux.izq is not None):
        aux.izq.padre = nodo
    aux.padre = nodo.padre

    if(nodo.padre is not None):
        if (nodo.padre.izq == nodo):
            nodo.padre.izq = aux
        else:
            nodo.padre.der = aux

        aux.izq = nodo
        nodo.padre = aux
```

Figura 19. Función rotar a la izquierda

Por su parte, la *eliminación* en un árbol rojo-negro inicialmente es igual a la usada para árbol binario, pero en este caso optaremos por una versión iterativa para encontrar el nodo a eliminar. Una vez identificado dicho nodo, si se trata de uno intermedio se busca el nodo que lo reemplazará y se intercambian los valores, sino se procede de la misma manera que para un árbol AVL. Finalmente si el color del nodo eliminado es negro –sabemos que se rompe una regla del árbol– y tenemos que usar la función para reparar el árbol, como se puede ver en las figuras 20 y 21.

```
def eliminar_nodo(raiz, clave):
    """Elimina un elemento del árbol y lo devuelve si lo encuentra."""
    dato = None
    if(raiz is not None):
        aux = raiz
        while(aux is not None and aux.info != clave):
            if(clave < aux.info):
                aux = aux.izq
            else:
                aux = aux.der
        if(aux is not None):
            dato = aux.info
            x = None
            y = None

            if (aux.izq is None or aux.der is None):
                y = aux
            else:
                y = remplazar(aux.izq)
            if(y.izq is not None):
                x = y.izq
            else:
                x = y.der
```

Figura 20. Función para eliminar nodo en árbol rojo-negro parte 1

```
if(y.padre is not None):
    if(y.padre.izq is not None and y.padre.izq == y):
        y.padre.izq = x
    elif(y.padre.der is not None and y.padre.der == y):
        y.padre.der = x

    if(x is None and y.padre is not None and y.color == 0):
        x = nodoArbolRN(0)
        x.color = y.color
    if(x is not None):
        x.padre = y.padre
    if(y != aux):
        aux.info = y.info

    if(y.padre is None and y.izq is None and y.der is None):
        raiz = x
        return raiz, dato

    if(y.color == 0):
        aux = reparar_eliminar(x)
        if(aux is not None):
            raiz = aux
return raiz, dato
```

Figura 21. Función para eliminar nodo en árbol rojo-negro parte 2

Cuando quitamos un nodo negro del árbol se rompe la cuarta regla y tenemos que acudir a las técnicas de coloreo y rotación para reparar el árbol. Para esto la función para reparar debe contemplar todos los casos mencionados anteriormente, al igual que antes, se comienza a trabajar desde el nodo eliminado y vamos subiendo a través del enlace padre del árbol aplicando la técnica que corresponda –dado que se puede necesitar más de una técnica–, hasta poder repararlo como se detalla en la figura 22 y 23.

```
"""Repara el equilibrio del árbol luego de eliminar un nodo."""
while (nodo is not None and nodo.padre is not None and nodo.color == 0):
    if (nodo == nodo.padre.izq or nodo.padre.izq is None):
        w = nodo.padre.der
        if (w.color == 1): # Caso 1
            w.color = 0
            nodo.padre.color = 1
            rotar_izquierda(nodo.padre)
            w = nodo.padre.der
        if((w.izq is None and w.der is None) or (w.izq is not None and
            w.izq.color == 0 and w.der is not None and w.der.color == 0)):
            w.color = 1
            nodo = nodo.padre
        else: # Caso 3 y 4
            if (w.der.color == 0): # Caso 3
                w.izq.color = 0
                w.color = 1
                rotar_derecha(w)
                w = nodo.padre.der
                w.color = nodo.padre.color # Caso 4
                nodo.padre.color = 0
                w.der.color = 0
                rotar_izquierda(nodo.padre)
```

Figura 22. Función para reparar equilibrio de árbol rojo-negro después de eliminar parte 1

```
else:
    w = nodo.padre.izq
    if (w.color == 1): # Caso 1 2
        w.color = 0
        nodo.padre.color = 1
        rotar_derecha(nodo.padre)
        w = nodo.padre.izq
    if((w.izq is None and w.der is None) or (w.der is not None and
        w.der.color == 0 and w.izq is not None and w.izq.color == 0)):
        w.color = 1
        nodo = nodo.padre
    else: # Caso 3 y 4
        if (w.izq is not None and w.izq.color == 0): # Caso 3
            w.der.color = 0
            w.color = 1
            rotar_izquierda(w)
            w = nodo.padre.izq
            w.color = nodo.padre.color # Caso4
            nodo.padre.color = 0
            w.izq.color = 0
            rotar_derecha(nodo.padre)
            break
    nodo.color = 0
if(nodo is not None and nodo.padre is not None):
    while(nodo.padre is not None):
        nodo = nodo.padre
return nodo
if(nodo.padre is None and nodo.izq is None and nodo.der is None):
    return nodo
```

Figura 23. Función para reparar equilibrio de árbol rojo-negro después de eliminar parte 2

Además de las funciones que vimos debemos sumar al TDA árbol rojo-negro las funciones “árbol vacío”, buscar y los barridos inorden, preorder y postorden, las cuales son exactamente las mismas vistas en el capítulo anterior–, dado que ambas estructuras son árboles binarios de búsqueda.

Para probar el TDA árbol rojo-negro utilice el mismo ejemplo del capítulo anterior, de hecho las funciones se llaman de la misma manera por lo cual solo debe cambiar el nombre del TDA del cual importa las funciones necesarias para resolver el problema.

Guía de ejercicios prácticos

Los problemas que se plantean para resolver con el TDA árbol rojo-negro son los mismos de la guía del capítulo anterior, en los que debemos trabajar con árboles AVL.

Simpliquemos las cosas usando montones, ¡montículos para todo!

Ahora que ya contamos con ciertas bases fundamentales para poder entender los montículos, vamos a adentrarnos en esta estructura de datos que no es más que un árbol binario que cumple con ciertas propiedades. Por lo general, cuando hablamos de montículos nos referiremos a montículos binarios, este es una estructura muy simple que puede ser representada utilizando solo un vector. Además, podremos implementar de manera eficiente colas de prioridad –algo que mencionamos en el capítulo VII– y también el método de ordenamiento por montículos (*heapsort*) –que nombramos en el capítulo IV–.

En primer lugar, vamos estudiar qué es un montículo. Se trata de un árbol binario que debe cumplir dos propiedades esenciales para ser considerado como montículo, estas son de estructura y orden: para que se cumpla la propiedad de estructura el árbol binario debe ser completo. Esto implica que todos los niveles del árbol deben estar completos, a excepción del último que debe completar desde la izquierda. La segunda propiedad, la de orden, significa que el árbol debe estar ordenado por niveles, ya sea de manera ascendente o descendente. A partir de esto se pueden clasificar en montículos de orden máximo o de orden mínimo, en donde cada padre de los primeros es mayor que sus hijos y en los segundos los padres son menores que sus hijos.

En resumen, un montículo, cumple las dos condiciones mencionadas anteriormente y por ser binario cada padre solo puede tener como máximo dos hijos. Es importante que aclaremos que el orden de los nodos hermanos no está definido para ningún tipo de montículo. A continuación, en la figura 1 se presenta montículo máximo representado en forma de árbol binario, en el cual podemos ver que los nodos están ordenados por niveles pero no presentan ninguna relación de orden entre los nodos del mismo nivel.

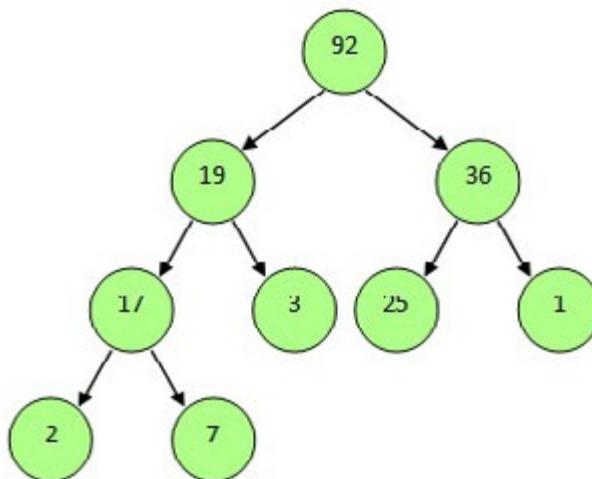


Figura 1. Representación de montículo binario en forma de árbol

Por su parte un árbol binario completo o casi completo es tan regular que se puede representar de manera sencilla utilizando un vector como se observa en la figura 2, y podremos operar sobre este con las siguientes reglas: la raíz del árbol está ubicada siempre en la posición cero del vector, los hijos de un nodo padre ubicado en la posición k se obtienen de la siguiente manera, el hijo izquierdo está en la posición $(2^*k) + 1$ y el hijo derecho está en $(2^*k) + 2$, mientras que la ubicación del padre de un hijo ubicado en la posición k se calcula de la siguiente forma $(k-1) // 2$.

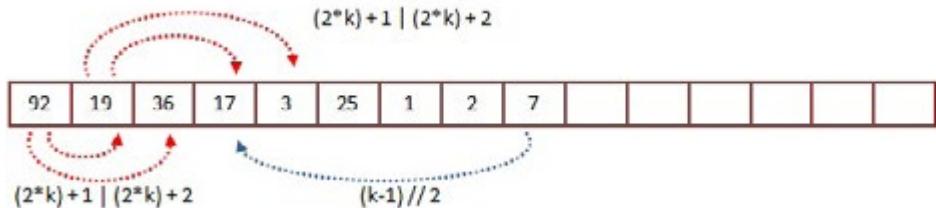


Figura 2. Representación de montículo binario en forma de vector

Resulta sencillo demostrar que un árbol binario está completo de la siguiente manera, si el árbol tiene un nivel n y una altura h la cantidad de nodos del mismo debe ser: nodos = $2^n - 1$. También podemos determinar si un árbol está balanceado o casi lleno cuando la cantidad de nodos que tiene está entre 2^h y $2^{h+1} - 1$, inclusive ambos valores.

Para realizar la definición del TDA montículo utilizaremos solamente un vector de n elementos que denominaremos “*heap*”, este será útil para representar un árbol binario como ya vimos anteriormente. También dispondremos de un conjunto de funciones que definen su funcionamiento:

1. **crear_montículo(tamaño).** Crea y devuelve un montículo vacío con la cantidad de elementos, determinado por el tamaño ingresado;
2. **agregar(montículo, dato).** Inserta un elemento al final del montículo y luego lo flota hasta que dicho elemento cumpla la propiedad de orden;
3. **quitar(montículo, dato).** Quita y devuelve el máximo o mínimo elemento del montículo –dependiendo de su tipo–, es decir el dato que se ubica en la raíz del árbol , y en su lugar se coloca el último elemento del montículo y luego lo hunde hasta que cumpla la propiedad de orden;
4. **flotar(montículo, índice).** Flota el dato almacenado en el montículo desde el índice indicado hasta que cumpla la propiedad de orden;
5. **hundir(montículo, índice).** Hunde el dato almacenado en el montículo desde el índice indicado hasta que cumpla la propiedad de orden;
6. **montículo_vacio(montículo).** Devuelve verdadero (*true*) si el montículo no contiene elementos;
7. **montículo_lleno(montículo).** Devuelve verdadero (*true*) si el montículo no puede almacenar más elementos;
8. **tamaño(montículo).** Devuelve la cantidad de elementos del montículo;
9. **monticulizar(montículo).** Convierte un vector de elementos en un montículo;

Cabe destacar que tanto las operaciones para insertar y eliminar elementos en un montículo son del orden de $O(\log n)$ lo cual es muy eficiente, dado que es una representación de un árbol binario y mantiene dicha propiedad.

Centrémonos en analizar de qué manera hacemos que se mantenga la propiedad de orden cuando agregamos o quitamos elementos, estas se resuelven de las siguientes maneras: cuando se insertar un nuevo elemento se agrega al final y luego se lo debe flotar hasta que cumpla la condición de orden –es decir, mientras sea mayor o menor que su padre dependiendo si es un montículo máximo o mínimo respectivamente–, por ejemplo suponga que en el montículo de la figura 1 se quiere agregar el valor 23, el mismo flotara dos veces hasta quedar en la posición correcta del montículo, como se detalla en la figura 3.

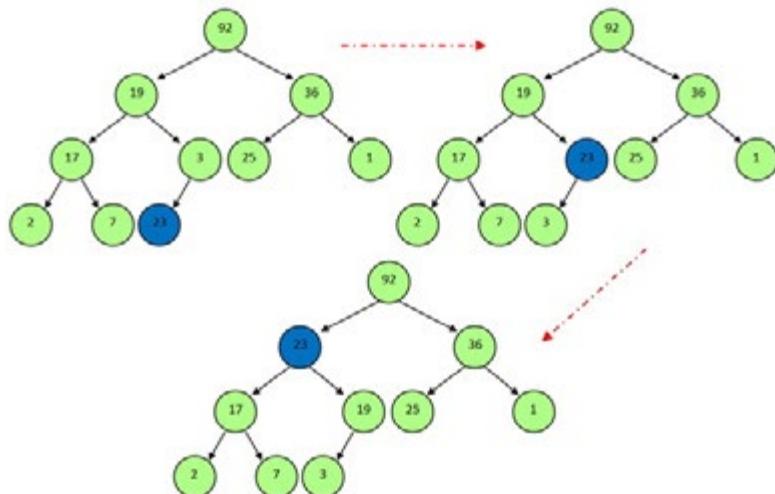


Figura 3. Secuencia de inserción de un elemento al montículo

En cambio, cuando se elimina un elemento, siempre se quita el elemento que está en la cima del mismo, para esto se procede de la siguiente manera: se lo intercambia con el último elemento del montículo, luego se hunde el elemento colocado en la cima hasta que cumpla la propiedad de orden –al igual que en la inserción dependiendo si es un montículo máximo o mínimo–. Continuando con el ejemplo, si se quiere quitar un elemento se quitará el 92, el 3 pasará a ocupar a cima del montículo y se hundirá 2 veces hasta quedar en su correspondiente posición, como se puede ver en la figura 4.

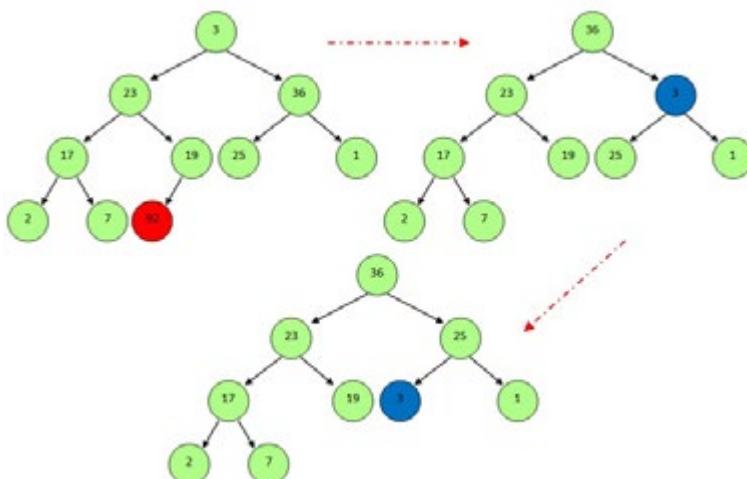


Figura 4. Secuencia de eliminación de un elemento del montículo

Continuemos con la implementación del TDA montículo, para esto utilizaremos una clase que dispondrá de un vector para representarlo y el tamaño para indicar la cantidad de elementos, esto lo podemos observar en la figura 5, luego en las figuras 6, 7 y 8 se presenta la implementación de las funciones mencionadas previamente que nos permitirán administrar el funcionamiento del montículo.

```
class Heap(object):
    """Crea un monticulo."""

    def __init__(self, tamano):
        """Crea el vector vacio para el monticulo."""
        self.tamano = 0
        self.vector = [None] * tamano
```

Figura 5. Definición de la estructura del TDA montículo

```
def agregar(heap, dato):
    """Agrega un dato en el monticulo."""
    heap.vector[heap.tamano] = dato
    flotar(heap, heap.tamano)
    heap.tamano += 1

def quitar(heap):
    """Quita el elemento en la cima del monticulo."""
    intercambio(heap.vector, 0, heap.tamano-1)
    dato = heap.vector[heap.tamano-1]
    heap.tamano -= 1
    hundir(heap, 0)
    return dato
```

Figura 6. Interfaz o eventos del TDA *heap* parte 1

```
def cantidad_elementos(heap):
    """Devuelve la cantidad de elementos en el monticulo."""
    return heap.tamano
```

```
def heap_vacio(heap):
    """Devuelve true si el monticulo esta vacio."""
    return heap.tamano == 0
```

```
def heap_lleno(heap):
    """Devuelve true si el monticulo esta lleno."""
    return heap.tamano == len(heap.vector)
```

Figura 7. Interfaz o eventos del TDA *heap* parte 2

```

def flotar(heap, indice):
    """Flota el elemento en la posición índice."""
    while(indice > 0 and heap.vector[indice] > heap.vector[(indice - 1) // 2]):
        padre = (indice - 1) // 2
        intercambio(heap.vector, indice, padre)
        indice = padre

def hundir(heap, indice):
    """Hunde el elemento en la posición índice."""
    hijo_izq = (indice * 2) + 1
    control = True
    while(control and hijo_izq < heap.tamanio):
        hijo_der = hijo_izq + 1
        aux = hijo_izq
        if(hijo_der < heap.tamanio):
            if heap.vector[hijo_der] > heap.vector[hijo_izq]:
                aux = hijo_der

        if (heap.vector[indice] < heap.vector[aux]):
            intercambio(heap.vector, indice, aux)
            indice = aux
            hijo_izq = (indice * 2) + 1
        else:
            control = False

```

Figura 8. Interfaz o eventos del TDA *heap* parte 3

Cada vez que un elemento se inserta al montículo se llama a la función flotar, la cual mientras el índice donde fue ingresado sea mayor que cero y dicho elemento sea mayor que su padre –que se encuentra ubicado en la posición $(índice - 1) // 2$ –, entonces se intercambiarán dichos valores y actualizará la posición del índice por la del padre y se continúa repitiendo esto hasta que se cumpla la condición de orden. En el peor de los casos el elemento flotará hasta la cima del mismo. Por otro lado, cuando se quita un elemento se llama a la función hundir, esta primero determina si tiene un hijo –el izquierdo–, de ser así significa que se debe comparar. Si es necesario hundir el parente, antes de hacer esto determina si tiene hijo derecho. Luego, se determina cuál de los hijos es el mayor, en el caso de que ambos existan, si el hijo mayor es mayor que el parente se procede a intercambiarlo con su parente, a continuación se actualiza la posición del índice del parente por la del hijo que se intercambió y se repite este procedimiento hasta que el elemento cumpla la condición de orden, es decir que sea mayor que sus dos hijos o quede ubicado en el fondo del montículo.

Pero ¿Cómo hacemos para construir un montículo a partir de un vector que ya está cargado? Para crear un montículo a partir de un árbol binario almacenado en un vector –que podría no estar ordenado por niveles– o simplemente a partir de un vector de elementos, recurriremos a un procedimiento denominado “monticulizar”, para lo cual se aplica la función flotar desde el segundo elemento del vector hasta el último elemento, como se observa a continuación en la figura 9, esta operación es del orden de $O(n \log n)$.

