#### 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 desarro- llar 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 resol- ver 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 clasifi- carla 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 calcu- lar 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!  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:

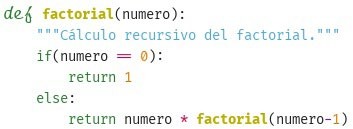


Figura 2. Implementación de función recursiva en Python

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 resul- tado, 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 des- borde 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.



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 parti- cular 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 proble- mas más pequeños sin alterar la naturaleza del problema y su solución. Sin embargo, esto no impli- ca que esta sea la mejor solución. Estos problemas también pueden ser resueltos de manera iterati- va, 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 cam- bio, el modelo iterativo solo requiere de un ciclo formado por una variable de control, más las va- riables 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.

#### 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 re- suelve. 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 teó- ricas. 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 necesa- rios 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 asignatura 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 progra- mador (persona). Además se ahorra el tiempo que se habría invertido en el desarrollo de un algorit- mo 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 ejem- plar 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)<= *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 va- lores 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*(n2), o de tiempo cuadrático;
6. en el orden de *O*(n3), o de tiempo cúbico;
7. en el orden de *O*(nk), o de tiempo polinómico;
8. en el orden de *O*(cn), o de tiempo exponencial;
9. en el orden de *O*(n!), o de tiempo factorial.

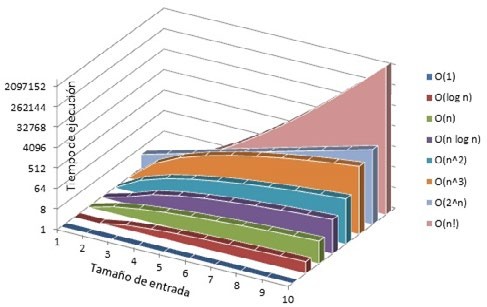


Figura 1. Gráfica de comparación de órdenes de crecimiento de algoritmos

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*(n2), 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 ocu- rran (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 asignatura 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 re- queridos 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 fun- ciones en el límite, es decir para valores significativamente grandes de sus parámetros. Esta nota- ción 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 re- quieren un tiempo de *O*(n), *O*(n3) y *O*(n log n), por lo tanto el algoritmo requiere un tiempo de *O*(n + n3 + n log n).

Pero aunque el tiempo requerido por el algoritmo es lógicamente la suma de los tiempos requeri- dos 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 + n3 + n log n) = *O*(*máximo*(n, n3, n log n)) = *O*(n3)

## 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 1 (asignación, en- trada/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*(1).

**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 *F1* de coste 𝛼 y *F2* de coste 𝛽 que forman un programa, el coste de dicho programa será la suma de ambos fragmentos *O*(𝛼 + 𝛽).

**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* + *máximo*(rama verdadera, rama falsa)).

**Ciclo *for*:** en los ciclos con contador explícito (ciclo desde-hasta) que encierran un fragmento de código *F1* 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 *F1*, 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 *F1* 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*(n2).

**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 estructu- ras 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 ca- sos 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 regula- ridad para realizar el cálculo del coste. Estas técnicas antes mencionadas son avanzadas y exceden los alcances de este asignatura 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 fiegura 2, dicho algoritmo es iterativo.



Figura 2. Código de algoritmo “numeros\_pares\_impares”

El análisis de un algoritmo se realiza desde adentro hacia afuera, entonces lo primero que determi- namos es el orden del condicional (*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 *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 1 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 algo- ritmo (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*: N � 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, Fibo- nacci 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 opera- ción 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*(n0) 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

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

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

*=* 22*T*(n-2) + 22 c

= 2k*T*(n-k) + 2k c

Cuando k = n-1 tenemos que

*T*(n) = 2n-1*T*(0) + 2n-1 c

= 1 + (n-1)

= 2n-1

Entonces podemos establecer que Fibonacci recursivo es del orden de *T*(n) ≅ 2n-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/2i) + c con *T*(n<=1) = 1 y la variable *i* que toma valores de 1 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/2k) + 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:

1 = n/2k

1. 2k = n 2k = n

log2 2k = log n k log2 2 = log n k 1 = log n

k = log n

Como la ecuación n/2k = 1 se resuelve en tiempo k = log2 n, nos queda que

*T*(n) = *T*(n/2k) + kc

= *T*(1) + log2 n

Entonces de esta manera obtenemos que la función búsqueda binaria recursiva es del orden de

*T*(n) ≅ log n, dado que si n no es múltiplo de dos no obtendremos un numero entero como resultado.

Como puede observase, 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 ex- periencia por parte de quien la realice.

#### 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 des- cendente (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 realiza- remos 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 com- paración 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 alidadas 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.

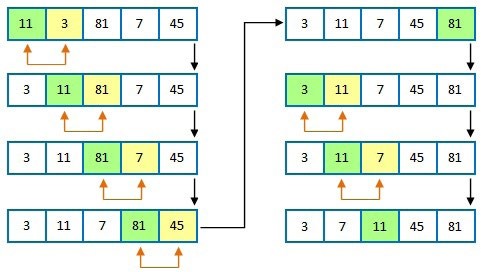


Figura 1. Esquema de funcionamiento del algoritmo burbuja

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

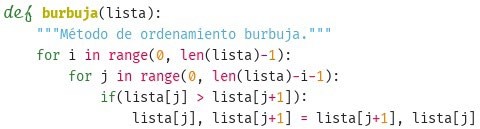


Figura 2. Método de ordenamiento burbuja

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 desperdiciando tiempo de procesa- miento y recursos, para lo cual podemos hacer una modificación del algoritmo anterior para me- jorar 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.