```

def monticulizar(heap):
    """Transforma un vector en un mantículo."""
    for i in range(len(heap.vector)):
        flotar(heap, i)

```

Figura 9. Función para monticulizar un vector

Ahora que ya conocemos cómo funcionan los montículos podremos usarlos para implementar colas, en particular para implementar colas de prioridad las cuales dejamos pendiente en el capítulo VII, las colas de prioridad son muy útiles para que los sistemas operativos puedan administrar los distintos recursos del sistema y para otros ejemplos que ya vimos en dicho capítulo. Al usar un montículo para aplicar cola de prioridad –la propiedad de orden será la prioridad asignada al dato–, quedando en la raíz del árbol el nodo de mayor o menor prioridad dependiendo del tipo de montículo (máximo o mínimo). Como ya sabemos, en un montículo los elementos se agregan al final y luego se lo flota hasta cumplir la propiedad de orden. Esto permite respetar la naturaleza de una cola –permitiendo que los elementos de mayor prioridad se adelanten– y tener un coste de inserción del orden de $O(\log n)$. A continuación, se presentan las funciones que necesitaremos definir para poder implementar las colas de prioridad:

1. arribo(montículo, dato, prioridad). Agrega el elemento al final del montículo y luego lo flota según su criterio de prioridad, para utilizarlo como cola de prioridad;
2. atención(montículo). Elimina y devuelve el elemento almacenado en la cima del montículo, utilizado como cola de prioridad;
3. cambiar_prioridad(montículo, índice, prioridad). Cambia la prioridad de un elemento del montículo y lo flota o hunde según corresponda.

Sigamos ahora observando la figura 10 en la cual se presenta la implementación de las funciones anteriores, para poder utilizar el montículo como si fuera una cola. La función arribo solamente llamará a la función agregar definida en el TDA montículo, y agrupará los campos prioridad y dato –para no tener que alterar dicha función–, mientras que la función atención solo llamará a la función quitar.

```

def arribo(heap, dato, prioridad):
    """Arriba el dato a la cola utilizando prioridad."""
    agregar(heap, [prioridad, dato])

def atencion(heap):
    """Antiendo el elemento en el frente de la cola y lo devuelve."""
    return quitar(heap)[1]

def cambiar_prioridad(heap, indice, prioridad):
    """Cambia la prioridad de un elemento y lo acomoda en el monticulo."""
    prioridad_anterior = heap[indice][0]
    heap[indice][0] = prioridad
    if(prioridad > prioridad_anterior):
        flotar(heap, indice)
    elif(prioridad < prioridad_anterior):
        hundir(heap, indice)

```

Figura 10. Cola de prioridad con montículo

Pasemos a un ejemplo de uso del TDA montículo (como cola de prioridad) para la resolución de un problema en la figura II. En este caso, se deben generar números aleatorios (del 0 al 100) junto con prioridades también generadas aleatoriamente (de 1 a 3) hasta completar el montículo y luego realizar la atención de todos los elementos de la cola de acuerdo a su orden de prioridad.

```

from tda_heap import Heap, heap_lleno, heap_vacio, agregar, atencion
from random import randint

heap = Heap(10)

while(not heap_lleno(heap)):
    num = randint(0, 500)
    prioridad = randint(1, 3)
    agregar(heap, [prioridad, num])
    print(heap.vector)
while(not heap_vacio(heap)):
    dato = atencion(heap)
    print(dato)

```

Figura II. Ejemplo de uso del TDA Cola

Hagamos una breve descripción de las acciones realizadas para la resolución del ejercicio anterior, para comprender el uso del TDA montículo como cola de prioridad utilizando los eventos definidos previamente.

Lo primero que hacemos es importar del TDA montículo las funciones que se utilizaremos, incluyendo el módulo *random* para generar números aleatorios, luego hay que crear la variable de tipo *heap* para después generar los números y su prioridad de manera aleatoria y agregarlos a la cola. Finalmente, realizamos la atención de todos los elementos de la cola mostrando sus valores.

Flotando y hundiendo los elementos mediante ordenamiento por montículo

Es un algoritmo cuyo funcionamiento es del tipo selección, creado por John William Joseph Williams en 1964¹. Funciona siguiendo los principios de montículo, es decir que los elementos a ordenar siguen las propiedades de dicha estructura, luego para ordenar los elementos toma el mayor o menor (dependiendo si es un montículo de tipo máximo o mínimo), es decir el que está situado en la primera posición del vector y lo intercambia con el último elemento del vector. Luego reubica el elemento colocado en la primera posición hundiéndolo (para restablecer la propiedad de orden del montículo) y marca el último elemento como ordenado decrementando el tamaño del vector. Y se repite sucesivamente este proceso con el resto de los elementos del vector hasta que el tamaño del vector desordenado es uno. A continuación, en la figura 12 se puede observar el funcionamiento del algoritmo para ordenar un vector, nótese que inicialmente se debe llamar a la función monticulizar para transformar el vector en un montículo.

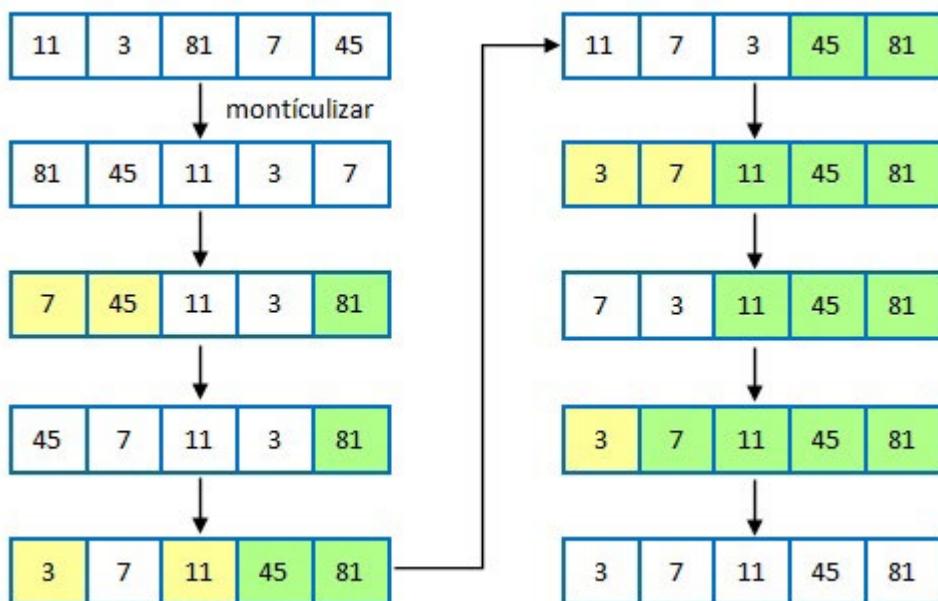


Figura 12. Ejemplo de funcionamiento del algoritmo *heapsort*

¹ Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", Communications of the ACM, 7 (6): 347–348, doi:10.1145/512274.512284

Pasemos ahora a observar la implementación del algoritmo de ordenamiento por montículos en la figura 13. Nótese que es uno de los algoritmos de ordenamientos más sencillos que hemos visto, dado que solo son cuatro líneas de código, esto es posible por la simplicidad del uso de los montículos.

```
def heapsort(heap):
    """Método de ordenamiento heapsort."""
    aux = heap.tamanio
    for i in range(heap.tamanio):
        quitar(heap)
    heap.tamanio = aux
```

Figura 13. Método de ordenamiento *heapsort*

Finalmente realicemos la comparación del algoritmo de ordenamiento por montículos respecto a los otros algoritmos de ordenamiento, utilizando los mismos aspectos que mencionamos en el capítulo IV.

Complejidad temporal	$O(n \log n)$
Complejidad espacial	En el lugar
Estabilidad	Inestable
Interno /Externo	Interno
Recursivo/No recursivo	No recursivo
Ordenamiento por comparación	Comparativo

Guía de ejercicios prácticos

A continuación se plantean una serie de problemas, los cuales deberán resolver utilizando el TDA montículo.

1. Resolver el ejercicio 15 de la guía de ejercicios del capítulo VII, utilizando cola de prioridad.
2. Desarrollar un algoritmo que permita implementar cola de prioridad para administrar los turnos de atención de la guardia de un hospital, con el siguiente criterio: consultas (1), emergencias (2) y urgencias (3). Resuelva las siguientes actividades:
 - a. cargar 10 pacientes con diferentes prioridades;
 - b. realizar la atención de los pacientes, luego de atender cada paciente se pueden agregar nuevos a la cola;
 - c. antes de realizar todas las atenciones, cambiar la prioridad del turno que está en la posición i -ésima.
3. El general Hux es la persona encargada de administrar todas las operaciones de la base Starkiller, para lo cual nos solicita desarrollar un algoritmo que permita controlar las actividades que se realizan, contemplando lo siguiente:
 - a. debe contemplar distintas prioridades para el control de operaciones de acuerdo al siguiente criterio: pedidos de líder supremo Snoke y de Kylo Ren nivel tres, de capitán Phasma nivel dos y el resto de las operaciones nivel a cargo de los generales de la base de nivel uno;
 - b. de cada actividad se conoce quien es el encargado, una descripción, la hora y opcionalmente la cantidad de Stormtroopers requeridos;
 - c. utilizar una cola de prioridad para administrar las distintas operaciones, debe cargar al menos ocho: dos de nivel tres, cuatro de nivel dos y cuatro de nivel uno;
 - d. opcionalmente se podrán agregar operaciones luego de atender una;
 - e. realizar la atención de las operaciones de la cola;
 - f. luego de atender la quinta operación, agregar una operación solicitada por capitán Phasma para revisión de intrusos en el hangar B7 que requiere 25 Stormtroopers;
 - g. luego de atender la sexta operación, agregar una operación solicitada por el líder supremo Snoke para destruir el planeta Takodana.
4. Resolver el ejercicio 4 de la guía de ejercicios del capítulo IV utilizando el método de ordenamiento *heapsort* y comparar con los otros algoritmos probados en dicho capítulo.

5. El gran almirante Thrawn es el estratega del imperio galáctico para combatir contra los rebeldes, el mismo normalmente se encuentra desbordado de pedidos de sugerencia de cómo actuar por los distintos generales, para lo cual nos solicita desarrollar un algoritmo que le permita atender los pedidos de ayuda de acuerdo a la prioridad de los mismo en base a los siguientes requerimientos:
 - a. se deben contemplar tres niveles de prioridad para la cola;
 - b. cada pedido de sugerencia cuenta con la siguiente información: nombre del general solicitante, planeta en el que se encuentra o el más próximo y descripción del pedido;
 - c. aquellos pedidos que provengan del “Gran Inquisidor”, el planeta de Lothal o la descripción mencione a “Hera Syndulla” tendrán la mayor prioridad;
 - d. si el pedido es del “Agente Kallus” o la descripción menciona “Destructor Estelar” o vehículos “AT-AT” tendrán prioridad media;
 - e. el resto de los pedidos serán de prioridad baja;
 - f. realizar la atención de la cola almacenando los pedidos de mayor prioridad en una pila llamada “bitácora” para revisión y seguimiento;
 - g. luego de cada atención se podrá agregar un pedido a la cola.
6. El comandante de la estrella de la muerte el gran Moff Tarkin debe administrar las asignaciones de vehículos y Stormtroopers a las distintas misiones que parten desde la estrella de la muerte, para facilitar esta tarea nos encomienda desarrollar las funciones necesarias para gestionar esto mediante prioridades de la siguiente manera:
 - a. de cada misión se conoce su tipo (exploración, contención o ataque), planeta destino y general que la solicitó;
 - b. si la misión fue pedida por Palpatine o Darth Vader estas tendrán alta prioridad, sino su prioridad será baja;
 - c. si la misión es de prioridad alta los recursos se asignarán manualmente independientemente de su tipo;
 - d. si la misión es de baja prioridad se asignarán los recursos de la siguiente manera dependiendo de su tipo:
 - I. exploración: 15 Scout Troopers y 2 speeder bike,
 - II. contención: 30 Stormtroopers y tres vehículos aleatorios (AT-AT, AT-RT, AT-TE, AT-DP, AT-ST) pueden ser repetidos,
 - III. ataque: 50 Stormtroopers y siete vehículos aleatorios (a los anteriores se le suman AT-M6, AT-MP, AT-DT),

- e. realizar la atención de todas las misiones y mostrar los recursos asignados a cada una, permitiendo agregar nuevos pedidos de misiones durante la atención;
- f. indicar la cantidad total de recursos asignados a las misiones.

Conectando puntos. Cómo formar una red conocida como grafo

Es momento de enfocarnos en una nueva estructura ramificada, formada por una red de nodos conectados entre sí, que se conoce como grafo. Anteriormente, estudiamos la estructura de árbol y sus diferentes tipos, estos también son casos particulares de grafos pero con ciertas restricciones. Los grafos son útiles para abordar problemas de distintas disciplinas como ciencias de la computación, matemáticas, ingeniería y otras tantas, en las que es necesario representar algún tipo de relación entre datos se almacenan en dicha estructura. A diferencia de un árbol, el grafo no tiene un nodo raíz, además desde un nodo se puede acceder a muchos otros y un nodo puede ser accedido desde distintos nodos. A continuación se analizará en detalle la estructura de datos grafos y sus tipos, las distintas alternativas de representación y también las diferentes variantes de exploración y búsqueda. Además se hará foco en algunos algoritmos especiales para resolver los problemas clásicos de grafo, como determinar el camino más corto o encontrar el árbol de expansión mínimo, entre otras cuestiones.

La teoría de grafos tuvo su origen a partir de un trabajo de Leonhard Euler matemático suizo en 1736¹, como resultado al histórico problema matemático de los siete puentes Königsberg. La cual actualmente es la ciudad de Kaliningrado en Rusia, esta conocida por los siete puentes que conecta los márgenes del río Pregel con dos de sus islas, como se puede ver en la figura 1. El problema en cuestión era idear un paseo por la ciudad que cruzara cada uno de esos puentes una sola vez y volver al punto de inicio. De manera general el problema consistía en determinar si: *¿es posible, partiendo de un lugar arbitrario, regresar a dicho lugar de partida cruzando cada ruta una sola vez?*

Euler demostró que el problema no tenía solución representando el problema de los puentes de Königsberg con un grafo –dando origen a la teoría de grafo–. La mayor dificultad que tuvo en el proceso fue el desarrollo de una técnica de análisis adecuado y las pruebas posteriores que establecieron dicha afirmación con rigor matemático. En efecto Euler logra resolver un problema más general: *¿qué condiciones debe cumplir un grafo para que se pueda volver al punto de partida pasando por todos los nodos?* Para esto todos los vértices deben ser de grado par y esto se denomina como ciclo euleriano.

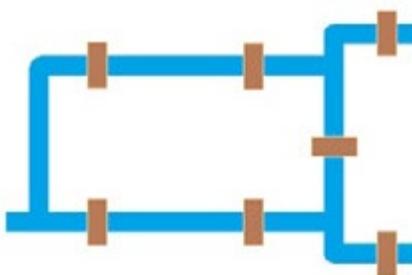


Figura 1. Esquema de puentes de Königsberg

¹ Euler, Leonhard (1736). "Solutio problematis ad geometriam situs pertinentis". Comment. Acad. Sci. U. Petrop 8, 128–40.

Básicamente existen dos tipos de grafos, dirigidos y no dirigidos, comencemos a estudiar el primero de estos también conocido como digrafos. Este es un conjunto $G = (V, A)$ que está compuesto por un conjunto de vértices o nodos $V (v_1, v_2, v_3, \dots, v_n)$ y un conjunto de aristas o arcos $A (a_1, a_2, a_3, \dots, a_n)$. Una arista está compuesta por un par ordenado (a, b) , se decir que a es el vértice origen y b el vértice destino, es decir que el vértice b es accedido desde a y se lo representa con una flecha, como se observa en la figura 2. Nótese que la arista tiene una dirección, por eso se denominan grafos. Esta arista que representa la relación entre dos vértices también se denomina adyacencia, se dice entonces que el vértice b es adyacente de a .



Figura 2. Arista que une el vértice a con el b

Mediante grafos podemos representar y modelar múltiples situaciones de nuestra vida cotidiana por ejemplo: ciudades y rutas que las conectan, aeropuertos y sus correspondientes vuelos que los unen, puertos y las rutas marítimas, circuitos eléctricos, una red neuronal, puntos de conexión de una red, relaciones entre personas, diagrama de actividades, seguidores o amigos de un usuario en una red social y un largo etcétera. Tomando el ejemplo de las redes sociales, los vértices serían personas y las aristas las relaciones de amistad –o si sigue a una determinada persona–. A continuación en la figura 3 se presenta un grafo dirigido que tomaremos como ejemplo para explicar la terminología básica referida a grafos. Este nos ayudará a comprender los conceptos que se desarrollan a lo largo del capítulo.

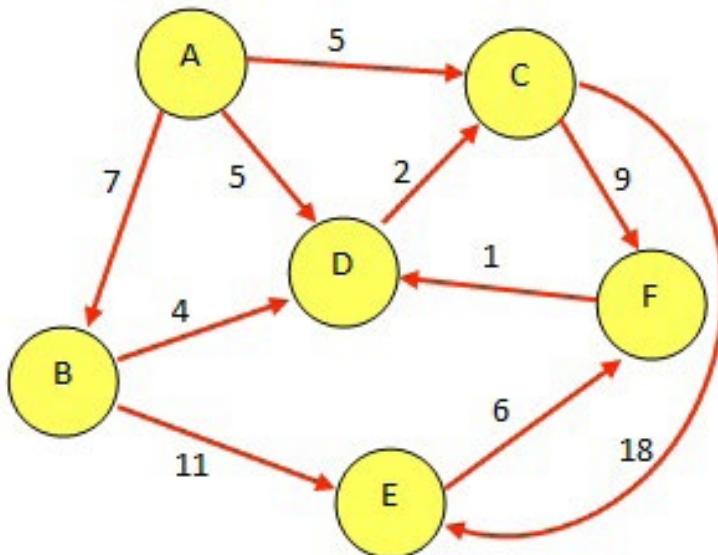


Figura 3. Estructura de un grafo dirigido

Camino: en un grafo dirigido en un conjunto de vértices (v_1, v_2, v_3) tal que existen los arcos del camino (v_1-v_2, v_2-v_3). Por ejemplo el camino de vértices A, C, F, D , está compuesto de los arcos AC, CF, FD .

Longitud de un camino: es la cantidad de aristas del camino o la cantidad de vértices del camino menos uno. Siguiendo el ejemplo del camino anterior, la longitud es tres.

Etiqueta de una arista: es un valor que está asociado a la relación entre dos vértices. Este suele ser un valor numérico que representa tiempo, distancia, cantidades que indican el valor de ir desde el vértice origen al destino, también se lo denomina peso. Por ejemplo la etiqueta de la arista CF es nueve.

Conexo: un grafo dirigido es conexo si desde cualquier vértice se puede acceder a cualquier otro, ya sea de manera directa o indirecta –es decir pasando por nodos intermedios-. Para el caso de la figura no es conexo dado que, por ejemplo, desde el vértice C no puedo acceder al vértice B de ninguna manera.

Acíclico: un grafo dirigido es acíclico si no tiene ciclos, esto implica que para cada vértice del grafo no exista ningún camino que empiece en un vértice y termine en el mismo –un vértice podría tener un arco a sí mismo y formar un ciclo-. Continuando con el ejemplo de la figura, el grafo no es acíclico dado que existe un ciclo entre $C-F-D$.

Camino hamiltoniano: es un camino que consiste en visitar todos los vértices de grafo pasando solo una vez por cada uno, si además el primer vértices es adyacente de el último el camino también es un ciclo hamiltoniano. En el ejemplo el camino $A-B-E-F-D-C$ es un camino hamiltoniano.

Camino euleriano: es un camino que consiste en pasar por cada arista del grafo solo una vez –no importa que visite los vértices más de una vez-, asimismo si el vértice de partida es el vértice de llegada el camino es un ciclo euleriano (este último es el famoso problema de los puentes de Königsberg).

Por lo cual, podemos definir un grafo de la siguiente manera basándonos en su estructura y su mecánica de funcionamiento: es una estructura ramificada de datos llamados vértices, relacionados entre sí por medio de arista que pueden tener una etiqueta que refleja alguna característica de dicha relación.

Para comenzar a trabajar con grafos primero debemos entender de qué manera podemos representar todos los conceptos abordados. Como ya sabemos un grafo está formado por un conjunto de vértices y un conjunto de aristas que enlazan los vértices. Cuando el número de aristas es elevado respecto a la cantidad de vértices se dice que el grafo es denso, en el caso contrario es disperso. La primera alternativa que tenemos es una forma muy intuitiva de representar un grafo, para lo cual utilizamos una matriz cuadrada donde la etiqueta de los vértices serán las filas y columnas de la misma. Luego, como es un grafo dirigido, cargamos las aristas de la siguiente manera: el origen será la fila y el destino la columna. Y en la intersección colocaremos el peso de la etiqueta de la arista –si es sin peso colocaremos un 1 para indicar que dichos vértices están conectados–, siguiendo con el ejemplo tomamos el vértice A (fila) y luego completamos en la intersección de las columnas de los vértices adyacentes: en B se coloca un 7, en C ponemos un 5, en D completamos con un 5, después en el resto de las columnas se pone cero y se repite el mismo procedimiento para las demás filas como se ve en la figura 4. Si fuera un grafo no dirigido sería una matriz simétrica, es decir tanto en la intersección fila-columna y columna-fila deberá ir el peso de la etiqueta.

	A	B	C	D	E	F
A	0	7	5	5	0	0
B	0	0	0	4	11	0
C	0	0	0	0	18	9
D	0	0	2	0	0	0
E	0	0	0	0	0	6
F	0	0	0	1	0	0

Figura 4. Matriz de adyacencia de un grafo dirigido

La principal desventaja de utilizar una matriz de adyacencia es que si el grafo no es denso se desperdicia mucho espacio de la matriz dado que solo se almacenan ceros, como se observa en la figura anterior, y además las etiquetas de los vértices deben estar enumeradas consecutivamente con números o caracteres que puedan ser mapeados de manera directa. Caso contrario necesitaremos una función que se encargue de realizar la transformación de la etiqueta del vértice a una posición fila-columna cuyo datos estarán almacenados en un vector que altera la eficiencia de acceso. Pero no todo es malo, la ventaja de esta representación es que rápidamente podemos localizar si existe una arista, por ejemplo si queremos saber si existe una conexión entre el vértice D y F, accedemos a la matriz en la posición correspondiente a cada vértice (matriz[3] [5]) y nos preguntamos si es distinto de cero, con un coste de acceso de orden $O(1)$, lo cual es muy eficiente.

Otra alternativa de representación es utilizar un vector de listas de adyacencia, en el vector tendremos cada uno de los vértices y una lista de aristas, las cuales almacenarán la etiqueta del vértice destino y el peso de ir del vértice origen al destino. Como se observa en la figura 5 solo se representan las conexiones, por esto ocupa menos espacio que la representación anterior, lo cual es una buena estrategia a utilizar cuando el grafo no es denso, dado que el tamaño es proporcional al número de aristas del grafo. La desventaja de esta representación es que si queremos determinar si hay una conexión entre dos vértices debemos recorrer la lista de adyacencia del vértice origen para determinar si está el destino lo cual es una operación de costo lineal del orden de $O(n)$; además las etiquetas de los vértices deben ser enumeradas consecutivamente o se requerirá una función que se encargue de transformar las etiquetas como en el caso anterior.

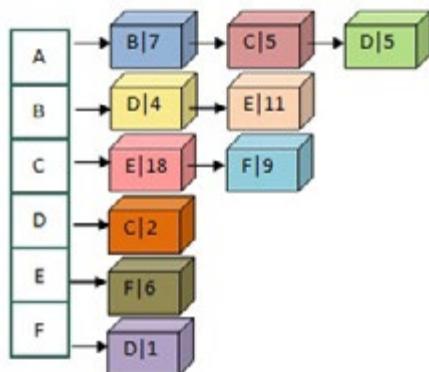


Figura 5. Vector de listas de adyacencia de un grafo dirigido

La última alternativa que podemos utilizar es completamente proporcional al número de vértices y aristas, como se puede observar a continuación en la figura 6, usando lista de listas de adyacencia. La ventaja de este modelo respecto de los anteriores es que es completamente dinámico, por lo cual se puede cambiar el número de vértices sin necesidad de volver a crear la matriz o el vector requerido en los casos anteriores. Al igual que en el caso anterior su desventaja es que el costo de acceso es lineal tanto cuando buscamos un vértice o una arista. Esta implementación es recomendada cuando hay que representar un grafo disperso y volátil, es decir que se agreguen y eliminen elementos con mucha frecuencia.

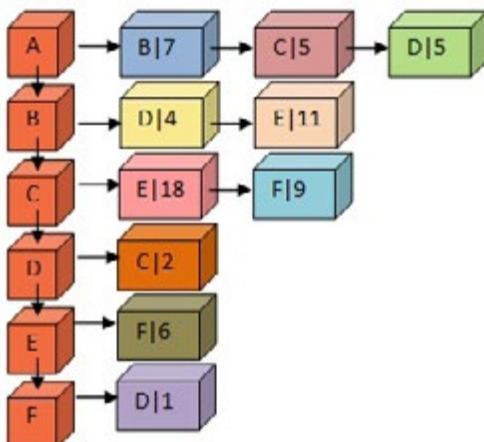


Figura 6. Lista de listas de adyacencia de un grafo dirigido

En resumen el criterio para elegir el tipo de representación a utilizar no será siempre el mismo y dependerá de los siguientes parámetros: del tipo de problema a trata, su densidad y volatilidad.

Pongamos la atención por un momento en como recorrer un grafo, algunas preguntas que seguramente se te vendrán a la mente: *¿Cómo se realiza un barrido de esta red de nodo? ¿Por dónde se debe comenzar?* Básicamente existen dos tipos de barridos que se pueden aplicar a grafos ya sean dirigido o no dirigido, para esto primero debemos marcar todos los vértices como no visitados y cuando pasamos por cada nodo lo marcamos como visitado, para evitar pasar dos veces por el mismo nodo –y entrar en un ciclo infinito– dado que hay más de un camino al mismo nodo:

Exploración en profundidad: se comienza por un vértice arbitrariamente, se trata dicho vértice y se lo marca como visitado, luego se trata recursivamente todos los vértices adyacentes no visitados, si al finalizar quedan vértices sin visitar se toma el siguiente vértice sin tratar y se repite el procedimiento. Continuando con el grafo de ejemplo, si aplicamos este barrido se obtendrá como salida *A, B, D, C, E, F*.

Exploración en amplitud: se crea una cola a la cual se agrega un vértice arbitrariamente y se lo marca como visitado, luego mientras la cola no esté vacía se atiende el primer elemento de la cola y se trata el vértice. A continuación se agregan a la cola y se marcan como visitados todos los vértices adyacentes que aún no hayan sido visitados; cuando la cola queda vacía si aún quedan vértices sin visitar, se toma el siguiente vértice sin tratar y repite el procedimiento anterior. La salida si aplicamos este barrido al grafo de ejemplo es *A, B, C, D, E, F*.

Comencemos a pensar en el diseño del TDA grafo, este tendrá muchas similitudes con el TDA lista dado que optaremos por una representación de lista de listas de adyacencia. En el TDA tendremos la clase *grafo* compuesta básicamente de dos elementos denominados *inicio* y *dirigido*, el primero apuntará a un *nodoVertice* y segundo es un campo booleano que indicará si es un grafo dirigido o no. El *nodoVertice*, por su parte, contará con los siguientes campos: *información* que almacena el dato del vértice, *siguiente* que almacena la dirección del siguiente vértice de la lista, *visitado* es un campo booleano utilizado para los barridos y *adyacentes* que es de tipo arista que representa la lista de arcos que sale de dicho vértice es decir sus adyacentes. La clase *arista* es una lista de *nodoArista* el cual consta de los siguientes campos *información* que almacena el peso de la arista, *destino* con el valor de la etiqueta del vértice destino y *siguiente* que guardará la dirección de la siguiente arista adyacente. Opcionalmente puede agregarse el campo tamaño tanto a la clase *grafo* y *arista* para facilitar el cálculo de la cantidad de elementos en cada una sin necesidad de utilizar funciones extras. Además tendremos un conjunto de operaciones que dispondremos para administrar el grafo, las cuales son enumeradas a continuación:

1. *insertar_vértice(grafo, dato)*. Agrega el elemento como un vértice al grafo;
2. *insertar_arista(grafo, dato, vértice origen, vértice destino)*. Agrega el elemento como una arista desde el vértice destino al vértice origen, si el grafo es dirigido y si es no dirigido también lo inserta desde el vértice destino al origen;
3. *agregar_arista(vértice origen, dato, vértice destino)*. Agrega una arista a la lista de aristas del vértice origen al vértice destino;
4. *eliminar_vértice(grafo, clave)*. Elimina y devuelve del grafo si encuentra un vértice que coincida con la clave dada –el primero que encuentre– y además recorre el resto de los vértices eliminando las aristas cuyo destino sea el vértice eliminado, si devuelve *None* significa que no se encontró la clave en el grafo, y por ende no se elimina ningún elemento;
5. *eliminar_arista(vértice, destino)*. Elimina y devuelve del vértice si encuentra una arista que coincida con el destino dado –el primero que encuentre–, si devuelve *None* significa que no se encontró la arista destino en el vértice, y por ende no se elimina ningún elemento;
6. *buscar_vértice(grafo, clave)*. Devuelve un puntero que apunta al vértice que contiene un elemento que coincide con la clave –el primero que encuentra–, si devuelve *None* significa que no se encontró la clave en el grafo;
7. *buscar_arista(grafo, vértice origen, vértice destino)*. Devuelve un puntero que apunta a la arista que contiene el elemento que coincide con el destino –el primero que encuentra–, en la lista de aristas del vértice origen, si devuelve *None* significa que no se encontró el destino desde el vértice origen;
8. *grafo_vacio(grafo)*. Devuelve verdadero (*true*) si el grafo no contiene elementos;
9. *tamaño(grafo)*. Devuelve la cantidad de vértices en la grafo;
10. *barrido_vértices(grafo)*. Realiza un barrido de los vértices ordenados;

- ii. es_adyacente(vértice, destino). Devuelve verdadero (*true*) si el destino es un nodo adyacente al vértice;
- 12. adyacentes(vértice). Realiza un barrido de los nodos adyacentes al vértice;
- 13. marcar_no_visitado(grafo). Marca todos los nodos vértices como no visitados poniendo el campo visitado con valor falso (*false*);
- 14. existe_paso(grafo, vértice origen, vértice destino). Devuelve verdadero (*true*) si es posible ir desde el vértice origen hasta el vértice destino, caso contrario retornará falso (*false*);
- 15. barrido_profundidad(grafo, vértice inicio). Realiza un barrido en profundidad del grafo a partir del vértice de inicio;
- 16. barrido_amplitud(grafo, vértice inicio). Realiza un barrido en amplitud del grafo a partir del vértice de inicio;
- 17. dijkstra(grafo, vértice origen, vértice destino). Devuelve el camino más corto desde el vértice origen al vértice destino;
- 18. prim(grafo, vértice inicio). Devuelve el árbol de expansión mínimo del grafo a partir del vértice de inicio;
- 19. kruskal(grafo, vértice inicio). Devuelve el árbol de expansión mínimo del grafo a partir del vértice de inicio.

Empecemos con la implementación del TDA para lo cual debemos crear las clases nodoVertice y nodoArista detalladas anteriormente. Además se deben definir los tipos de dato arista y grafo, ambos con los campos inicio y tamaño con valores *None* y cero respectivamente –como lo hicimos en el TDA lista–, las definiciones de todos estos tipos de datos se presentan en las figuras 7 y 8.

```

class nodoArista(object):
    """Clase nodo vértice."""

    def __init__(self, info, destino):
        """Crea un nodo arista con la información cargada."""
        self.info = info
        self.destino = destino
        self.sig = None


class nodoVertice(object):
    """Clase nodo vértice."""

    def __init__(self, info):
        """Crea un nodo vértice con la información cargada."""
        self.info = info
        self.sig = None
        self.visitado = False
        self.adyacentes = Arista()

```

Figura 7. Definición de la estructura del TDA grafo

```

class Grafo(object):
    """Clase grafo implementación lista de listas de adyacencia."""

    def __init__(self, dirigido=True):
        """Crea un grafo vacío."""
        self.inicio = None
        self.dirigido = dirigido
        self.tamano = 0


class Arista(object):
    """Clase lista de aristas implementación sobre lista."""

    def __init__(self):
        """Crea una lista de aristas vacía."""
        self.inicio = None
        self.tamano = 0

```

Figura 8. Definición de la estructura del TDA grafo

Además desde la figura 9 hasta la 19 se detalla la implementación de las funciones TDA mencionadas anteriormente, que serán los eventos que nos permitirán utilizar el TDA grafo.

```

def insertar_vertice(grafo, dato):
    """Inserta un vértice al grafo."""
    nodo = nodoVertice(dato)
    if (grafo.inicio is None or grafo.inicio.info > dato):
        nodo.sig = grafo.inicio
        grafo.inicio = nodo
    else:
        ant = grafo.inicio
        act = grafo.inicio.sig
        while(act is not None and act.info < nodo.info):
            ant = act
            act = act.sig
        nodo.sig = act
        ant.sig = nodo
    grafo.tamanio += 1

```

Figura 9. Interfaz o eventos del TDA grafo parte 1

```

def insertar_arista(grafo, dato, origen, destino):
    """Inserta una arista desde el vértice origen al destino."""
    agregar_arista(origen.adyacentes, dato, destino.info)
    if(not grafo.dirigido):
        agregar_arista(destino.adyacentes, dato, origen.info)

def agregar_arista(origen, dato, destino):
    """Agrega la arista desde el vértice origen al destino."""
    nodo = nodoArista(dato, destino)
    if (origen.inicio is None or origen.inicio.destino > destino):
        nodo.sig = origen.inicio
        origen.inicio = nodo
    else:
        ant = origen.inicio
        act = origen.inicio.sig
        while(act is not None and act.destino < nodo.destino):
            ant = act
            act = act.sig
        nodo.sig = act
        ant.sig = nodo
    origen.tamanio += 1

```

Figura 10. Interfaz o eventos del TDA grafo parte 2

```

def eliminar_vertice(grafo, clave):
    """Elimina un vertice del grafo y lo devuelve si lo encuentra."""
    x = None
    if(grafo.inicio.info == clave):
        x = grafo.inicio.info
        grafo.inicio = grafo.inicio.sig
        grafo.tamanio -= 1
    else:
        ant = grafo.inicio
        act = grafo.inicio.sig
        while(act is not None and act.info != clave):
            ant = act
            act = act.sig
        if (act is not None):
            x = act.info
            ant.sig = act.sig
            grafo.tamanio -= 1
    if(x is not None):
        aux = grafo.inicio
        while(aux is not None):
            if(aux.adyacentes.inicio is not None):
                eliminar_arista(aux.adyacentes, clave)
            aux = aux.sig
    return x

```

Figura II. Interfaz o eventos del TDA grafo parte 3

```

def buscar_vertice(grafo, buscado):
    """Devuelve la direccion del elemento buscado."""
    aux = grafo.inicio
    while(aux is not None and aux.info != buscado):
        aux = aux.sig
    return aux

def buscar_arista(vertice, buscado):
    """Devuelve la direccion del elemento buscado."""
    aux = vertice.adyacentes.inicio
    while(aux is not None and aux.destino != buscado):
        aux = aux.sig
    return aux

```

Figura I2. Interfaz o eventos del TDA grafo parte 4

```
def tamano(grafo):
    """Devuelve el numero de vertices en el grafo."""
    return grafo.tamano
```

```
def grafo_vacio(grafo):
    """Devuelve true si el grafo esta vacio."""
    return grafo.inicio is None
```

Figura 13. Interfaz o eventos del TDA grafo parte 5

```
def eliminar_arista(vertice, destino):
    """Elimina una arsita del vertice y lo devuelve si lo encuentra."""
    x = None
    if(vertice.inicio.destino == destino):
        x = vertice.inicio.info
        vertice.inicio = vertice.inicio.sig
        vertice.tamano -= 1
    else:
        ant = vertice.inicio
        act = vertice.inicio.sig
        while(act is not None and act.destino != destino):
            ant = act
            act = act.sig
        if (act is not None):
            x = act.info
            ant.sig = act.sig
            vertice.tamano -= 1
    return x
```

Figura 14. Interfaz o eventos del TDA grafo parte 6

```

def existe_paso(grafo, origen, destino):
    """Barrido en profundidad del grafo."""
    resultado = False
    if(not origen.visitado):
        origen.visitado = True
        vadyacentes = origen.adyacentes.inicio
        while(vadyacentes is not None and not resultado):
            adyacente = buscar_vertice(grafo, vadyacentes.destino)
            if(adyacente.info == destino.info):
                return True
            elif(not adyacente.visitado):
                resultado = existe_paso(grafo, adyacente, destino)
                vadyacentes = vadyacentes.sig
    return resultado

```

Figura 15. Interfaz o eventos del TDA grafo parte 7

```

def adyacentes(vertice):
    """Muestra los adyacents del vertice."""
    aux = vertice.adyacentes.inicio
    while(aux is not None):
        print(aux.destino, aux.info)
        aux = aux.sig

def es_adyacente(vertice, destino):
    """Determina si el destino es adyacente directo."""
    resultado = False
    aux = vertice.adyacentes.inicio
    while(aux is not None and not resultado):
        if(aux.destino == resultado):
            resultado = True
        aux = aux.sig
    return resultado

```

Figura 16. Interfaz o eventos del TDA grafo parte 8

```

def marcar_no_visitado(grafo):
    """Marca todos los vertices del grafo como no visitados."""
    aux = grafo.inicio
    while(aux is not None):
        aux.visitado = False
        aux = aux.sig

def barrido_vertices(grafo):
    """Realiza un barrido de la grafo mostrando sus valores."""
    aux = grafo.inicio
    while(aux is not None):
        print(aux.info)
        aux = aux.sig

```

Figura 17. Interfaz o eventos del TDA grafo parte 9