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 *fiag*) 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 “bur- buja 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 vuel- ve 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 alumno.

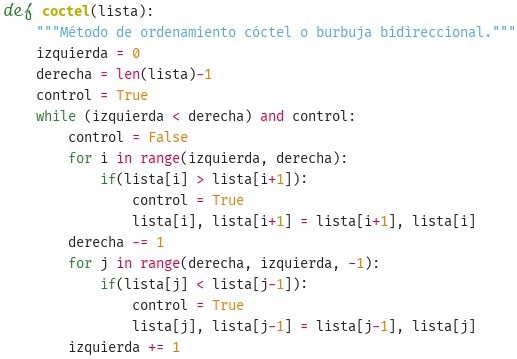


Figura 4. Método de ordenamiento cóctel

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

|  |  |
| --- | --- |
| Complejidad temporal | O(n2) |
| 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.

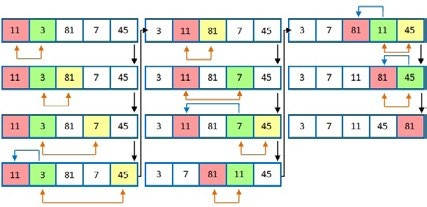


Figura 5. Funcionamiento del algoritmo selección

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



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 ite- ración 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 *mínimo* 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 mínimo (el elemento menor de los no ordenados).

Finalmente haremos la tabla comparativa de este algoritmo siguiendo los aspectos anteriormen- te especificados.

|  |  |
| --- | --- |
| Complejidad temporal | O(n2) |
| 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 ordena- dos, 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.

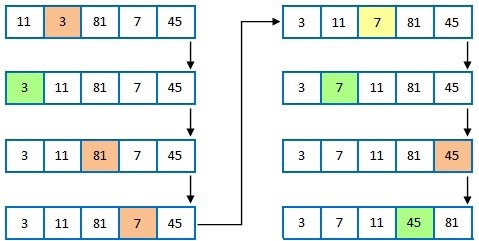


Figura 7. Forma de operar del algoritmo inserción

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



Figura 8. Método de ordenamiento inserción

Vamos a describir en detalle este algoritmo, el cual utiliza dos ciclos, un ciclo *para* de 1 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 elemen- to que se va a *ordenar* (o *insertar*) en cada iteración del ciclo *para* y el ciclo *mientras* el valor *k* sea me- nor 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 decrementa la variable *k* para ver si es necesario continuar con el desplazamiento del elemento a la izquierda.

|  |  |
| --- | --- |
| Complejidad temporal | *O*(n2) |
| 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 alumno 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 iz- quierda 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.

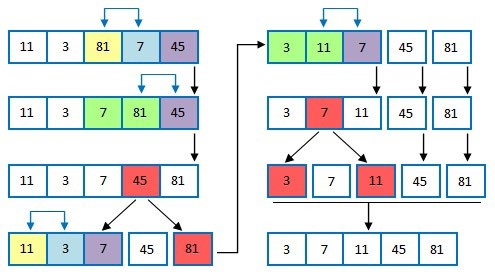


Figura 9. Esquema de funcionamiento del algoritmo de ordenamiento rápido

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.



Figura 10. Método de ordenamiento quicksort

Desglosemos analíticamente la figura anterior, se asignan los valores iniciales a la variables ín- dices 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 pivote y en la sublista derecha los elementos mayores. Luego se procede a ubicar los elementos menores al pivote 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 pivote 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 pivote se decrementa el índice derecha. Cuando estos dos ciclos se detienen, si el índice izquierda es menor que el derecha signi- fica que deben *intercambiar* los valores almacenados en dichos índices. Luego se continúa con el ciclo del *mientras* izquierda sea menor que derecha para terminar de reubicar los valores menores y mayores, cuando se terminan de intercambiarse estos elementos, se procede a ubicar el elemento pivote 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* actualizan- do 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 com- parativa 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 imple- menta 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 al- goritmo 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 su- blistas 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 acti- vidad rápida. A continuación en la figura 11 se ilustra la mecánica de funcionamiento del algoritmo.

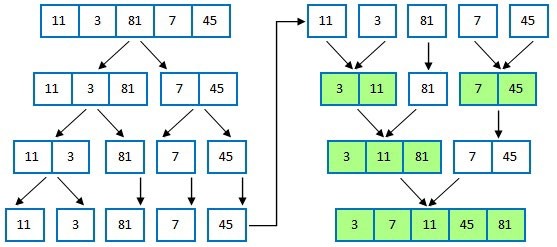


Figura 11. Funcionamiento del ordenamiento por mezcla

1. 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 pri- mera la podemos ver en la figura 12.



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 divido 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. 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 sublis- tas 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.