```

def barrido_profundidad(grafo, vertice):
    """Barrido en profundidad del grafo."""
    while(vertice is not None):
        if(not vertice.visitado):
            vertice.visitado = True
            print(vertice.info)
            adyacentes = vertice.adyacentes.inicio
            while(adyacentes is not None):
                adyacente = buscar_vertice(grafo, adyacentes.destino)
                if(not adyacente.visitado):
                    barrido_profundidad(grafo, adyacente)
                adyacentes = adyacentes.sig
        vertice = vertice.sig

```

Figura 18. Interfaz o eventos del TDA grafo parte 10

```

def barrido_amplitud(grafo, vertice):
    """Barrido en amplitud del grafo."""
    cola = Cola()
    while(vertice is not None):
        if(not vertice.visitado):
            vertice.visitado = True
            arribo(cola, vertice)
            while(not cola_vacia(cola)):
                nodo = atencion(cola)
                print(nodo.info)
                adyacentes = nodo.adyacentes.inicio
                while(adyacentes is not None):
                    adyacente = buscar_vertice(grafo, adyacentes.destino)
                    if(not adyacente.visitado):
                        adyacente.visitado = True
                        arribo(cola, adyacente)
                    adyacentes = adyacentes.sig
                vertice = vertice.sig

```

Figura 19. Interfaz o eventos del TDA grafo parte II

Como habrán podido notar muchas de las funciones vistas en las figuras anteriores son similares a las del TDA lista así que no las volveremos a analizar –dado que solo presentan pequeños cambios de adaptación que quedarán a cargo del lector interpretarlas–, pero sí nos detendremos para desglosar analíticamente las actividades que se realizan en las funciones “*barrido en profundidad*”, “*barrido en amplitud*”, “*existe paso*” e “*insertar arista*”, de las cuales podemos resaltar lo siguiente: el barrido en profundidad parte desde una lista de vértices y consiste en tratar el nodo inicial y luego tratar todos sus nodos adyacentes de manera recursiva –cada vez que trata un nodo se lo marca como visitado para no caer en un ciclo–, es decir que podemos conocer todos los vértices accesibles desde dicho vértice, si luego de terminar las llamadas recursivas quedaron vértices sin visitar se toma el siguiente de la lista –no visitado– y se repite el procedimiento hasta que todos los vértices estén marcados como visitados. En cambio en el barrido en amplitud parte de una lista de vértices creando una cola con el vértice inicial, luego mientras la cola no está vacía realiza las siguientes actividades, primero atiende el primer vértice de la cola lo trata y lo marca como visitado y luego agrega a la cola todos los vértices adyacentes a este que estén no visitados –para evitar caer en un ciclo–; una vez que la cola está vacía si quedaron vértices sin visitar en la lista se toma el siguiente –no visitado– y se repite el procedimiento hasta que todos los vértices hayan sido visitados. Por su parte para determinar si existe paso se utiliza el barrido en profundidad para determinar si es posible acceder al vértice destino desde el origen, solamente que al final las llamadas recursivas no se vuelve a tomar el siguiente vértice de la lista, si no se encontró el vértice destino significa que no existe paso entre origen y destino. Por su parte la función insertar arista nos simplifica la tarea de insertar una arista, si el grafo es dirigido agrega la arista del vértice origen al destino y si es no dirigido además la inserta desde el destino hacia el origen. Esto permite un manejo transparente al momento de insertar una arista.

Por su parte un grafo no dirigido es un conjunto de vértice V y aristas A , es decir $G = (V, A)$, que se diferencia de un grafo dirigido en que cada arista A no es un par ordenada (a, b) sino que $(a, b) = (b, a)$,

es decir no tiene asociado un sentido, los grafos no dirigidos se los suele denominar simplemente grafos. Gráficamente los vértices están unidos con una línea en lugar de una flecha, como el grafo que se presenta en la figura 20. Respecto de todo lo visto anteriormente aplica también para grafos no dirigidos, es decir: técnicas de representación, barridos e implementación. Respecto de esto último ya mencionamos que solo debemos indicar mediante la variable booleana *dirigido*, si el grafo es dirigido tendrá valor verdadero y si es no dirigido falso.

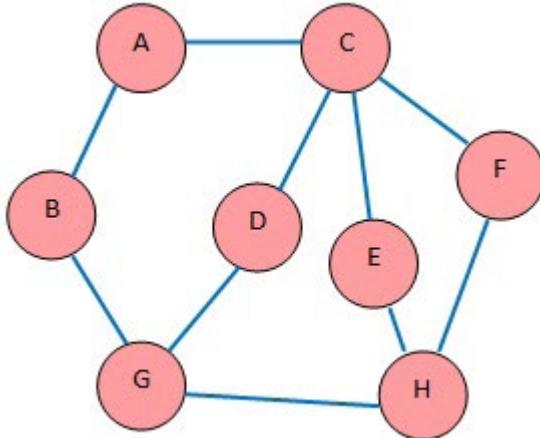


Figura 20. Grafo no dirigido

Encontrando el camino más corto con la ayuda del algoritmo de Dijkstra

Este algoritmo nos permite encontrar el camino más corto entre dos vértices de un grafo cuya aristas tienen peso positivo –funciona tanto para grafos dirigidos como no dirigidos–. Su nombre proviene de su inventor Edsger Dijkstra, científico de ciencias de la computación holandés, quien lo creó en 1956 y fue publicado unos años más tarde². Si el grafo tiene aristas con peso negativo se puede utilizar el algoritmo de Bellman³-Ford⁴.

La esencia del algoritmo consiste en partir desde un nodo origen desde el cual se calculan las distancias a cada uno de sus vértices adyacentes (no tratados), luego se selecciona el vértice adyacente más cercano no tratado y se vuelve a repetir este procedimiento; el algoritmo termina cuando todos los vértices del grafo han sido tratados. Tenga en cuenta que si el grafo es dirigido previamente debería determinar si existe paso desde el origen al destino para no ejecutar el algoritmo en vano.

² Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1: 269–271. doi: 10.1007/BF01386390 (<http://dx.doi.org/10.1007/BF01386390>).

³ Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics. 16: 87–90.

⁴ Ford, Lester R. Jr. (August 14, 1956). Network Flow Theory. Paper P-923. Santa Monica, California: RAND Corporation.

La mejor manera de entender este algoritmo es analizar su mecánica de funcionamiento a través de un ejemplo, para esto vamos a ver todas las etapas de iteración del algoritmo partiendo del grafo de la siguiente figura para encontrar el camino más corto desde el vértice T hasta el Z:

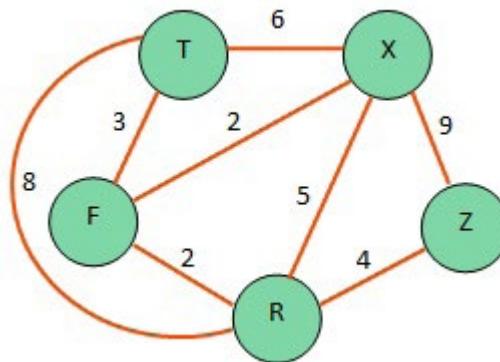


Figura 21. Grafo no dirigido cuyas aristas tiene peso positivo

Primero debemos asignar valor infinito a las distancias desde el inicio a todos los nodos a excepción del inicial que tendrá valor cero y no tiene vértice anterior, además todos los vértices están marcados como no visitados, como se observa en la siguiente tabla.

Vértice	Peso desde inicio	Vértice anterior	Visitado
T	0	-	No
X	∞		No
F	∞		No
R	∞		No
Z	∞		No

Ahora tomamos el vértice de menor distancia no visitado y calculamos la distancia a todos los vértices adyacentes a este no visitado, es decir se elige el vértice T y sus adyacentes son X a 6, F a 3, R 8. Si dichas distancias son menores a las actuales se actualizan dichos valores –en este caso son todas menores dado que tiene valor infinito–, además si se cambian las distancias se actualiza el vértice anterior de dichos nodos con el valor del vértice actual que se está tratando (en este caso se coloca T) y luego se indica como tratado el vértice actual como se observa a continuación.

Vértice	Peso desde inicio	Vértice anterior	Visitado
T	0	-	Si
X	≈ 6	T	No
F	≈ 3	T	No
R	≈ 8	T	No
Z	∞		No

Para continuar, volvemos a elegir el vértice que tiene menor distancia de los no tratado, en este caso F, se calcula la distancia a sus adyacentes –teniendo en cuenta que se debe considerar la distancia para requerida para llegar al vértice F desde el anterior (3) por lo cual las distancias serán X a 5 y R a 5, en ambos casos los vértices están a 2 de distancia del vértice actual por lo cual la distancia total es

5 ($3 + 2$), como las distancias son menores a las actuales se actualizan dichos valores y se actualiza el vértice anterior por el actual y se indica que el vértice F fue visitado como se ve en la tabla.

Vértice	Peso desde inicio	Vértice anterior	Visitado
T	0	-	Si
X	≈ 6.5	T F	No
F	≈ 3	T	Si
R	≈ 8.5	T F	No
Z	∞		No

En la siguiente iteración ocurre que tenemos dos vértices que comparten la distancia mínima, para lo cual el criterio de desempate es arbitrario, continuando con el ejemplo tomamos el vértice X , las distancias a sus adyacentes son R a 10, Z a 14, por lo cual solo se actualiza la distancia a Z ya que la nueva distancia a R es mayor que la que tiene actualmente, después actualizamos el vértice anterior de Z y indicamos que X ya está visitado. Como podemos observar a continuación en la tabla.

Vértice	Peso desde inicio	Vértice anterior	Visitado
T	0	-	Si
X	≈ 6.5	T F	Si
F	≈ 3	T	Si
R	≈ 8.5	T F	No
Z	≈ 14	X	No

Seguimos repitiendo el mismo procedimiento, en este caso el de menor distancia es el vértice R y su adyacente es Z a 9, dado que los demás vértices ya han sido visitados, como la distancia es menor que la actual procedemos a actualizar la distancia y también el vértice anterior por el actual, por último nos resta indicar que el vértice R ya fue visitado como se observa en la tabla.

Vértice	Peso desde inicio	Vértice anterior	Visitado
T	0	-	Si
X	≈ 6.5	T F	Si
F	≈ 3	T	Si
R	≈ 8.5	T F	Si
Z	≈ 14	X R	No

Finalmente solo nos queda un vértice por visitar Z , por lo cual esta es la iteración final y solo marcamos el vértice como visitado y finaliza el algoritmo.

Vértice	Peso desde inicio	Vértice anterior	Visitado
T	0	-	Si
X	≈ 6.5	T F	Si
F	≈ 3	T	Si
R	≈ 8.5	T F	Si
Z	≈ 14	X R	Si

Una vez finalizado el algoritmo ya con toda la tabla completa, *¿Cómo determino el camino mínimo y su costo?*, continuando con nuestro ejemplo de encontrar el camino más corto para llegar del vértice T a Z: para determinar el costo mínimo del camino nos fijamos en la columna peso o distancia del vértice destino, es decir, para este caso Z y el costo es 9, mientras que para obtener el camino –es decir los vértices por los que debe pasar– se construye desde atrás hacia adelante: nuevamente nos posicionamos en la fila de Z y no fijamos el vértice anterior (R) y luego repetimos este proceso hasta llegar al vértice origen –es decir el que no tiene anterior–, por lo cual para este caso el camino más corto es T-F-R-Z, como se describe gráficamente en la siguiente tabla.



Vértice	Peso desde inicio	Vértice anterior	Visitado
T	0	-	Si
X	≈6 5	T F	Si
F	≈3	T	Si
R	≈8 5	T F	Si
Z	≈14 9	X R	Si

De acuerdo al proceso descrito anteriormente necesitaremos un estructura auxiliar para poder implementar el algoritmo de Dijkstra, para esto utilizaremos una cola de prioridad (basada en un montículo mínimo). Esta se encargara de ordenar los vértices de acuerdo a la distancia hasta el vértice origen, para que al realizar una atención en cada iteración podamos tomar el vértice de menor distancia. Si se opta por otra manera de implementación sin cola, se deben utilizar dos vectores auxiliares de n elementos (donde n representa el número de vértices del grafo), uno para guardar la distancia de cada vértice al origen y el otro para indicar si dicho vértice fue visitado.

Pero hablando en términos de eficiencia, *¿Cuál es la complejidad del algoritmo de Dijkstra?* Esta dependerá de la implementación utilizada: si se usa cola de prioridad, la complejidad es del orden de $O(a \log_2 n)$ donde a es el número de aristas y n el número de vértices, en cambios si se opta por usar vectores la complejidad es del orden de $O(n^2)$.

A continuación, en la figura 22, se pude ver la implementación del algoritmo de Dijkstra utilizando colas de prioridad, la devuelve una pila con los resultados de la tabla para obtener el camino.

```

def dijkstra(grafo, origen, destino):
    """Algoritmo de Dijkstra para hallar el camino mas corto."""
    no_visitados = Heap(tamanio(grafo))
    camino = Pila()
    aux = grafo.inicio
    while(aux is not None):
        if(aux.info == origen):
            arribo_h(no_visitados, [aux, None], 0)
        else:
            arribo_h(no_visitados, [aux, None], inf)
        aux = aux.sig

    while(not heap_vacio(no_visitados)):
        dato = atencion_h(no_visitados)
        apilar(camino, dato)
        aux = dato[1][0].adyacentes.inicio
        while(aux is not None):
            pos = buscar_h(no_visitados, aux.destino)
            if(no_visitados.vector[pos][0] > dato[0] + aux.info):
                no_visitados.vector[pos][1][1] = dato[1][0].info
                cambiar_prioridad(no_visitados, pos, dato[0] + aux.info)
            aux = aux.sig
    return camino

```

Figura 22. Implementación algoritmo de Dijkstra

Expandiendo un árbol a través de un grafo que conecte todos los vértices

Suponga un grafo no dirigido G conexo con aristas etiquetadas con peso, un árbol de expansión mínimo para el grafo G , es un árbol libre que conecta todos los nodos con las aristas de menor peso y además no debe existir ciclo entre sus nodos. A continuación en la figura 23 se observa un grafo y todos los posibles árboles de expansión que se pueden obtener de dicho grafo, de los cuales el de la izquierda es el árbol de expansión mínimo que tiene un peso total de 6, nótese que no pueden existir ciclos.

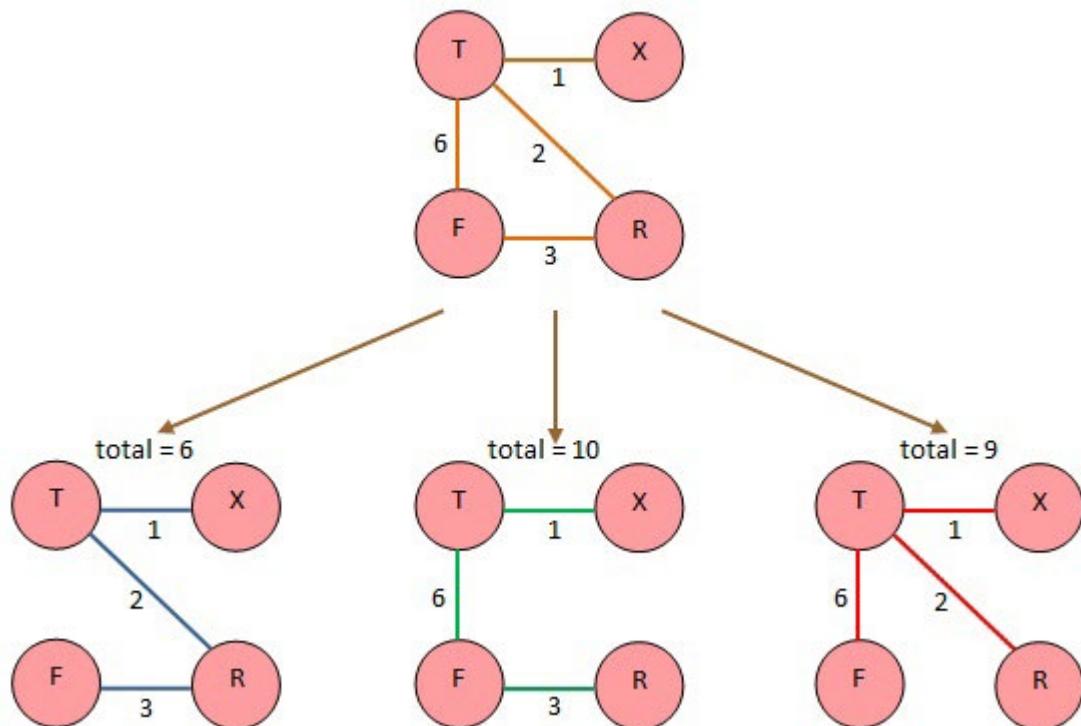


Figura 23. Árboles de expansión de un grafo

Los árboles de expansión mínimos son muy útiles por ejemplo para determinar una ruta para recorrer todos los puntos en menor tiempo, saber cuál es el orden de las actividades de un proyecto para poder minimizar costos, suponiendo en ambos caso que existiera múltiples caminos a al menos un vértice. Aunque en ocasiones también puede ser útil hallar el árbol de expansión máximo, por ejemplo: si las aristas del grafo representan beneficios y se quiere maximizarlos, en este caso se deben invertir los algoritmos; otra alternativa es transformar el peso de las aristas de beneficios a costos y hallar el árbol de expansión mínimo. Para encontrar el árbol de expansión mínimo existen dos técnicas populares el algoritmo de Prim y el de Kruskal que analizaremos en detalle a continuación.

Arranquemos estudiando el algoritmos de Prim que nos permite encontrar el conjunto de arista que une todos los vértices de un grafo con el costo mínimo formando un árbol de expansión, el cual tiene diversas aplicaciones como ya se mencionó previamente. Fue desarrollado en 1930 por el

matemático checo Vojtěch Jarník⁵ y luego redescubierto y republicado por Robert C. Prim⁶ en 1957. Este algoritmo tiene una complejidad del orden de $O(a \log v)$ si utilizamos cola de prioridad, donde a es el número de aristas del grafo y v la cantidad de vértices.

Comprendamos la esencia de este algoritmo a través de un ejemplo, sigamos trabajando con el grafo de la figura 21. Inicialmente creamos un árbol con un nodo con el valor del vértice inicial –elegido arbitrariamente–, para nuestro ejemplo arrancaremos desde T . Después debemos tomar la arista de menor peso que conecte a alguno de los vértices adyacentes que aún no esté en el árbol. Como se observa en la figura 24 las opciones son los vértices F a 3, R a 8 y X a 6, de los cuales se selecciona el F y se agrega el árbol como hijo de T .

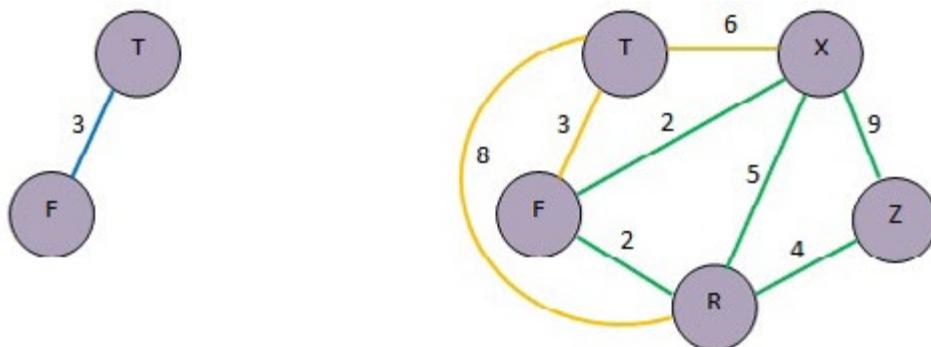


Figura 24. Algoritmo de Prim iteración 1

Luego se vuelve a repetir el mismo procedimiento hasta que el árbol tenga la misma cantidad de nodos que vértices en el grafo, pero al momento de seleccionar la arista de menor peso se suman además las aristas del último vértice agregado al árbol. Por lo que ahora las posibles alternativas son desde T : X a 6 y R a 8 y desde F : X a 2 y R a 2. Como verán, tenemos dos aristas con el mínimo peso, la elección es de manera arbitraria. Para este ejemplo tomaremos la arista de F a X de peso 2 y agregaremos X al árbol como hijo F , como se observa a continuación. Nótese que las aristas que conectan dos vértices que ya estén en el árbol, se eliminarán ya sea porque forman parte del árbol o porque dichos vértices ya están conectados en el árbol con un camino más corto.

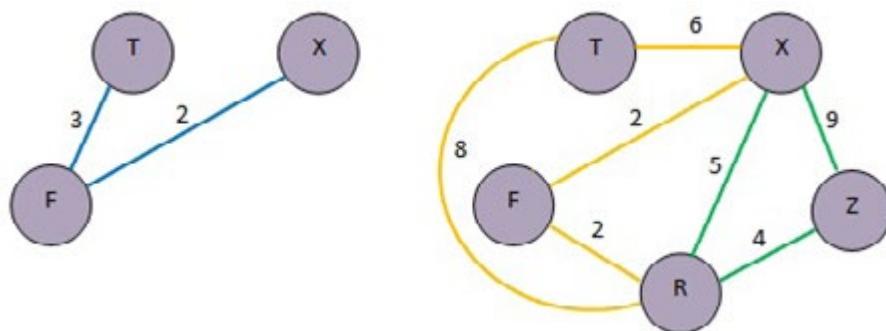


Figura 25. Algoritmo de Prim iteración 2

⁵ Jarník, V. (1930), "O jistémproblémum minimálním" [About a certain minimal problem], Práce Moravské Přírodovědecké Společnosti (in Czech), 6 (4): 57–63.

⁶ Prim, R. C. (November 1957), "Shortest connection networks And some generalizations", Bell System Technical Journal, 36 (6): 1389–1401, Bibcode:1957BSTJ...36.1389P.

En la siguiente iteración las posibles aristas a utilizar son desde T : R a 8, desde F : R a 2 y desde X : R a 5 y Z a 9, por lo cual se elige la arista de F a R y se agrega R al árbol como hijo de F , como se ve en la siguiente figura.

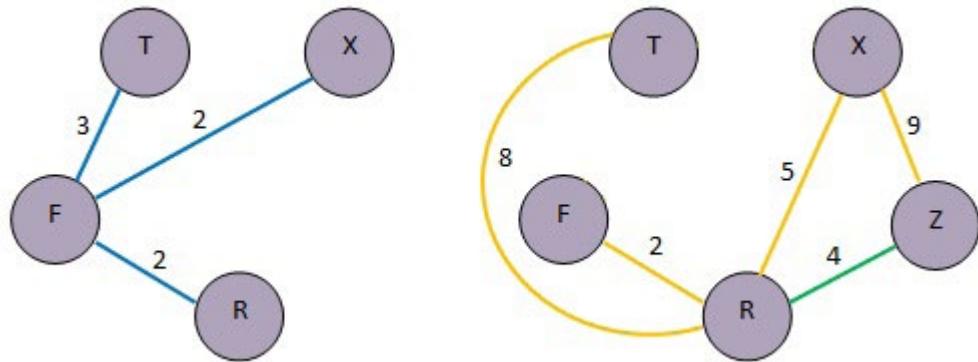


Figura 26. Algoritmo de Prim iteración 3

Ya solo resta agregar un vértice al árbol por lo cual esta será la última iteración del algoritmo, en la que quedan dos posibilidades de R : Z a 4 y desde X : Z a 9, de las cuales se elige de R a Z y se agrega Z al árbol como hijo de R , como se describe en la siguiente figura, para finalmente obtener el árbol de expansión mínimo (a la izquierda en la figura 27), cuyo peso total es 11.

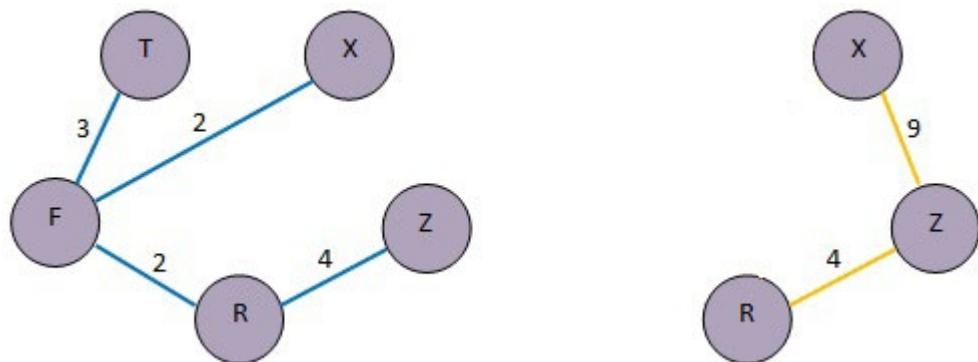


Figura 27. Algoritmo de Prim iteración 4

Para implementar el algoritmo de Prim utilizaremos una cola de prioridad, usando un montículo mínimo, para almacenar las aristas disponibles para elegir, de esta manera en cada iteración sacaremos la arista de menor peso que está en el frente de la cola, el árbol de expansión se guarda en un vector auxiliar llamado bosque en forma de par ordenado parente-hijo (o vértice origen-destino). En la figura 28 y 29 se presenta la implementación completa del algoritmo.

```

def prim(grafo):
    """Algoritmo de Prim para hallar el árbol de expansión mínimo."""
    bosque = [[grafo.inicio.info]]
    aristas = []
    adyacentes = grafo.inicio.adyacentes.inicio
    while(adyacentes is not None):
        aristas.append([grafo.inicio.info, adyacentes.destino, adyacentes.info])
        adyacentes = adyacentes.sig
    while(len(bosque[0]) // 2 < tamano(grafo)-1):
        menor = inf
        menor_arista = None
        tipo = None
        for arista in aristas:
            origen = arista[0]
            destino = arista[1]
            if(origen not in bosque[0] and destino in bosque[0]):
                if(arista[2] < menor):
                    menor, menor_arista = arista[2], arista
                    tipo = True
            if(origen in bosque[0] and destino not in bosque[0]):
                if(arista[2] < menor):
                    menor, menor_arista = arista[2], arista
                    tipo = False

```

Figura 28. Implementación algoritmo de Prim parte 1

```

arista = aristas.pop(aristas.index(menor_arista))
if(len(bosque[0]) != 1):
    bosque[0] += [arista[0], arista[1]]
else:
    bosque.pop()
    bosque.append([arista[0], arista[1]])
aux = None
if(tipo):
    aux = buscar_vertice(grafo, arista[0])
else:
    aux = buscar_vertice(grafo, arista[1])
adyacentes = aux.adyacentes.inicio
while(adyacentes is not None):
    aristas.append([aux.info, adyacentes.destino, adyacentes.info])
    adyacentes = adyacentes.sig
return bosque

```

Figura 29. Implementación algoritmo de Prim parte 2

Otra alternativa que nos permite hallar el árbol de expansión mínimo es el algoritmo de Kruskal, fue publicado por el matemático estadunidense Joshep Kruskal⁷ y de ahí proviene su nombre. Además de estos dos algoritmos existen otros similares menos conocidos como el algoritmo de

⁷ Kruskal, J. B. (1956). «On the shortest spanning subtree and the traveling salesman problem». Proceedings of the American Mathematical Society (7): 48-50. JSTOR 2033241.

Boruvka⁸ utilizado como un método eficiente para construir la red eléctrica de Moravia en 1926⁹. Este algoritmo tiene una complejidad del orden de $O(a \log v)$ al igual que el visto anteriormente dado que se implementa con cola de prioridad.

Para demostrar que este algoritmo devuelve el mismo resultado que el anterior trabajaremos con el mismo grafo de ejemplo. Al iniciar el algoritmo creamos un bosque donde cada uno de los nodos del grafo es un árbol solo con raíz y además se crea una cola de prioridad –utilizando un montículo mínimo– a la cual se agregan todas las aristas del grafo, con su origen-destino y peso, como el montículo es mínimo las aristas quedarán ordenadas de menor a mayor, como se puede observar a continuación.

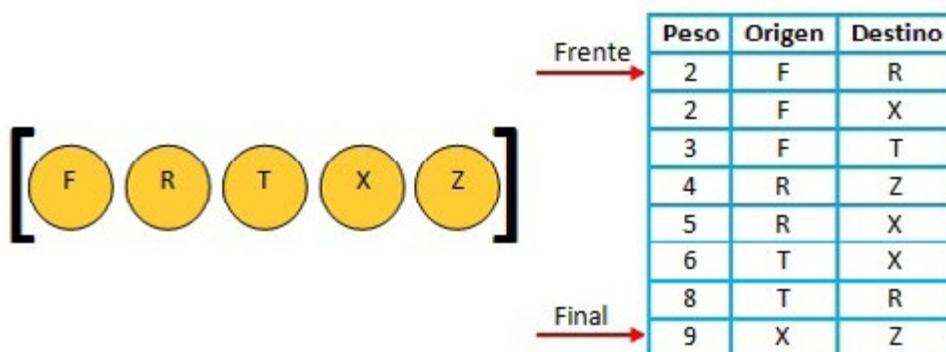


Figura 30. Algoritmo de Kruskal configuración inicial

Ahora mientras el tamaño del bosque sea mayor que uno –es decir que haya más de un árbol en el bosque–, ya que cuando el bosque tenga tamaño uno significa que tendremos todos los nodos conectados, repetiremos el siguiente procedimiento: hacemos la atención de la primera arista de la cola para este caso la arista de F hasta R con peso 2. Si dicha arista conecta dos árboles del bosque, se quita del bosque el árbol que contiene el nodo igual al destino de la arista y se lo agrega al árbol que contenga el nodo igual al origen, como hijo de este; caso contrario se descarta la arista dado que implica que dicho nodos ya están conectados en algún árbol del bosque. Para este caso se quita el árbol con raíz R y se agrega como hijo del árbol con raíz F , esto se puede ver en la siguiente figura.

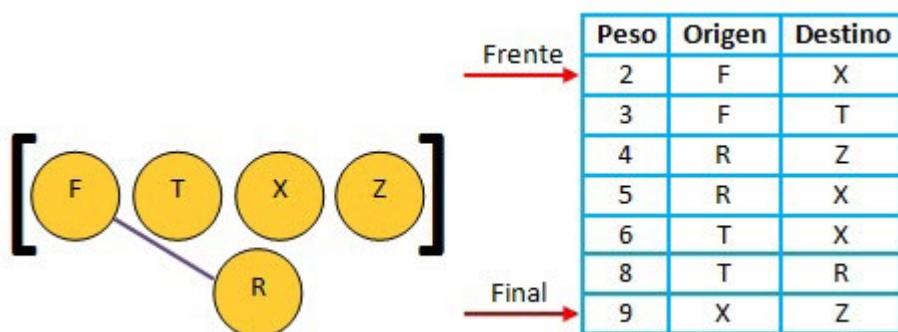


Figura 31. Algoritmo de Kruskal iteración 1

-
- 8 Borůvka, Otakar (1926). "O jistém problémum minimálním" [About a certain minimal problem]. Práce Mor. Přírodotv. Spol. V Brně III (in Czech and German).3: 37–58.
- 9 Borůvka, Otakar (1926). "Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí (Contribution to the solution of a problem of economical construction of electrical networks)". Elektronický Obzor (in Czech). 15: 153–154.

En la siguiente iteración, la arista que se atiende es de F hasta X cuyo peso es 2, como la arista conecta dos árboles del bosque se quita el árbol que contiene X del bosque y se agrega como hijo de F , como se detalla en la siguiente figura. Como las aristas están almacenadas en una cola de prioridad mínima por su campo peso en cada atención se extraerá siempre la de menor peso.

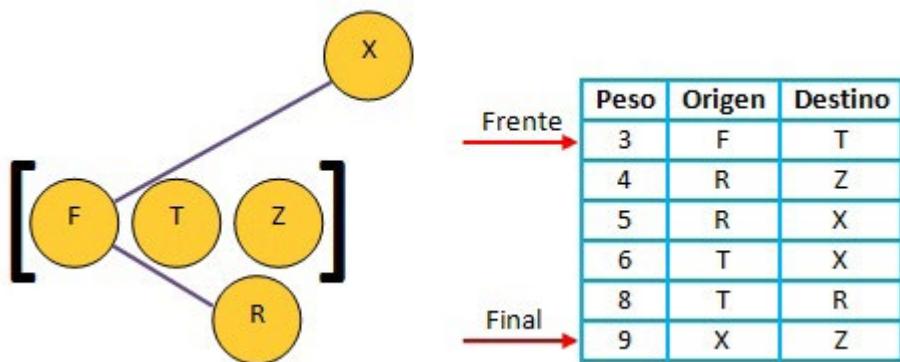


Figura 32. Algoritmo de Kruskal iteración 2

Continuando con el ejemplo es el turno de atender la arista de peso 3 que va desde F hasta T , esta también une dos árboles del bosque por lo que volvemos a repetir el procedimiento y combinamos los dos árboles en uno solo como en los casos anteriores. Como se aprecia a continuación ya solo quedan dos árboles en el bosque.

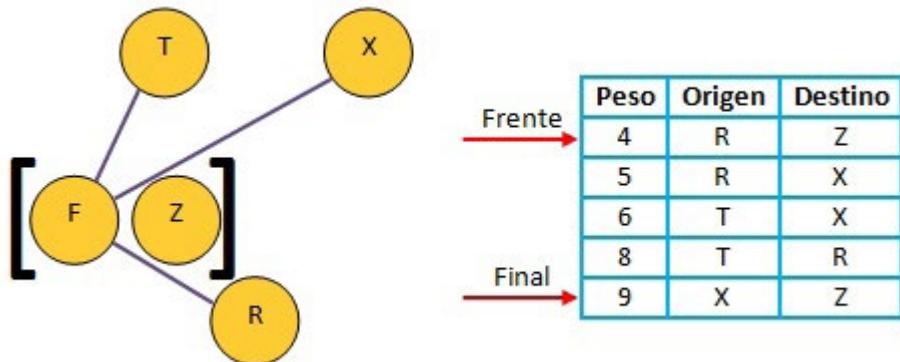


Figura 33. Algoritmo de Kruskal iteración 3

Por último la arista que se encuentra en el frente de cola para ser atendida tiene peso 4 y va desde R a Z , en esta ocasión también la arista conecta dos árboles del bosque por lo cual se agrega la Z como hijo de R ; al terminar esta iteración el bosque tiene tamaño uno, lo que indica que ya hemos encontrado el árbol de expansión mínimo y se detiene el algoritmo. Si alguna de las aristas que se atiende no conecta dos árboles se descarta dado que implica que dichos valores ya están incluidos en un mismo árbol. En la siguiente figura, a la izquierda está el árbol de expansión mínimo obtenido el cual tiene un peso total de 11, como se habrá notado es el mismo que obtuvimos con el algoritmo anterior.

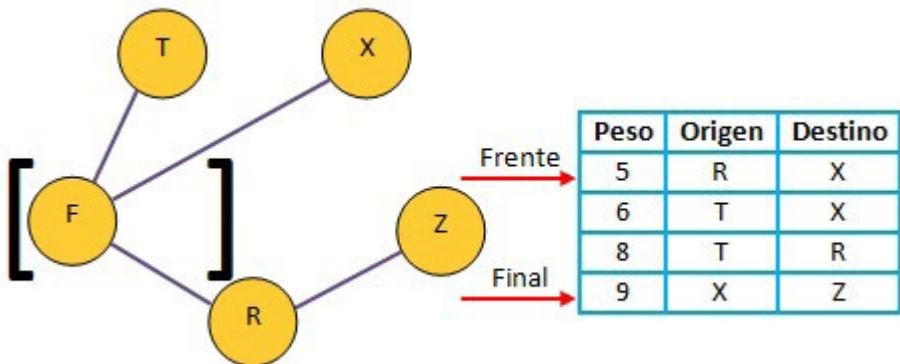


Figura 34. Algoritmo de Kruskal iteración 4

En lo que respecta a la implementación del algoritmo de Kruskal basado en el uso de cola de prioridad y se utiliza un vector denominado bosque para guardar el árbol de expansión mínimo, como se detalla a continuación en la figura 35 y 36.

```
def kruskal(grafo):
    """Algoritmo de Kruskal para hallar el árbol de expansión mínimo."""
    bosque = []
    aristas = Heap(tamanio(grafo) ** 2)
    aux = grafo.inicio
    while(aux is not None):
        bosque.append([aux.info])
        adyacentes = aux.adyacentes.inicio
        while(adyacentes is not None):
            arribo_h(aristas, [aux.info, adyacentes.destino], adyacentes.info)
            adyacentes = adyacentes.sig
        aux = aux.sig
```

Figura 35. Implementación algoritmo de Kruskal parte I

```

while(len(bosque) > 1 and not heap_vacio(aristas)):
    dato = atención_h(aristas)
    origen = None
    for elemento in bosque:
        if(dato[1][0] in elemento):
            origen = bosque.pop(bosque.index(elemento))
    destino = None
    for elemento in bosque:
        if(dato[1][1] in elemento):
            destino = bosque.pop(bosque.index(elemento))
    if(origen is not None and destino is not None):
        if(len(origen) > 1 and len(destino) = 1):
            destino.insert(0, dato[1][0])
        elif(len(destino) > 1 and len(origen) = 1):
            origen.append(dato[1][1])
        elif(len(destino) > 1 and len(origen) > 1):
            origen += [dato[1][0], dato[1][1]]
            bosque.append(origen+destino)
        else:
            bosque.append(origen)
    return bosque

```

Figura 36. Implementación algoritmo de Kruskal parte 2

Volviendo un poco para atrás, mencionamos que a veces es útil hallar el árbol de expansión máximo, pero *¿Cómo hacemos para hallarlo?* Los algoritmos son los mismos que utilizamos anteriormente, pero para que en cada iteración tomen la arista de mayor peso, debemos utilizar un montículo máximo para la cola de prioridad. Otra alternativa para encontrar el árbol de expansión máximo sin modificar los algoritmos hechos para hallar el árbol de expansión mínimo, es transformar el peso de las aristas a mínimo. Para hacer esto primero buscamos la arista de mayor peso del grafo y luego a dicho valor le restamos el original de cada arista, como se puede observar en la siguiente tabla. Nótese que ahora las aristas que antes tenían mayor peso, ahora tienen el menor peso y las menores ahora son mayores, lo cual nos permite aplicar los algoritmos desarrollados para hallar el árbol mínimo.

Arista	Peso	Nuevo Peso
T-Z	3	$9 - 3 = 6$
F-X	2	$9 - 2 = 7$
T-X	6	$9 - 6 = 3$
T-R	8	$9 - 8 = 1$
F-R	2	$9 - 2 = 7$
R-Z	4	$9 - 4 = 5$
X-Z	9	$9 - 9 = 0$
R-X	5	$9 - 5 = 4$

Cabe aclarar que para los dos algoritmos descritos anteriormente si el grafo no es conexo, se encontrarán los árboles de expansión para cada uno de los componentes conexos que forman dicho grafo.

Para terminar nos queda probar a través de un ejemplo el uso del TDA grafo para la resolución de un problema. Para lo cual continuaremos trabajando con el grafo que hemos usado para explicar los distintos algoritmos –el de la figura 21–. Sobre el cual realizaremos un barrido en profundidad y en amplitud, además determinaremos el camino más corto desde el vértice T hasta Z , y también debemos hallar el árbol de expansión mínimo de dicho grafo, esto se aprecia a continuación en la figura 37.