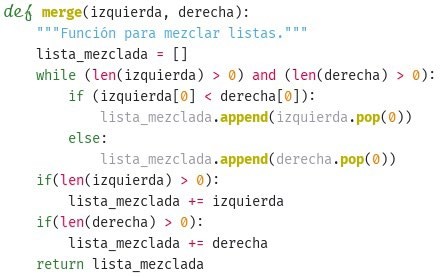


Figura 13. Función para mezclar listas

La segunda función de este algoritmo se encarga de mezclar dos listas que ya se encuentran orde- nadas, 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 me- jor de cada uno de estos para logar 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 funcio- namiento de dicho TDA, para ello se recomienda que el alumno 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 al- goritmo 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.

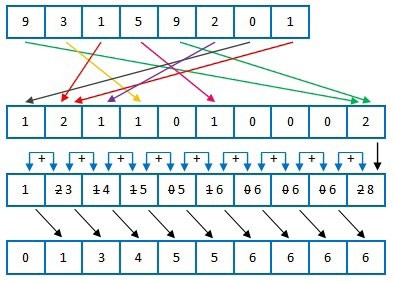


Figura 14. Generación de vector de conteo

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

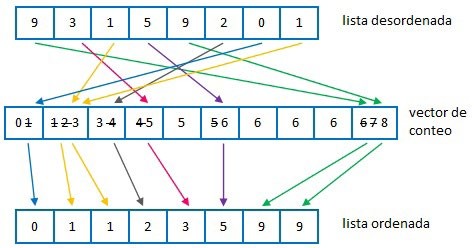


Figura 15. Funcionamiento del algoritmo ordenamiento por conteo

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 es- tructura de ciclo de ciclo, lo cual le permite mejorar significativamente su rendimiento respecto de los visto anteriormente.



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 de- fecto 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 or- denar y lo usamos como indice para incrementar en uno el valor almacenado en dicha posición en la lista de conteo. Después, tenemos que realizar una sumatorio y desplazamiento de los elementos

de la lista de conteo, tomamos los elementos de a dos, los sumamos y los almacenamos en la posi- ción 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 almace- nado en dicha posición determinamos la ubicación de dicho elemento en la lista ordenada y tam- bién 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 no- tación 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, pre- sentamos en detalle la comparación de los tiempos requeridos por los distintos algoritmos de *orde- namiento* 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*(n2) y esta eficiencia mejora notablemente a medida que au- menta el tamaño de la entrada.

|  |  |  |
| --- | --- | --- |
| **Tamaño de entrada** | **Ordenamiento** | |
| Burbuja - Selección - Inserción  O(n2) | 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 |

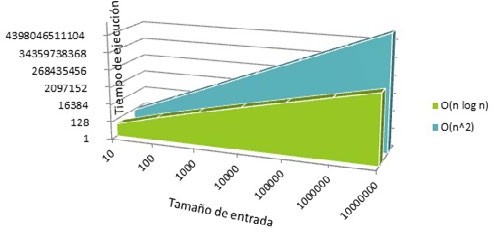


Figura 17. Comparación comportamiento de algoritmos de ordenamiento

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 deter- minar 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 ana- lizaremos 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 re- correr la lista completa. A continuación en la figura 18 se puede ver la implementación del algoritmo.

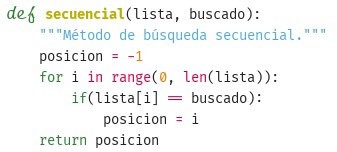


Figura 18. Método de búsqueda secuencial con centinela

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 0 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.

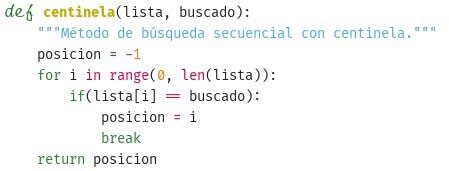


Figura 19. Método de búsqueda secuencial con centinela usando break

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 po- sición. Existen autores que no utilizan la sentencia *break* dentro de sus algoritmos por criterios par- ticulares, 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.

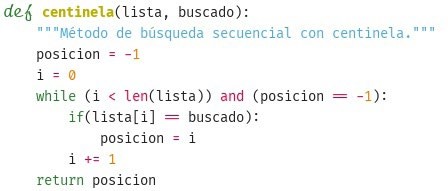


Figura 20. Método de búsqueda secuencial con centinela sin usar break

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 rendi- miento 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 esta- blecidos previamente.

|  |  |
| --- | --- |
| Complejidad temporal | *O*(n) |
| Datos ordenados/desordenados | Desordenados |
| Recursivo/No recursivo | No recursivo |
| Elemento retornado si hay repetidos | Primero con cetinela  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 ele- mentos, 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 ele- mento buscado descartando una sublistas 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.

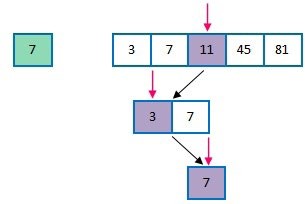


Figura 21. Esquema de funcionamiento de búsqueda binaria

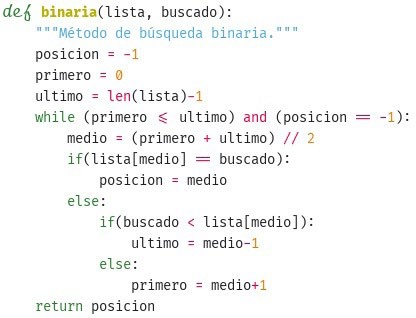


Figura 22. Método de búsqueda binaria

Es momento de desglosar analíticamente este algoritmo, primero le asignamos a la variable posi- ción 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 úl- timo (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 corres- ponda, 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 compa- ración 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 |

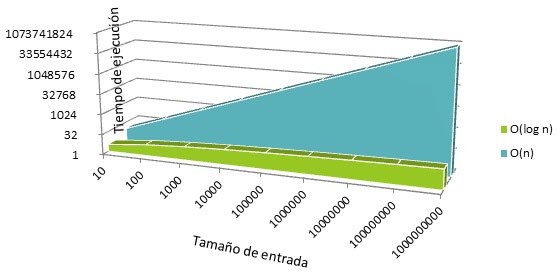


Figura 23. Comparación comportamiento de algoritmos de búsqueda

#### 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 acti- vidad 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 funcio- namiento –es decir que actividades se pueden realizar–.

Para poder efectuar esta abstracción de un modelo real a uno virtual se utiliza una técnica denomi- nada 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 asignatura 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 asignatura. 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 nece- sario que se definan algunos conceptos básicos:

1. 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 diná- mica– 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. De- pendiendo de esta pueden variar sus atributos, pero en su esencia un nodo debe tener míni- mamente 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, que- dará 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 direc- ción del puntero que la señala.

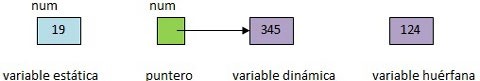


Figura 1. Variable estática, dinámica y huérfana

En la figura 2 se puede observar el funcionamiento asociado entre un puntero y una variable diná- mica. 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, 7, 3, 87] valor lista2 [1, 7, 3, 87]) las variables son punteros que apuntan a la misma va- riable 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–.

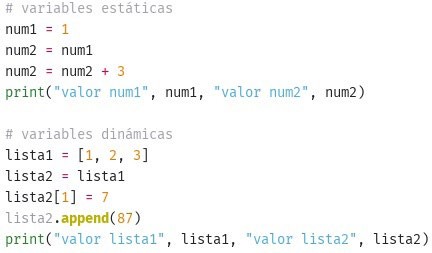


Figura 2. Variables estáticas y dinámicas

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 asigna- das automáticamente por el sistema operativo cuando creamos las variables dinámicas.

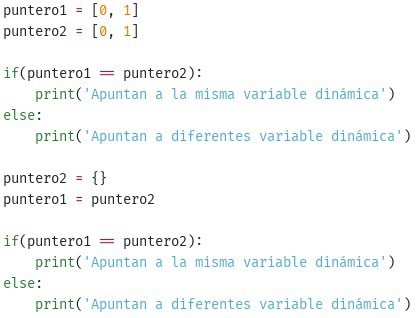


Figura 3. Comparación de punteros

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 enla- zados 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.



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 previa- mente, información (o dato) y siguiente (o enlace). Aunque puede tener más si la estructura a mode- lar 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 ana- liza en detalle su definición se puede observar que un nodo está compuesto de dos campos; infor- mación 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 repre- senta 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 sim- plemente 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–.



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 com- portamiento. 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 xn, xn-1,…, x0– 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 termino mayor del polinomio, como se puede observar en la figura 7.



Figura 7. Definición de tipos de datos

Por su parte en el bloque de la interfaz se desarrollan inicialmente los eventos necesarios para ad- ministrar 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.

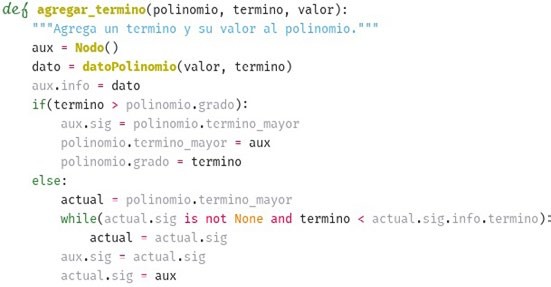


Figura 8. Eventos para administrar el polinomio parte 1

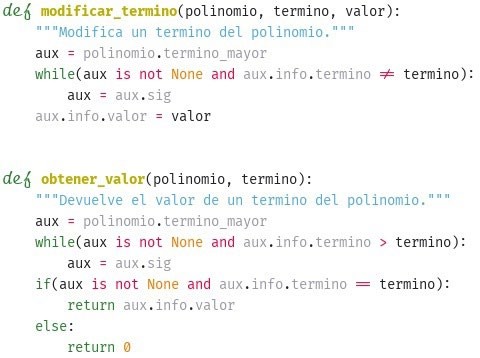


Figura 9. Eventos para administrar el polinomio parte 2

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 10 y 11.

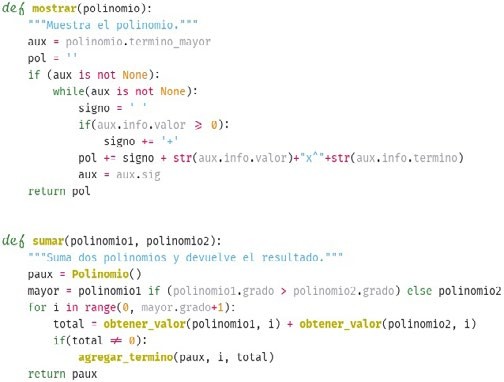


Figura 10. Eventos de operación sobre polinomios parte 1

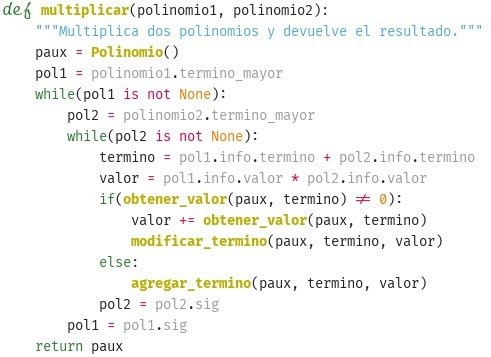


Figura 11. Eventos de operación sobre polinomios parte 2

Ahora quedará a cargo del alumno 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.