```
from tda_pila import pila_vacia, desapilar
from tda_grafo import Grafo, barrido_profundidad, barrido_amplitud
from tda_grafo import marcar_no_visitado, buscar_vertice, existe_paso
from tda_grafo import dijkstra, kruskal

grafo = Grafo(False)
cargar_grafo(grafo)

barrido_profundidad(grafo, grafo.inicio)
marcar_no_visitado(grafo)
barrido_amplitud(grafo, grafo.inicio)

origen = buscar_vertice(grafo, 'T')
destino = buscar_vertice(grafo, 'Z')
camino_mas_corto = None
if(origen is not None and destino is not None):
    if(existe_paso(grafo, origen, destino)):
        camino_mas_corto = dijkstra(grafo, 'T', 'Z')
        fin = destino.info
        while(not pila_vacia(camino_mas_corto)):
            dato = desapilar(camino_mas_corto)
            if(fin == dato[1][0].info):
                print(dato[1][0].info)
                fin = dato[1][1]

marcar_no_visitado(grafo)
arbol_minimo = kruskal(grafo)
print(arbol_minimo)
```

Figura 37. Ejemplo de uso del TDA grafo

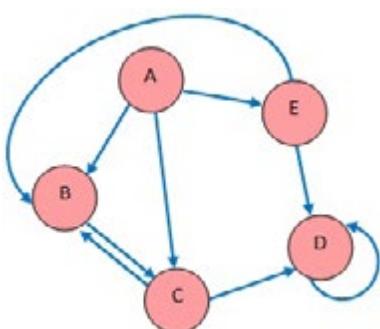
Describamos las acciones realizadas para lograr comprender el manejo del funcionamiento del TDA grafo mediante el uso de los eventos definidos: primero vamos a importar del TDA grafo las funciones que vamos a utilizar y del TDA pila las funciones que necesitaremos para procesar la salida del algoritmo de Dijkstra, primero vamos a crear un grafo no dirigido pasándole el parámetro *False* y procedemos a cargarlo con una función para cargar los vértices y las aristas del grafo del ejercicio –no se agrega el código de esta función, la misma queda a cargo del lector–. Continuamos realizando un barrido en profundidad y amplitud para ver los elementos del grafo para lo cual previamente se

marcan todos los vértices del grafo como no visitados. Luego se busca si existen los vértices origen y destino, en el caso que ambos vértices existan, determina si existe paso desde el origen hasta el destino. Si existe paso entonces obtenemos el camino más corto con Dijkstra y se procesa el resultado obtenido vaciando la pila mostrando el camino más corto y su peso. Nuevamente se marcan todos los vértices del grafo como no visitados para poder obtener el árbol de expansión mínimo, en este caso utilizamos Kruskal, para finalmente mostrar el árbol obtenido por dicho algoritmo.

Guía de ejercicios prácticos

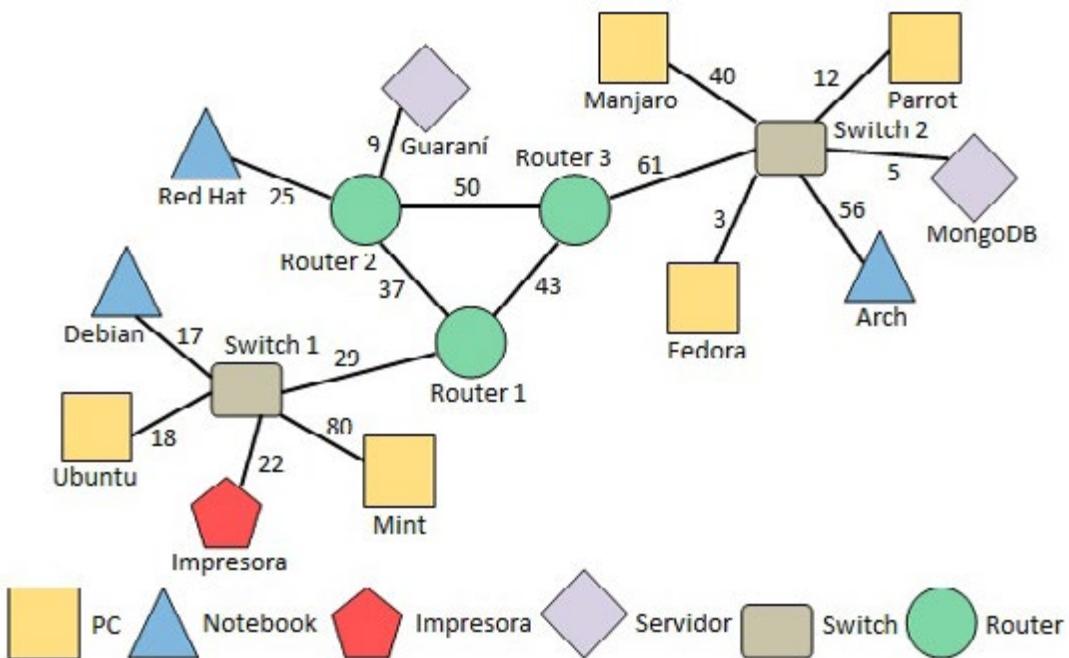
A continuación se plantean una serie de problemas, que se deberán resolver utilizando el TDA grafo, para lo cual se implementarán grafos dirigidos salvo que el ejercicio indique lo contrario.

1. Generar un grafo con 15 vértices aleatorios no repetidos (con números de 1 a 100), luego agregar 30 aristas –no repetidas– que conecten vértices de manera aleatoria, con etiquetas –también aleatorias– dentro del rango de 1 a 100, después resolver las siguientes actividades:
 - a. primero eliminar los vértices que hayan quedado desconectados, es decir, que ningún otro vértice tenga una arista que lo apunte y que de él no salga ninguna arista;
 - b. determinar el nodo con mayor cantidad de aristas que salen de él, puede ser más de uno;
 - c. determinar el nodo con mayor cantidad de aristas que llegan a él, puede ser más de uno;
 - d. indicar los vértices desde los cuales no se puede acceder a otro vértice;
 - e. contar cuantos vértice componen el grafo, dado que se genera aleatoriamente y se eliminan los vértices que quedan desconectados;
 - f. determinar cuántos vértices tienen un arista a sí mismo, es decir, un ciclo directo;
 - g. determinar la arista más larga, indicando su origen, destino y valor –puede ser más de una.
2. Sobre el siguiente dígrafo implementar los algoritmos necesarios para resolver las tareas que se presentan a continuación:
 - a. represéntelo como arreglo de listas de adyacencias y lista de listas de adyacencia;
 - b. cargue el valor de las etiquetas de todas las aristas;
 - c. encuentre el árbol de expansión mínima, para este punto considere el grafo como no dirigido;
 - d. agregue un arco de *E* hasta *C*;
 - e. encuentre el camino más corto de *A* hasta *D*.



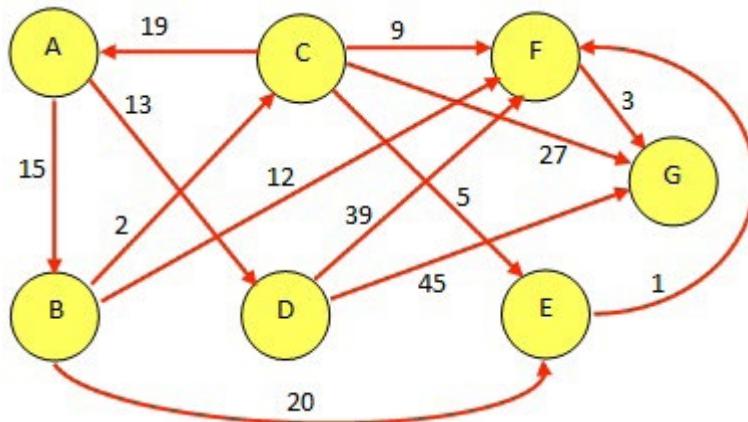
3. Implementar un grafo no dirigido que permita cargar puertos y las aristas que conecten dichos puertos, que permita resolver las siguientes tareas:
 - a. cada arista debe tener la distancia que separa dichos puertos;
 - b. realizar un barrido en profundidad desde el primer puerto en el grafo;
 - c. determinar el camino más corto desde puerto Madero al puerto de Rodas;
 - d. determinar el puerto con mayor número de aristas y eliminarlo.
4. Una empresa de telefonía celular dispone de la información de sus antenas, de las cuales se conoce: su ubicación (latitud y longitud), código de identificación, velocidad de transferencia en megabytes/segundos, y además las antenas a las que transmite y las distancias a cada una de estas. Implementar un algoritmo que permita resolver los siguientes requerimientos:
 - a. utilizar un grafo no dirigido;
 - b. cargar la información de antenas y la relación con las demás;
 - c. determinar el tamaño del grafo;
 - d. determinar el camino más corto para transmitir desde la antena ubicada en la capital de Mendoza a la antena ubicada en la capital de Misiones, utilizando el algoritmo de Dijkstra;
 - e. encontrar el árbol de expansión mínimo del grafo, utilizando Prim o Kruskal;
 - f. determinar si la antena con código “TGK-783” existe, de ser así mostrar toda su información.
5. Cargar el esquema de red de la siguiente figura en un grafo e implementar los algoritmos necesarios para resolver las tareas, listadas a continuación:
 - a. cada nodo además del nombre del equipo deberá almacenar su tipo: pc, notebook, servidor, *router*, *switch*, impresora;
 - b. realizar un barrido en profundidad y amplitud partiendo desde la tres notebook: Red Hat, Debian, Arch;
 - c. encontrar el camino más corto para enviar a imprimir un documento desde la pc: Manjaro, Red Hat, Fedora hasta la impresora;
 - d. encontrar el árbol de expansión mínima;
 - e. determinar desde que pc (no notebook) es el camino más corto hasta el servidor “Guaraní”;
 - f. indicar desde que computadora del *switch* oí es el camino más corto al servidor “MongoDB”;

- g. cambiar la conexión de la impresora al *router o2* y vuelva a resolver el punto b;
- h. debe utilizar un grafo no dirigido.



- 6. Partiendo del árbol genealógico de los dioses griegos que se observa en la imagen del ejercicio 20 de la guía de árboles (capítulo X), convertirlo en un grafo y resolver las siguientes actividades:
 - a. además del nombre de los dioses, deberá cargar una breve descripción de quien es o lo que representa, no más de 20 palabras;
 - b. deberá cargar todas las relaciones entre los distintos dioses: padre, madre, hijo, hermano, pareja, la etiquetas de dichas aristas son estos nombre de relación.
 - c. dado el nombre de un dios mostrar los hijos de este;
 - d. dado el nombre de un dios mostrar su nombre, padre, madre, hermanos y sus hijos;
 - e. determinar si existe relación directa entre dos vértice cualquiera, de ser así cual es la relación entre ambos;
 - f. dados dos dioses determinar el camino más corto entre estos y mostrarlo. Considere como camino más corto el que tenga menor número de aristas;
 - g. realizar un barrido en profundidad y amplitud de dicho grafo;
 - h. realizar un barrido mostrando el nombre de cada dios y el de su madre;
 - i. mostrar todos los ancestros de un determinado dios;

- j. mostrar todos los nietos de Cronos;
 - k. mostrar todos los hijos de Tea;
 - l. persista los datos del grafo en archivos, uno para los vértices y otro para las aristas.
7. Implementar los algoritmos necesarios para resolver las siguientes tareas sobre el grafo de la figura:
- a. barrido en profundidad y amplitud partiendo de A, C, F;
 - b. el camino más corto de A hasta F, de C hasta D, de B hasta G;
 - c. agrega una arista de C hasta A, de C hasta B, de G hasta D y vuelva a ejecutar los puntos del ítem anterior sin camino;
 - d. realice la representación de matriz de adyacencia del grafo.



8. Implementar un grafo social y los algoritmos necesarios para atender los siguientes requerimientos:
- a. cargar personas como vértices del grafo;
 - b. cargar aristas con las siguientes etiquetas: Twitter, Instagram, Facebook y la cantidad de retwitters y me gusta respectivamente, que representan si la persona del vértice origen sigue o es amigo de la persona del vértice destino;
 - c. hallar el árbol de expansión máximo para cada red social –considere el grafo como no dirigido para este punto–, es decir que las conexiones deben ser las de mayor peso –ósea el seguidor que tenga mayor interacción–; para lo cual si desea utilizar Prim o Kruskal sin modificar el código, puede determinar la arista de mayor peso entonces cuando aplique estos algoritmo el peso de cada arista será la arista de mayor peso menos el peso de la arista;
 - d. determine si es posible conectar la persona Guido Rossum con Mark Hamill a través de la red social Twitter;

- e. determine si es posible conectar la persona Tom Holland con Robert Downey a través de cualquier red social;
 - f. indique a todas las personas que sigue a través de su red de Instagram la persona Daisy Ridley.
9. Implementar un grafo no dirigido que permita administrar vuelos internacionales contemplando los siguientes requerimientos:
- a. de cada aeropuerto se conoce: su nombre, ubicación (latitud y longitud) y cantidad de pistas;
 - b. cada arista representa un viaje de un aeropuerto a otro, en cada una de esta puede haber más de un vuelo, de los cuales se conoce: hora de salida, hora de arribo, nombre de la empresa, costo del pasaje –considere que todos los pasajes cuestan lo mismo–, duración del viaje y distancia en km;
 - c. debe persistir los datos del grafo en archivos;
 - d. el grafo debe contener los aeropuertos de los siguientes países: Argentina, China, Brasil, Tailandia, Grecia, Alemania, Francia, Estados Unidos, Japón y Jamaica;
 - e. calcular el camino más corto desde el aeropuerto de Argentina a Tailandia considerando los siguientes criterios:
 - I. menor distancia,
 - II. menor duración de tiempo,
 - III. menor costo,
 - IV. menor número de escalas.
 - f. determinar todos los aeropuertos a los que se puede arribar desde Grecia de manera directa o indirecta.
10. Generar un grafo no dirigido con planetas de Star Wars y diseñar los algoritmos necesarios para resolver las siguientes actividades:
- a. los siguientes planetas deben estar en el grafo: Alderaan, Endor, Dagobah, Hoth, Tatooine, Kamino, Naboo, Mustafar, Scarif, Bespin, agregue 7 más;
 - b. genere al menos 4 aristas para cada uno de los planetas del grafo, no puede haber nodos con arcos a sí mismo;
 - c. encuentre el árbol de expansión mínima en cuanto a costos para recorrer todos los planetas;

- d. hallar el camino más corto desde:
- I. Tatooine hasta Dagobah,
 - II. Alderaan hasta Endor,
 - III. Hoth hasta Tatooine;
- e. determinar todos los planteas a los que se puede llegar desde Tatooine.
- ii. Construir un grafo de afinidad entre los superhéroes de la película “Capitán América: Civil War” y resuelva las siguientes tareas:
- a. cada vértice representara un superhéroe y las arista el nivel de afinidad (de 1 a 10);
 - b. cargar los siguientes super héroes: Capitan America, Iron Man, Black Widow, Falcon, The Winter Soldier, War Machine, Spider-Man, Ant-Man, Black Panther, Hawkeye, Scarlet Witch, Vision;
 - c. obtener el árbol de expansión máximo del *Team Cap (CapitanAmerica)* y del *Team Stark (IronMan)*, considerando solo las arista cuyo valor sea superior a 7;
 - d. obtener el árbol de expansión máximo completo de Black Widow y Black Panther.
- i2. Representar en un grafo no dirigido las actividades de un proyecto, en el cual los arcos representan el tiempo de dicha tarea y cuáles son las que pueden realizarse a continuación, teniendo en cuenta lo siguiente:
- a. además del tiempo las aristas deberá almacenar el nombre de la persona que la realiza –un vértice no puede tener más de una arista de la misma persona–;
 - b. determinar el tiempo total mínimo del proyecto, las personas que realizan cada tarea;
 - c. determinar el menor tiempo requerido partiendo desde la actividad “análisis de requerimientos” hasta llegar a la actividad “presentación de mockups”;
 - d. indicar todas las tareas que puede realizar la persona Harry.
- i3. El servicio postal de su ciudad le solicita desarrollar un algoritmo para mejorar el recorrido de sus carteros, considerando las siguientes cuestiones:
- a. cada vértice del grafo tendrá la dirección de los puntos de entrega, además si se trata de una carta o un paquete –en el caso de este último también se dispondrá del peso–;
 - b. cada arista tendrá las distancias entre los distintos puntos de entrega, debe utilizar un grafo no dirigido;
 - c. encontrar el árbol de expansión mínimo de acuerdo a las distancias de los puntos de entrega;

- d. encontrar el árbol de expansión máximo de acuerdo al peso de la entrega –en el caso de ser una carta considerar como peso cero–.
14. Implementar sobre un grafo no dirigido los algoritmos necesario para dar solución a las siguientes tareas:
- a. cada vértice representar un ambiente de una casa: cocina, comedor, cochera, quincho, baño 1, baño 2, habitación 1, habitación 2, sala de estar, terraza, patio;
 - b. cargar al menos tres aristas a cada vértice, y a dos de estas cárguele cinco, el peso de la arista es la distancia entre los ambientes, se debe cargar en metros;
 - c. obtener el árbol de expansión mínima y determine cuantos metros de cables se necesitan para conectar todos los ambientes;
 - d. determinar cuál es el camino más corto desde la habitación 1 hasta la sala de estar para determinar cuántos metros de cable de red se necesitan para conectar el *router* con el *Smart Tv*.
15. Se requiere implementar un grafo para almacenar las siete maravillas arquitectónicas modernas y naturales del mundo, para lo cual se deben tener en cuenta las siguientes actividades:
- a. de cada una de las maravillas se conoce su nombre, país de ubicación (puede ser más de uno en las naturales) y tipo (natural o arquitectónica);
 - b. cada una debe estar relacionada con las otras seis de su tipo, para lo que se debe almacenar la distancia que las separa;
 - c. hallar el árbol de expansión mínimo de cada tipo de las maravillas;
 - d. determinar si existen países que dispongan de maravillas arquitectónicas y naturales;
 - e. determinar si algún país tiene más de una maravilla del mismo tipo;
 - f. deberá utilizar un grafo no dirigido.
16. Implementar un grafo no dirigido para almacenar puntos turísticos de interés de un determinado país teniendo en cuenta los siguientes requerimientos:
- a. debe ser un grafo completo es decir que todos los vértices se deben conectar con todos;
 - b. cargar los siguientes lugares (con sus coordenadas de latitud y longitud) templos de: Atenas (Partenón), Zeus (Olimpia), Hera (Olimpia), Apolo (Delfos), Poseidón (Suniún), Artemisa (Éfeso) y Teatro de Dionisio (Acrópolis)
 - c. hallar el árbol de expansión mínimo partiendo de cualquiera de estos lugares;

- d. hallar el camino más corto para ir desde el templo de Atenea, el Partenón, en Atenas hasta el templo de Apolo, en Delfos.
17. Dado un grafo no dirigido con personajes de la saga Star Wars, implementar los algoritmos necesarios para resolver las siguientes tareas:
- cada vértice debe almacenar el nombre de un personaje, las aristas representan la cantidad de episodios en los que aparecieron juntos ambos personajes que se relacionan;
 - hallar el árbol de expansión máximo desde el vértice que contiene a C-3PO, Yoda y la princesa Leia;
 - determinar cuál es el número máximo de episodio que comparten dos personajes, e indicar todos los pares de personajes que coinciden con dicho número
 - cargue al menos los siguientes personajes: Luke Skywalker, Darth Vader, Yoda, Boba Fett, C-3PO, Leia, Rey, Kylo Ren, Chewbacca, Han Solo, R2-D2, BB-8;
 - indicar qué personajes aparecieron en los nueve episodios de la saga.
18. Implemente un grafo geográfico y los algoritmos necesarios que contemplen los siguientes requerimientos:
- el grafo debe ser no dirigido y almacenar los países de América y sus capitales (omitar los países que son islas);
 - las aristas representan la distancia entre las capitales de los países;
 - cada país solo puede tener aristas a los países limítrofes;
 - obtener el árbol de expansión mínimo;
 - determinar el camino más corto para ir desde Argentina hasta Canadá;
 - persista los datos en archivos –los vértices y aristas por separado–;
 - determinar el país con mayor cantidad de países adyacentes.
19. Se requiere implementar una red de ferrocarriles compuesta de estaciones de trenes y cambios de agujas (o desvíos). Contemplar las siguientes consideraciones:
- cada vértice del grafo no dirigido tendrá un tipo (estación o desvío) y su nombre, en el caso de los desvíos el nombre es un número –estos estarán numerados de manera consecutiva–;
 - cada desvío puede tener múltiples puntos de entrada y salida;
 - se deben cargar seis estaciones de trenes y doce cambios de agujas;

- d. cada cambio de aguja debe tener al menos cuatro salida o vértices adyacentes;
 - e. y cada estación como máximo dos salidas o llegadas y no puede haber dos estaciones conectadas directamente;
 - f. encontrar el camino más corto desde:
 - I. la estación King's Cross hasta la estación Waterloo,
 - II. la estación Victoria Train Station hasta la estación Liverpool Street Station,
 - III. la estación St. Pancras hasta la estación King's Cross;
 - g. encontrar el árbol de expansión mínimo del ramal del tren.
20. Se dispone del acervo bibliográfico de tesis de carreras de una facultad de informática. Implementar los algoritmos necesarios para satisfacer los requerimientos que nos solicitan:
- a. cada vértice tendrá el nombre de la tesis, el autor de la misma y cuatro palabras claves;
 - b. las aristas representan la relación entre dos tesis cuyo contenido es similar, para lo cual consideramos como mínimo dos palabras claves en común –se deben cargar al menos tres aristas por vértice–;
 - c. realizar un barrido en profundidad desde una tesis que contenga la palabra clave “programación” o “inteligencia artificial”, considerar solamente las tesis que son accesibles desde el inicio sin reiniciar el barrido (si quedaron sin tratar);
 - d. obtener el árbol de expansión máximo partiendo desde un vértice que tenga la palabra clave “minería de datos”.

Técnicas de diseño de algoritmos, intentando desarrollar algoritmos eficientes

En la ciencia de la computación se han identificado diversas técnicas generales que producen algoritmos eficientes para la resolución de muchas clases de problemas, entre estas podemos destacar: voraces o ávidas, divide y vencerás, programación dinámica, programación paralela, ramificación y poda, búsqueda con retroceso, entre otros. A continuación se analizarán en detalle las primeras tres de estas y se describirán de manera general el resto. Se debe tener en cuenta que cada una de estas técnicas es eficiente para resolver determinados problemas. Por lo tanto, el tipo o clase del problema es fundamental para la elección correcta de la técnica a utilizar.

Existen esencialmente tres tipos o clases de problemas de computación genéricos: problemas de optimización, de búsqueda y de decisión. Básicamente se hará enfoque en el uso de técnicas eficientes para el diseño de algoritmos genéricos que sirvan como métodos generales para abordar y solucionar estos tipos de problemas.

Devorando nuestros problemas con algoritmos voraces

Esta es una de las técnicas más simples y a su vez muy utilizada, normalmente es usada para resolver problemas de optimización –estos pueden ser de minimización o de maximización–, también denominados método codicioso. Se puede considerar como esencia de esta técnica que recibe como entrada un conjunto de candidatos –es decir un conjunto de elementos que podrían formar parte de la solución– a partir de los cuales, un subconjunto de estos satisfacen determinadas condiciones o restricciones que los convierten en una *solución factible*¹ al problema. El objetivo del algoritmo es encontrar la solución factible óptima que minimice o maximice una determinada función objetivo, denominada *solución óptima*². Es importante destacar que en muchos casos no siempre el enfoque voraz llegará a dar la solución óptima, este puede dar una buena solución factible, pero el resultado general puede ser pobre –esto dependerá en gran parte de como se diseñe del algoritmo–.

¹ Solución factible: son aquellos subconjuntos de elementos que cumplen todas las condiciones para ser solución del problema.

² Solución óptima: es aquel subconjunto de elementos que permite obtener el valor mínimo o máximo de una determinada función objetivo de un problema de optimización.

A lo largo del libro ya hemos visto y analizado algunos algoritmos voraces, por ejemplos en el capítulo X se describe el algoritmo voraz para generar códigos de Huffman que permite encontrar la forma óptima de representar caracteres con códigos binarios de acuerdo a su frecuencia de aparición. Luego, en el capítulo XIII, se presentaron tres algoritmos ávidos: el primero el algoritmo de Dijkstra que encuentra el camino mínimo desde un vértice origen a un vértice destino de un grafo con aristas de peso positivo. Los otros dos algoritmos, Prim y Kruskal, obtienen el árbol de expansión mínimo de un grafo no dirigido y conexo, todos estos se basan en el uso de estrategias voraces distintas.

La mecánica de funcionamiento de los algoritmos voraces es en pasos. En cada paso se toma una decisión que parece ser buena sin considerar las consecuencias futuras –es decir se selecciona el que se considera en ese momento el mejor candidato del conjunto de candidatos–, esto significa que se elige un óptimo local. Cuando el algoritmo termina de ejecutarse, se espera que el óptimo local sea igual al óptimo global, de no ser así habremos hallado una solución subóptima –es decir una solución factible–. El nombre voraz proviene de la estrategia de “tomar lo que sea mejor ahora” en cada etapa. En la figura 1 podemos observar el esquema general de un algoritmo ávido:

```
def voraz(conjunto_candidatos):
    """Esquema de algoritmo voraz."""
    solucion = []
    while(not conjunto_candidatos and not es_solucion(solucion)):
        x = seleccionar_mejor(conjunto_candidatos)
        if(es_factible(x, solucion)):
            solucion.append(x)
    if(es_solucion(solucion)):
        return solucion
    else:
        return None
```

Figura 1. Esquema general de un algoritmo ávido

Como se observa en la figura anterior un algoritmo voraz es fácil de implementar y cuando están bien diseñados son eficientes –y suelen producir una solución óptima–. Sin embargo hay muchos problemas que no se pueden resolver con este enfoque. Básicamente estos algoritmos están compuestos por los siguientes elementos:

Conjunto de candidatos que almacena todos los elementos que pueden formar parte de la solución;

Conjunto solución, inicialmente está vacío y en cada paso del algoritmo se intenta agregar “el mejor elemento del conjunto de candidatos” basado en alguna función de optimización denominada función de selección;

Función de selección del mejor candidato, se encarga de determinar cuál es el elemento más prometedor del conjunto de candidatos en cada etapa del algoritmo;

Función de factibilidad, se utiliza para determinar si un elemento candidato –elegido por la función de selección– es factible para formar parte de la solución. Si es factible dicho elemento, se agrega al conjunto solución y permanecerá siempre en él, caso contrario es descartado y no vuelve a ser considerado;

Función solución, su objetivo es determinar si los elementos en el conjunto solución son una solución factible del problema o no, independientemente de que sea la óptima o no.

Cabe destacar que el aspecto clave a tener en cuenta al momento del diseño de un algoritmo ávido es diseñar una buena función de selección. Esto será crucial para garantizar que el algoritmo obtenga la solución óptima al problema en lugar de una solución subóptima.

Suponga que debemos solucionar *el problema del cambio*, el cual consiste en devolver un monto en algún sistema monetario utilizando el menor número de monedas posibles, para lo cual partimos de un conjunto de monedas validas –euros en este caso–, las cuales se observan en la figura 2, las cuales se supone que hay la cantidad suficiente para devolver el importe requerido y el valor de dicho importe.

```
from tda_algoritmos import cambio

monedas = [0.01, 0.05, 0.1, 0.25, 0.5, 1, 2]
vuelto = cambio(monedas, 3.81)
print(vuelto)
```

Figura 2. Conjunto de monedas validas

De acuerdo a los conceptos descritos anteriormente en las figuras 3 y 4, se presenta la implementación de la función “es solución” y el algoritmo voraz para solucionar el “problema del cambio” respectivamente.

```
def es_solucion(solucion, valor_devolver):
    """Determina si el conjunto solucion, es una solucion al problema."""
    total = 0
    for moneda in solucion:
        total = round(total + (moneda[0] * moneda[1]), 2)
    if(total == valor_devolver):
        return True
    else:
        return False
```

Figura 3. Función es solución para el algoritmo voraz del problema cambio

```

def cambio(conjunto_candidatos, valor_devolver):
    """Algoritmo voraz para resolver el problema del cambio."""
    solucion = []
    restante = valor_devolver
    while conjunto_candidatos and not es_solucion(solucion, valor_devolver):
        dato = conjunto_candidatos.pop()
        if(dato <= restante):
            cantidad = restante // dato
            solucion.append([dato, cantidad])
            restante = round(restante - (dato * cantidad), 2)
    if(es_solucion(solucion, valor_devolver)):
        return solucion
    else:
        return None

```

Figura 4. Algoritmo voraz para solucionar el cambio

Como se puede observar el conjunto solución inicialmente está vacío, luego mientras haya elementos en el conjunto de candidatos y no se llegue a una solución se procede con el algoritmo. Primero se selecciona el mejor candidato –para nuestro caso la moneda de mayor valor– y se verifica si puede formar parte de la solución. Para esto se chequea si el valor de dicha moneda es menor que el importe a devolver; si es así se determina cuántas monedas de ese monto se pueden utilizar y se agrega al conjunto solución. Y continuamos repitiendo este procedimiento hasta que no se cumpla una de las condiciones mencionadas previamente, finalmente evaluamos si el conjunto solución es una solución factible, de ser así devolvemos el conjunto solución en caso contrario el algoritmo devolverá *None*. La función solución se encarga de evaluar si la suma de los elementos almacenados en el conjunto soluciones es igual al importe del monto a devolver.

El algoritmo codicioso del cambio produce una solución óptima general debido a las propiedades de las monedas, siempre y cuando existan los valores de monedas que permitan devolver cualquier valor –no todos los sistemas monetarios lo tienen-. Ahora, *¿qué pasa si eliminamos las monedas de 1 y 2 euros de nuestro sistema monetario y agregamos una de valor 0.75?* Si se debe dar como vuelto 1 euro, la solución de nuestro algoritmo voraz es una moneda de 0.75, una de 0.2, y una de 0.05. Pero, *¿esa es la solución óptima?* Como se habrán dado cuenta no lo es, la solución óptima es dos monedas de 0.5. Por eso es que el algoritmo que se diseñe también dependerá del problema y de las características de los elementos que forman parte del conjunto de candidatos.

Ahora veamos otro ejemplo, supongamos que debemos resolver el problema de la mochila: el mismo consiste en llenar una mochila que soporta un peso máximo con un conjunto de objetos, cada uno de estos con un peso y un beneficio; los objetos colocados en la mochila deben maximizar el beneficio obtenido sin exceder el límite de peso que soporta la mochila. Este problema tiene su origen en los primeros trabajos del matemático Tobias Dantzig, quien hace referencia al problema de empacar los artículos más valiosos o útiles sin sobrecargar el equipaje.

Formalmente se puede definir el problema de la siguiente manera: se dispone de n objetos de los cuales supondremos cantidades x infinitas –inicialmente para simplificar el problema–, cada uno de dicho objetos tiene un peso p y un beneficio b , la mochila además tiene una capacidad máxima $cmax$. El objetivo es llenar la mochila maximizando el beneficio total obtenidos de los objetos almacenados, sin exceder la capacidad de la mochila. Por lo tanto, el problema se puede expresar matemáticamente de la siguiente manera:

$$\begin{aligned}
 & \text{maximizar } Z(x) = \sum_{i=1}^n b_i x_i \\
 & \text{sujeto a que } P(x) = \sum_{i=1}^n p_i x_i \leq c_{\max} \\
 & \text{con } 0 \leq x_i \leq q_i, 1 \leq q_i \leq \infty
 \end{aligned}$$

Siguiendo con el ejemplo de la mochila, en la figura 5 y 6 se describe la implementación de la función “es solución” y el algoritmo voraz para resolver el problema de la mochila, para este caso solo hay una unidad de cada elemento, no son infinitos ni fraccionables. Como pueden observar las funciones son muy similares a las utilizadas en el problema del cambio dado que la estrategia es la misma.

```

def es_solucion(solucion, capacidad):
    """Determina si el conjunto solucion, es una solucion al problema."""
    total = 0
    for elemento in solucion:
        total = round(total + (elemento[1]), 2)
    if(total == capacidad):
        return True
    else:
        return False

```

Figura 5. Función es solución para el algoritmo ávido de la mochila

```

def mochila(conjunto_candidatos, capacidad):
    """Algoritmo voraz para reolver el problema de la mochila."""
    solucion = []
    restante = capacidad
    while conjunto_candidatos and not es_solucion(solucion, capacidad):
        dato = conjunto_candidatos.pop()
        if(dato[1] <= restante):
            solucion.append(dato)
            restante = round(restante - dato[1], 2)
    return solucion

```

Figura 6. Algoritmo ávido para el problema de la mochila

Como ya se mencionó previamente la función que selecciona el mejor candidato del conjunto de soluciones es quizás el punto crítico para que el algoritmo logre encontrar una solución óptima, para este caso particular de los elementos almacenados en el conjunto de candidatos se conoce: nombre, peso y beneficio. Entonces lo que podemos hacer es agregar un nuevo valor, que es beneficio dividido el peso del elemento, para poder obtener una medida del beneficio del elemento respecto de su peso. Luego ordenamos el vector de candidatos de menor a mayor tomando como criterio de orden este nuevo valor –es decir que al final del vector estarán los elementos con mejor

relación *beneficio-peso*. Esto le permite al algoritmo en cada iteración tomar el elemento con mejor relación *beneficio-peso* para agregarlo al conjunto solución, siempre y cuando no se exceda la capacidad máxima de la mochila. Cabe aclarar que a diferencia del caso anterior puede que no se utilice toda la capacidad de la mochila, es decir, que seguramente quedará un sobrante de capacidad disponible dado que los elementos son fraccionables.

Para este caso en particular el problema a resolver es poner en el portatermo (mochila) los elementos que nos brinden mayor beneficio. La implementación de lo mencionado anteriormente se puede apreciar en la figura 7.

```
from tda_algoritmos import mochila, beneficio

objetos = [['termo', 1.1, 20], ['mate', 0.3, 6], ['galletita', 0.5, 7],
           ['té', 0.09, 0.7], ['yerba', 0.3, 7], ['bombilla', 0.33, 6],
           ['azúcar', 0.5, 2], ['alfajor', 0.6, 13], ['facturas', 0.8, 6]]
porta_termo = []
cap_max = 3

for elemento in objetos:
    elemento.append(round(elemento[2]/elemento[1]))

objetos.sort(key=beneficio)
porta_termo = mochila(objetos, cap_max)
print(porta_termo)
```

Figura 7. Conjunto de elementos válidos y criterio de orden

En resumen, el esquema voraz es una estrategia heurística que no siempre conduce a la solución óptima, es muy útil cuando la solución óptima general puede construirse a partir de la selección de óptimos locales que se toman en cada paso sin depender de las futuras selecciones. Podríamos decir que esta estrategia voraz es del tipo *top-down* tomando una decisión voraz tras otra –es decir comiendo la mejor parte en cada etapa– reduciendo iterativamente el problema.

Divide y vencerás, también aplica al diseño de algoritmos

Esta es quizás una de las técnicas más importantes y utilizadas para el diseño de algoritmos eficientes, la cual consiste en descomponer un problema de tamaño n en subproblemas más pequeños del mismo problema –hasta que el problema se vuelva muy sencillo y la solución sea muy obvia–. Luego se resuelve cada subproblema de manera independiente de los demás y a partir de la solución de cada uno de estos, se combinan para construir la solución al problema completo. Cabe aclarar que esta técnica se aplica de manera recursiva o mediante el uso de pilas para sustituir las llamadas recursivas.

Esta técnica tiene varias aplicaciones en la vida real, no solo aplicada en el diseño de algoritmos eficientes sino también en el diseño de circuitos, para hacer demostraciones matemáticas, como estrategia militar y muchos otros campos. De hecho ya hemos utilizado varios algoritmos que utilizan esta técnica en este libro, por ejemplo en el capítulo IV se describió el algoritmo de búsqueda binaria que en cada iteración partitiona el vector de datos en dos y reduce así el espacio de búsqueda; a su vez en dicho capítulo también se analizaron dos algoritmos de ordenamiento basados en esta técnica: ordenamiento mezcal (*merge sort*) y ordenamiento rápido (*quicksort*), los cuales utilizan esta técnica pero de diferentes maneras.

El funcionamiento de esta técnica se divide en dos partes: la primera es “*dividir*” en subproblemas el problema general los cuales se resuelven de manera recursiva –a excepción del caso base del cual ya se conoce la solución–, y la segunda es “*vencer*” –es decir combinar las soluciones parciales– en la cual se construye la solución general a partir de las soluciones a los subproblemas. El nombre divide y vencerás proviene del hecho de que es más fácil y eficiente resolver muchos problemas sencillos y combinarlos para resolver un problema complejo que resolver dicho problema completo. A continuación en la figura 8 se describe el esquema general de un algoritmo divide y vencerás.