#### CAPÍTULO VI

# 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 den- tro 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-primero 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.

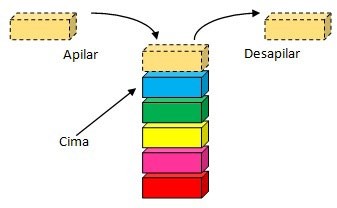


Figura 1. Funcionamiento de una pila

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 fun- cionamiento 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 acce- der 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 produc- tos en una góndola, asignaturas sobre una mesa, la ropa doblada, 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 es- tán 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 ma- nera 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 funcio- namiento, 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 iz- quierda 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 overfiow*). 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 memo- ria virtual (intercambiando la memoria a disco) extendiendo la memoria principal y por ende la capacidad de la pila.

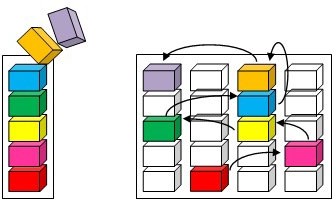


Figura 2. Tipos de representación de pila

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, compues- ta 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 informa- ción y siguiente cuyo valor inicial es *None* en ambos casos.

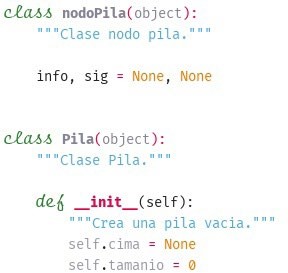


Figura 3. Definición de la estructura del TDA pila

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



Figura 4. Interfaz o eventos del TDA pila parte 1

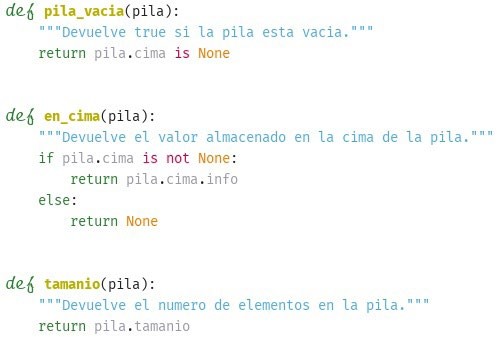


Figura 5. Interfaz o eventos del TDA pila parte 2

Cuando tenemos que apilar un nuevo elemento, se debe crea 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 direc- ción 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 que- dará a cargo del alumno.

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.

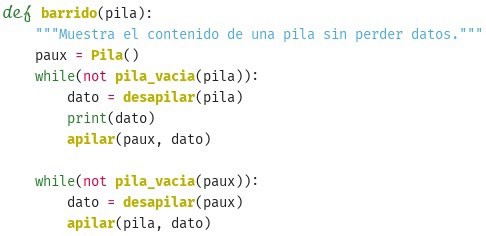


Figura 6. Barrido y reconstrucción de pila

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:



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 even- tos 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.

#### 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 ma- nejar 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-primero 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.

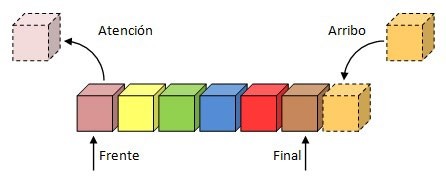


Figura 1. Funcionamiento de una cola

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 ac- ceder 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 to- que 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 ejemplos si hay una impresora en una casa u orga- nización a la cual se envían a imprimir documentos de diferentes maquinas, 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 funciona- miento: 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 or- denados siguiendo un criterio particular –salvo que se desarrolle un algoritmo que se encargue de ha- cerlo–, 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 men- cionamos previamente tendremos la limitación de cantidad de elementos si optamos por la im- plementación estática. Además como vemos en la figura 2 si el vector esta 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 esta 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 ín- dices 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 imple- mentarlo con nodos enlazados al igual que en el TDA pila.

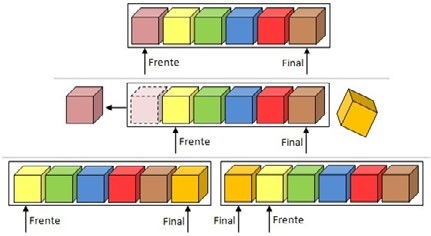


Figura 2.

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 vista anterior- mente–, y las funciones que describen su comportamiento, que se enumeran a continuación –op- cionalmente 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.