```
def divide_y_venceras(datos_problema):
    """Esquema de algoritmo divide y vencerás."""
    solucion = None
    if(es_caso_base(datos_problema)):
        return solucion
    else:
        particiones = descomponer(datos_problema)
        soluciones = []
        for datos_subproblema in particiones:
            soluciones.append(divide_y_venceras(datos_subproblema))
        solucion = combinar_soluciones(soluciones)
    return solucion
```

Figura 8. Esquema general de un algoritmo divide y vencerás

Como podemos observar en la figura anterior un algoritmo de tipo divide y vencerás debe constar de las siguientes partes:

Caso base, es decir la condición que indicará que ya no se deben realizar más divisiones del problema, porque se ha llegado a un caso simple cuya solución se conoce;

Descomponer el problema, es decir realizar las particiones del problema en subproblemas dependiendo del funcionamiento del algoritmo y del problema;

Resolver los subproblemas, es decir las llamadas recursivas al mismo algoritmo divide y vencerás, actualizando los parámetros de entrada de dicho algoritmo de acuerdo a las particiones realizadas en el punto anterior;

Combinar las soluciones de los subproblemas, es decir construir la solución al problema en base a los resultados obtenidos luego de resolver los subproblemas.

Por lo general se dice que los algoritmos que tiene al menos dos llamadas recursivas son de tipo divide y vencerás dado que supone que ambos subproblemas suelen ser independientes, no así los que solo tienen una llamada recursiva.

Por ejemplo, en el caso de tener que multiplicar dos números enteros, como ya lo describimos en el capítulo III, se lo puede considerar como una operación elemental, siempre y cuando los números a multiplicar entran en una variable de la computadora –a nivel de representación interna de hardware–. En caso contrario, cuando los enteros son grandes el coste de la multiplicación de dos números enteros es del orden de $O(n * m)$, donde n y m representan la cantidad de dígitos de los dos números a multiplicar, entonces por ejemplo si tenemos que multiplicar $1\ 257 * 34\ 908$ el coste computacional de esta operación es $O(20)$, lo cual es muy distinto del coste $O(1)$ de una operación elemental.

Para resolver problema multiplicación de enteros grandes existe una manera más eficiente de hacerlo que la anterior denominada algoritmo de Karatsuba que fue descubierto por Anatolii Karatsuba en 1960 y publicado un par de años más tarde en 1962³. Este algoritmo utiliza la técnica de divide y vencerás para resolver el problema de la multiplicación de enteros.

Por ejemplo suponga que tiene dos números enteros de la misma cantidad de dígitos num1 y num2 –funciona con número con distinta cantidad de dígitos también–, se los descompone en mitades con la misma cantidad de dígitos, por lo tanto obtendremos lo siguiente:

$$\begin{cases} \text{num1} = 10^s w + x \\ \text{num2} = 10^s y + z \\ \text{con } s = \frac{n}{2} \end{cases}$$

Para finalmente obtener que el resultado de multiplicación de ambos números enteros es:

$$\text{producto} = 10^{2s}wy + 10^s(wz + xy) + xz$$

Si resolvemos esto dentro del algoritmo, el coste de algoritmo seguirá siendo del orden de $O(n^2)$. Para lograr una eficiencia se debe calcular wy , $(wz + xy)$ e xz en tan solo tres multiplicaciones entonces sí:

$$\begin{aligned} r &= (w + x)(y + z) = wy + wz + xy + xz \\ r &= (w + x)(y + z) - wy - xz = (wz + xy) \end{aligned}$$

Y de esta manera se evita tener que hacer las dos multiplicaciones wz e xy , y en su lugar solo se realiza la multiplicación $(w + x)(y + z)$ más algunas sumas adicionales. Entonces se obtiene que el algoritmo está en el orden de $O(n^{\log 3}) \simeq O(n^{1.585})$. Si bien no aparenta haber mucha diferencia significativa

³ A. Karatsuba and Yu. Ofman (1962). «Multiplication of Many-Digital Numbers by Automatic Computers». Translation in Physics-Doklady, 7 (1963), pp. 595–596. Proceedings of the USSR Academy of Sciences 145: 293–294.

o interesante entre dicho orden y el del método clásico $O(n^2)$ se vuelve interesante cuando se debe multiplicar enteros grandes, por ejemplo si los números son de 700 dígitos: con la multiplicación de Karatsuba el coste es 32 320 mientras que con el método clásico el coste es 490 000, si son números de 1 000 dígitos los órdenes serán 56 885 y 1 000 000 para los mismos algoritmo respectivamente y aquí si es relevante el uso de un método u otro.

A continuación en la figura 9 se presenta la implementación del algoritmo de Karatsuba, nótese que solo se realizan las tres multiplicaciones descritas anteriormente para mejorar la eficiencia del algoritmo, este realiza las llamadas recursivas con los números particionados. En el mismo se puede configurar cuál es el tamaño de partición mínimo, es decir, cuándo se considera un caso simple y se procede a resolver la multiplicación de los números. Para este caso particular el valor es uno. Finalmente devuelve el valor de las multiplicaciones y los va sumando para obtener la solución final.

```
def multiplicacion_enteros(num1, num2):
    """Algoritmo divide y vencerás multiplicación de enteros grandes."""
    n = len(str(num1)) if len(str(num1)) > len(str(num2)) else len(str(num2))
    if n == 1:
        return num1 * num2
    else:
        s = n // 2
        w = num1 // 10**s
        x = num1 % 10**s
        y = num2 // 10**s
        z = num2 % 10**s
        r = multiplicacion_enteros(w+x, y+z)
        wy = multiplicacion_enteros(w, y)
        xz = multiplicacion_enteros(x, z)
        return (wy * (10**(2*s))) + ((r-wy-xz) * (10**s)) + xz
```

Figura 9. Algoritmo de Karatsuba

Menos iteraciones y más programación dinámica para reducir la complejidad

Es un método de optimización matemático y además una técnica de programación que fue creada por Richard Bellman⁴ en 1957. Se utiliza típicamente para resolver problemas de optimización resolviendo los problemas mediante secuencia de decisiones –al igual que los algoritmos voraces– con la diferencia de que se producen varias secuencias de decisiones y solamente al finalizar se sabe cuál es la mejor. Descompone un problema en pequeños subproblemas del mismo tipo –al igual que la técnica divide y vencerás– diferenciándose de estos en que los subproblemas están superpuestos entre sí, los cuales comparten soluciones que se calculan una vez luego se almacenan –dicha capacidad se denomina memorizar– y se reutilizan por los subproblemas que lo vuelven a necesitar. Esta técnica se basa en el principio de optimalidad de Bellman que dice lo siguiente “dada cualquier

⁴ Bellman, Richard (1957), Dynamic Programming, Princeton University Press. Dover paperback edition (2003), ISBN 0-486-42809-5.

secuencia de decisiones óptima, toda subsecuencia de decisiones de ella es óptima también". Esta técnica permite evitar explorar todas las secuencias de soluciones de un problema mediante la resolución de subproblemas de tamaño creciente memorizando los resultados en una tabla para facilitar cálculos posteriores que requieran dicho valor.

Avanzando y volviendo para atrás: búsqueda con retroceso

Se utiliza para encontrar la mejor solución de un conjunto de soluciones que cumplen determinadas condiciones, permite realizar una búsqueda sistemática a través de todas las soluciones posibles conocidas como espacio de solución que se representa mediante un árbol. Cada solución resulta de una secuencia de decisiones y además existe una función objetivo que se debe satisfacer por dicha solución, el objetivo principal de esta técnica es encontrar la solución que optimiza dicha función objetivo –ya sea que maximice o minimice, dependiendo del tipo de problema–. Este tipo de técnica se aplica a problemas de optimización en los cuales se cumple el principio de optimalidad de Bellman y en los que no queda otra forma de resolverlos más que buscar en el espacio de soluciones.

Explorar el espacio de soluciones por fuerza bruta por lo general resulta ineficiente, dado que dicho espacio suele ser muy grande. La manera de mejorar esto y alcanzar antes una solución es mediante el uso de la técnica de búsqueda con retroceso o vuelta atrás, mediante la cual la solución se construye de manera progresiva comprobando que para cada nodo interno del árbol –es decir cada elemento agregado a la solución– nos pueda llevar a una solución satisfactoria. Caso contrario, se realiza una vuelta atrás en el árbol de soluciones y se continua por otra rama, lo que nos permite evitar recorrer un subárbol cuyas soluciones no satisfacen las condiciones de la función objetivo.

Entonces, *¿Qué diferencia tiene este tipo de técnica respecto de las vistas anteriormente?* Las técnicas voraces construyen una solución tomando una decisión en cada paso, los elementos que son agregados a la solución siempre permanecen en esta y se descartan los que en ese momento no resultan factible. Por su parte en la búsqueda con retroceso la elección de un candidato en una etapa no es definitiva es decir que puede ser descartado si se encuentra una mejor alternativa realizando una vuelta atrás. El tipo de problemas que se abordarán con la técnica de búsqueda con retroceso no pueden ser descompuestos en subproblemas del mismo tipo, por lo cual no resulta factible aplicar técnicas con enfoque de divide y vencerás.

Expandiendo ramas útiles del árbol y podando las innecesarias con ramificación y poda

Esta técnica es utilizada para la resolución de problemas de optimización discretos y en problemas de juegos –es decir de tipo búsqueda–, también se lo suele considerar como una forma mejorada de la técnica de búsqueda con retroceso. La similitud con dicha técnica es que ambas generan un espacio de solución sobre un árbol que luego recorren sistemáticamente. Las principales diferencias respecto a la búsqueda con retroceso son:

Ramificación: el recorrido del árbol no es solo en profundidad, también puede ser en amplitud;

Poda: se realiza en base a estimaciones de cotas inferiores CI, superiores CS y de beneficio estimado BE antes de explorar cada nodo del árbol, a partir de estas cotas que deben ser fiables podemos determinar cuándo se debe realizar una poda y descartar dicho subárbol lo cual nos permitirá reducir significativamente el espacio de búsqueda.

En términos generales la estrategia del algoritmo es la siguiente para cada nodo i:

Ramificación: se hace un recorrido del árbol de soluciones –ya sea en profundidad, o en amplitud o según beneficio estimado–. Para hacer estos recorridos se utiliza una lista de nodos vivos –es decir aquellos que no son podados–; para determinar el siguiente nodo a explorar de la lista de nodos vivos se utilizan distintos enfoques pila (*lifo* en profundidad), cola (*fifo* en amplitud), o estrategias menor costo (*least cost*) que seleccionan el siguiente nodo que tenga mayor beneficio o menor costo pila (*lc-lifo*), cola (*lc-fifo*). Las dos primeras son búsquedas a ciegas o por fuerza bruta, mientras que en las segundas se tiene en cuenta el beneficio mediante una función de estimación. En resumen, la mejor estrategia es ramificar a partir de la cota BE(*i*) de cada nodo es decir el mejor de los nodos vivos, independientemente de que sea por profundidad o amplitud.

Poda: a partir de las cotas CI(*i*) y CS(*i*) se determina qué subárboles se deben podar, dado que no tiene sentido expandirlos si la soluciones en dichos subárboles no serán óptimas.

Números, operaciones aritméticas y más números: algoritmos numéricos

Un algoritmo numérico es un conjunto de instrucciones ordenadas para resolver un problema que involucra procesos matemáticos –con cálculo de fórmulas de manera repetida–. Este tipo de algoritmos no admiten ambigüedades y deben darse cada uno de los pasos para llegar a su solución. Este tipo de algoritmos suele tener un problema con la representación de los decimales dependiendo del lenguaje sobre el cual se implemente. Al representarse de distintas maneras los resultados nunca serán los mismos, este fenómeno se conoce como propagación de errores, dado que estas diferencias de decimales se propaga y acumula a través de las distintas iteraciones del algoritmo hasta obtener la solución. En algunas situaciones las diferencias son significativas y el porcentaje de error es muy importante.

Guía de ejercicios prácticos

A continuación se plantean una serie de problemas. Para resolverlos se deberá desarrollar un algoritmo utilizando algunas de las técnicas vistas.

1. Resuelva el problema de dar los siguientes cambios de monedas diseñando un algoritmo voraz:
 - a. el costo es 4.01 y se paga con 10,
 - b. el costo es 10.75 y se paga con 20,
 - c. el costo es 0.93 y se paga con 5.

Para ello, considerar los siguientes sistemas de monedas:

- d. monedas griegas: 0.01, 0.02, 0.05, 0.10, 0.20, 0.50, 1.00, 2.00 euros;
- e. monedas japonesas: 1, 5, 10, 50, 100, 500 yen;
- f. monedas rusas: 0.01, 0.05, 0.1, 0.5, 1, 2, 5, 10 rublo;
- g. monedas tailandesas: 0.01, 0.05, 0.1, 0.25, 0.5, 1, 2, 5 baht.

Responda las siguientes preguntas:

- h. ¿el mismo algoritmo funciona para todos los sistemas monetarios?;
 - i. ¿todos los sistemas monetarios tiene solución?;
 - j. ¿si se obtiene una solución siempre es óptima?
2. Implementar un algoritmo que permita determinar qué elementos debe llevar un Jedi en su mochila de manera que se optimice el beneficio, con las siguientes consideraciones:
 - a. los elementos tienen una cantidad máxima y no son fraccionables;
 - b. la mochila tiene una capacidad de 27 kilos;

c. los elementos son los siguientes:

Elemento	Peso	Beneficio	Cantidad
frutas	0.73	50	8
carpa	3	90	2
termo stanley	1.5	75	3
sable de luz	1.3	175	1
mapa holograma	1.1	93	1
traje Jedi	0.9	87	4
bolsa de créditos galácticos	5	100	3
lata de alimento	0.79	75	10
galletitas	1	60	15
yerba canarias	0.64	70	7
pan	2.5	65	5
botella de agua	1.9	99	5
soga	2.3	40	3
mate	0.8	75	6
blaster	8.3	85	2

3. Desarrollar un algoritmo que indique los alimentos que se deben llevar para un campamento, suponiendo que todos los alimentos son fraccionables y la capacidad de la mochila es de 21 kilos –no hay límite respecto de las cantidades de cada uno–, pero además se debe tener en cuenta la duración de dichos alimentos antes de ponerse en mal estado –considere 30 días como máximo–:

Alimento	Peso	Beneficio	Duración
arroz	5	250	30
manzana	3	231	19
caramelo	15	20	20
papa	6	215	23
galletita	10	50	30
fideo	7	175	30
cereales	13	150	15
zanahoria	4	245	20
choclo	7	166	27
queso	8	207	10
helado	9	90	0
lechuga	10	188	5
pollo	6	235	1
aceituna	17	117	11
pan	4	220	7

fiambre	15	124	1
tomate	9	193	6
carne	10	190	1
atún	9	230	1

4. Implemente un algoritmo para resolver el problema de la mochila en el cual, además de considerar el beneficio y el peso de cada elemento, se considere la cantidad de espacios que ocupa cada uno de estos. Entonces la mochila además de tener un peso máximo tendrá una cantidad máxima de lugares –considere que los objetos no son fraccionables para simplificar el problema–.
5. Resuelva el problema de colocar las n-reinas sin que las mismas se amenacen, para tableros de las siguientes dimensiones –solo es necesario encontrar una de las soluciones–:
 - a. 4 reinas tablero de 4 x 4;
 - b. 8 reinas tablero de 8 x 8;
 - c. 16 reinas tablero de 16 x 16.
6. Desarrollar un algoritmo del tipo divide y vencerás, que permita resolver el producto matrices de [2 x 2] y [3 x 3]. Este algoritmo debe ser del orden de $O(n^2)$ por lo cual debe evitar el tercer ciclo, sabiendo que las matrices solo pueden ser de las dimensiones indicadas.
7. Implementar un algoritmo para la multiplicación de enteros grandes que permita configurar el tamaño mínimo de la partición de los números enteros –utilizando la técnica divide y vencerás–. El algoritmo debe aceptar números de distinto largo, pruebe con los siguientes tamaños de partición si el algoritmo funciona: 2, 3, 5, 7.
8. Resuelva el problema de la torre de Hanói –ejercicio 23 de la guía de recursividad– de manera iterativa utilizando una de las técnicas vistas –a excepción de divide y vencerás cuyo mecánica es recursiva–.
9. Implementar un algoritmo del tipo divide y vencerás, que permita calcular la potencia de un número n elevado a la k , es decir (n^k) de manera eficiente.
10. Implementar un algoritmo que permita elaborar el calendario de los torneos de básquet y vóley de todos los equipos que asistieron al campus de verano, teniendo en cuenta las siguientes cuestiones:
 - a. durante el torneo todos los equipos deben enfrentarse contra todos lo demás;
 - b. cada equipo jugara un partida por día;
 - c. Los equipos de básquet son 10 y los de vóley 7.

- II. Implementar un algoritmo que permita resolver el problema del turista, el cual consiste en visitar todas las ciudades una sola vez y volver a la ciudad de partida recorriendo la menor distancia posible. Contemple lo siguiente:
- las ciudades son: Concepción del Uruguay (Argentina), Rodas (Grecia), Tulum (México), Phuket (Tailandia), Tokio (Japón), Atenas (Grecia), Orlando (Estados Unidos), Santiago (Chile), Moscú (Rusia), Bombinhas (Brasil), Londres (Reino Unido), Paris (Francia), Sídney (Australia), Berlín (Alemania);
 - debe considerar las distancias reales en kilómetros;
 - ciudad de salida Concepción del Uruguay;
 - ahora además de considerar la menor distancia, mejore el algoritmo para que también considere el menor costo.
12. Nick Fury se encuentra en los cuarteles generales de S.H.I.E.L.D. y debe visitar a varios superhéroes para convencerlos de unirse para formar un grupo de vengadores, dado que es un asunto de suma importancia nos solicita implementar un algoritmo que permita determinar el recorrido de menor distancia –el menor posible, no importa que sea el más óptimo– y terminar dicho recorrido de vuelta en los cuarteles (solo se puede pasar una vez por cada lugar).
- Considere los siguientes superhéroes: S.H.I.E.L.D. (a), Tony Stark (b), Natasha Romanoff (c), Steve Rogers (d), Carol Danvers (e), Bruce Banner (f), Scott Lang (g), Peter Quill (h);
 - las distancias entre la localización de cada superhéroe está cargada en la siguiente matriz:
- | | a | b | c | d | e | f | g | h |
|----------|-----|-----|-----|-----|-----|-----|-----|-----|
| a | | 675 | 400 | 166 | 809 | 720 | 399 | 233 |
| b | 675 | | 540 | 687 | 179 | 348 | 199 | 401 |
| c | 400 | 540 | | 107 | 752 | 521 | 385 | 280 |
| d | 166 | 687 | 107 | | III | 540 | 990 | 361 |
| e | 809 | 179 | 752 | III | | 206 | 412 | 576 |
| f | 720 | 348 | 521 | 540 | 206 | | 155 | 621 |
| g | 399 | 199 | 385 | 990 | 412 | 155 | | 100 |
| h | 233 | 401 | 280 | 361 | 576 | 621 | 100 | |
13. Resuelva el problema del laberinto –ejercicio 22 de la guía de recursividad– de manera iterativa, utilice la técnica que crea más conveniente.
14. Desarrollar un algoritmo numérico iterativo que permita calcular el método de la bisección de una función $f(x)$.
15. Desarrollar un algoritmo numérico iterativo que permita calcular el método de la secante de una función $f(x)$.

16. Desarrollar un algoritmo numérico iterativo que permita calcular el método de Newton-Raphson de una función $f(x)$.
17. Comparar los tres algoritmos anteriores para resolver la siguiente función: $x^3 + x + 16 = 0$, respecto de la cantidad de iteraciones necesarias por cada método para converger. ¿Cuánto es la diferencia en decimales entre las distintas soluciones?

Método	Cantidad de iteraciones	Solución
Secante		
Bisección		
Newton-Raphson		