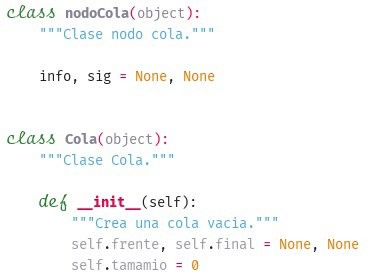


Figura 3. Definición de la estructura del TDA cola

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



Figura 4. Interfaz o eventos del TDA cola parte 1

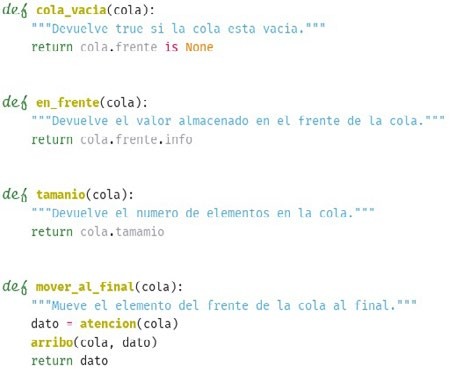


Figura 5. Interfaz o eventos del TDA cola parte 2

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 infor- mación 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 ineficientes ya que son del orden de O(n2).

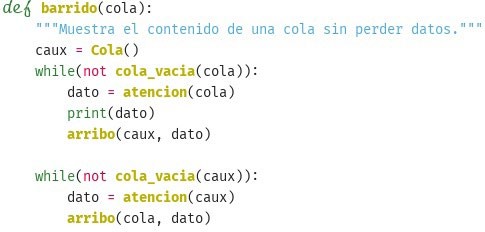


Figura 6. Barrido y reconstrucción de cola

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 tama- ño 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.

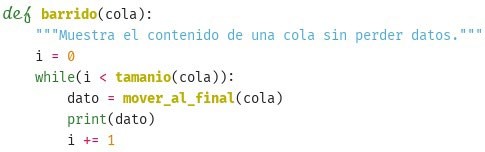
Ahora si comparamos la complejidad de ambos barridos, el primer caso está en el orden de *O*(n2) dado que se pasan todos los elementos a una cola auxiliar y después se los vuelve a pasar a la cola original para reconstruirla; mientras que el segundo caso es del orden de *O*(n) dado que solo pasa los elementos del frente al final logrando una mayor eficiencia por parte de este último algoritmo.

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 resolu- ción de un problema. Para este caso particular se cuenta con una cola de caracteres y se desea dejar en ella solamente las vocales:

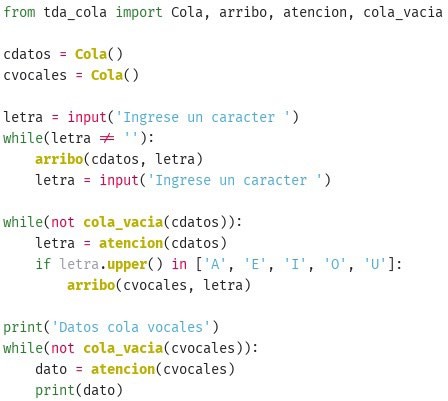


Figura 8. Ejemplo de uso del TDA Cola

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 pro- blema enunciado, luego se crean las variables de tipo cola que necesitaremos. Después 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 ira 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 utili- zan 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 maquina desde la cual se envié 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 segui- remos 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 san- gre, 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 que- dará 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 nue- va 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.

#### CAPÍTULO VIII

# 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 de- trás de otro siguiendo un orden, las listas elanzadas son su principal referente. Estas presentan mu- chas 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 aleatorio, 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 analizare- mos a lo largo del capítulo.

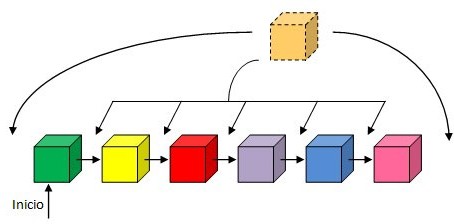


Figura 1. Funcionamiento de una lista

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 pue- den 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 electo- ral, etc. Nótese que en estos tres últimos casos es muy importante que el criterio de orden sea alfa- béticamente 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: con- tactos 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 an- teriormente, 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 imple- mentación estática o dinámica: con la primera de estas opciones cada vez que insertamos o elimina- mos 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*(n2). 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 podre- mos acceder mediante índices y aplicar búsqueda binaria, lo cual no se puede aplicar con la im- plementación 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 fun- cionamiento 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 al- macenado 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.



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.



Figura 3. Interfaz o eventos del TDA lista parte 1



Figura 4. Interfaz o eventos del TDA lista parte 2

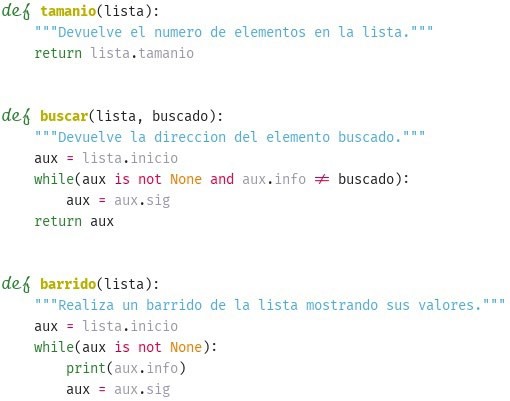


Figura 5. Interfaz o eventos del TDA lista parte 3

Cuando tenemos que agregar un nuevo elemento a la lista se crea un nodoLista, en el cual a su cam- po 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 contra- rio –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 ante- rior 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 coincida 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 direc- ción 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 proble- ma. 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:

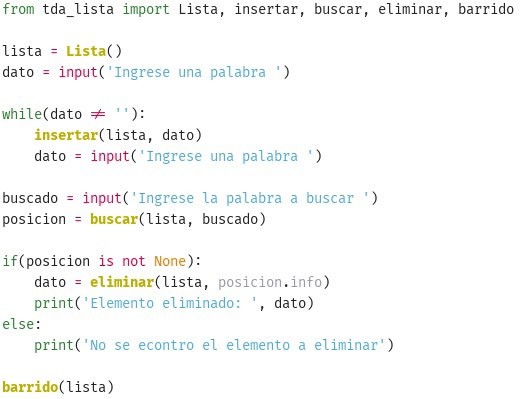


Figura 6. Ejemplo de uso del TDA Lista

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 inser- ción, 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 pro- bable 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 que- dar 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 cam- po 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 unívocamente –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 que- rer 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 re- solver 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 compara- ción en las funciones antes mencionadas.

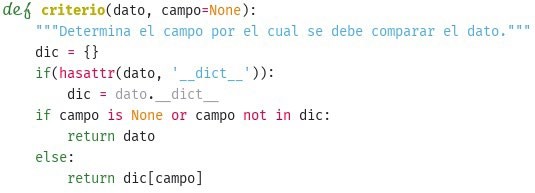


Figura 7. Función Criterio

Desglosada analíticamente la figura 7, primero necesitaremos determinar si el tipo de dato ingresa- do es un tipo de dato primitivo o una clase –utilizada como registro en este asignatura–. Una vez determi- nado 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 devuel- ve 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 primi- tivo, 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 domino 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 re- quiera. 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 quere- mos 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.



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



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

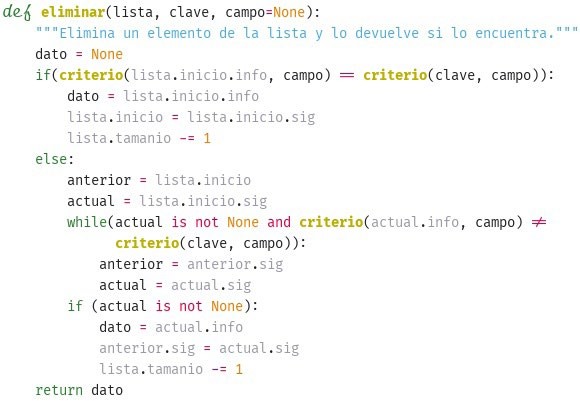


Figura 10. Uso de la función criterio en el TDA lista parte 3

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 si- guiente. Esto es muy útil para poder barrer fácilmente una lista de manera ascendente o descenden- te, 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 utili- zando un solo puntero, sin necesidad de recurrir a un puntero auxiliar como vimos anteriormente

–quedará a cargo del alumno 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 mien- tras que para operar con el campo siguiente se utiliza el puntero fin, como se observa en la figura 11.

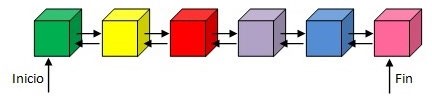


Figura 11. Esquema lista doblemente enlazada con puntero inicio y fin

Otro tipo es la lista circular. Es un caso particular de lista simplemente enlazada en la cual el atri- buto 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.

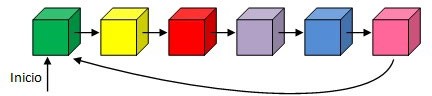


Figura 12. Esquema de lista circular

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.

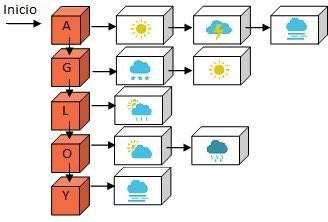


Figura 13. Estructura lista de lista dinámica

*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, sola- mente 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.



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

#### CAPÍTULO IX

# 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 sen- cilla 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.

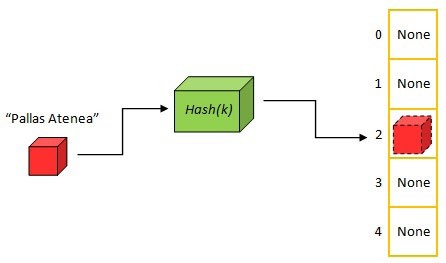


Figura 1. Esquema de una tabla *hash*

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 determi- nar 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.

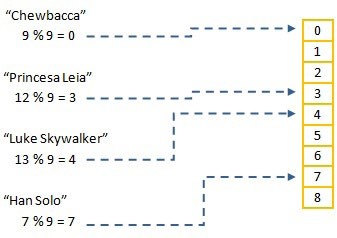


Figura 2. Funcionamiento de una tabla *hash*

Quizás el desafío más importante que enfrentaremos para utilizar esta estructura es poder encon- trar 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 busca- mos 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 es- tructura lineal estática de datos –también llamadas de tipo diccionario– a la que accedemos direc- tamente 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*, está se encargará de trans- formar 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 distribu- ya 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 obvia- mente tiene que retornar siempre la misma posición ante un mismo elemento ingresado. Por lo ge- neral, 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, comencemos 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:





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 2n, donde *n* es la cantidad de elementos de la tabla o bien 232–, finalmente del resultado obtenido será un número mayor que el tamaño de la tabla por lo que tendremos que aplicar el ope- rador 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 multiplica- ción queda de la siguiente manera:





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:







Donde el valor de 𝘢 proviene del siguiente cálculo:





Cuyo resultado aproximado es 𝘢 ≃ 2 654 435 769,4972305, dependiendo de las especificaciones del lenguaje en que se programe, y 𝜔 = 232, cuyo resultado es 𝜔 = 4 294 967 296. De esta forma, finalmente se obtiene que el valor sea aproximadamente A ≃ 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 procedimien- to 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:

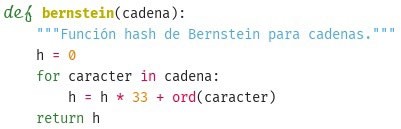


Figura 3. Función *hash* de Bernstein para cadenas de texto

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 previa- mente 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-𝘢)) donde 𝘢 representa el factor de carga de la tabla y se obtiene con la siguiente fórmula 𝘢 = 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 aglo- meramiento 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 aglo- meramiento 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 ma- yor 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 sonde si existen otros elementos en la tabla que coincida con la calve 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 eli- minar, 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.

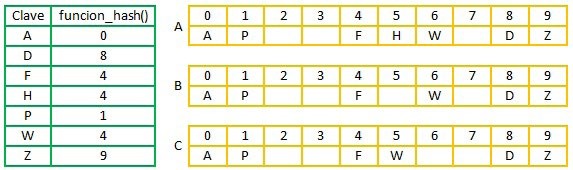


Figura 5. Eliminar elemento en tabla *hash* cerrada

Otra alternativa para manejar las colisiones –y evitar el problema del aglomeramineto 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 da- tos 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).

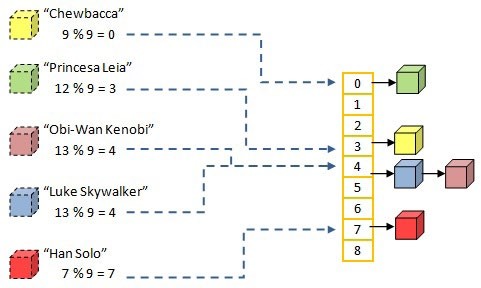


Figura 6. Tabla *hash* abierta o encadenada

Para que estas tablas funcionen eficientemente depende exclusivamente de la función *hash* utiliza- da. Esta debe distribuir los elementos de manera uniforme por la tabla, como mencionamos previa- mente, 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 + 𝘢) –si los elementos están dis- tribuidos uniformemente–, donde 𝘢 representan el factor de carga de los casilleros de la tabla y se calcula 𝘢 = 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 + 𝘢), 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 adentrar- nos 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, deter- minado 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 nin- gún elemento–;
4. buscar(tabla, dato). Devuelve la posición de la tabla en la que se encuentra el elemento busca- do, 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 vec- tor. 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 colisio- nes que ocurran.
7. cantidad\_elementos(tabla). Devuelve la cantidad de elementos en la tabla, puede requerir uti- lizar alguna función extra si es una tabla encadenada para contar los elementos de la estruc- tura 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 incomple- tas–; además solo se presenta una función *hash* y quedará a cargo del alumno 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*.



Figura 7. Interfaz o eventos del TDA tabla *hash* parte 1



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

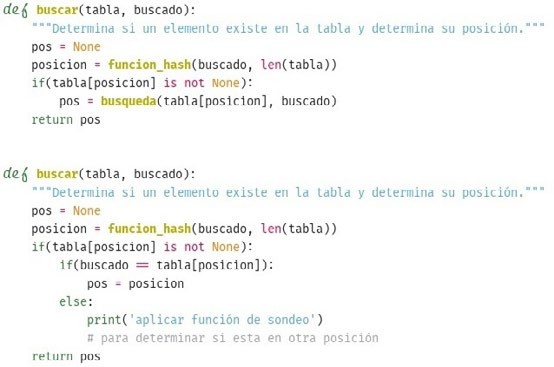


Figura 9. Interfaz o eventos del TDA tabla *hash* parte 3

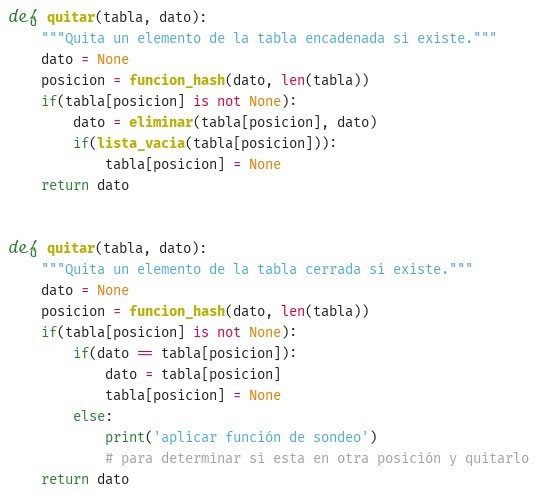


Figura 10. Interfaz o eventos del TDA tabla *hash* parte 4

En el momento que un nuevo elemento se va cargar a la tabla se utiliza la función *hash* para deter- minar 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 es- tar 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 ante- riormente. 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 11. 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 men- cionado previamente, que suma los caracteres del nombre meno los espacios para luego aplicar un *hash* por división.



Figura 11. 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 bus- car y se realiza la búsqueda para determinar si el personaje buscado está cargado en la tabla o no.