



**UTN.BA**  
UNIVERSIDAD TECNOLÓGICA NACIONAL  
FACULTAD REGIONAL BUENOS AIRES

**Centro de  
e-Learning**

# **FUNDAMENTOS DE LA PROGRAMACIÓN**

Centro de e-Learning - FRBA - UTN

**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**



## **MÓDULO 2 - UNIDAD 8**

# **Estructuras de datos**

Centro de e-Learning - FRBA - UTN



## Presentación:

En esta Unidad nos enfocaremos en comprender y analizar qué es una estructura de datos y cuáles son sus principales tipos.

Analizaremos las listas en sus dos principales tipos: lineales y simples, que son relativa facilidad para ser interpretados en pseudocódigo, dejando otras de mayor complejidad fuera del programa, como las listas doblemente enlazadas, para las cuales se requiere un nivel avanzado de conocimientos en programación.

También analizaremos las estructuras conocidas como pilas y colas, que conceptualmente comparten muchos paralelos, pero que su implementación y uso tienen unas importantes particularidades.



## Objetivos:

### Que los participantes:

- Comprendan el concepto de estructura de datos
- Conozcan las estructuras de datos principales: listas (lineales y simples), pilas y colas
- Comprendan el concepto de los métodos de ordenamiento por selección, por inserción, por intercambio y por intercalación
- Comprendan el concepto de los métodos de búsqueda secuenciales y búsqueda binaria



## Bloques temáticos:

### 1. Introducción

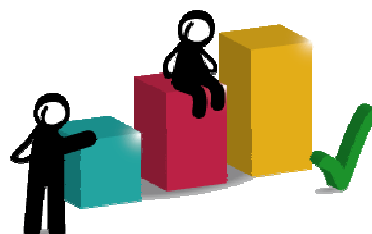
- 1.1 Listas lineales
- 1.2 Listas simples
- 1.3 Colas
- 1.4 Pilas

### 2. Ordenamientos

- 2.1 Ordenamiento por selección
- 2.2 Ordenamiento por inserción
- 2.3 Ordenamiento por intercambio (burbujeo)
- 2.4 Intercalación

### 3. Búsquedas

- 3.1 Búsqueda secuencial
- 3.2 Búsqueda binaria



## Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC\*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

\* El MEC es el modelo de E-learning colaborativo de nuestro Centro.



## Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.



## **1. Introducción**

En la Unidad anterior se daban ejemplos de composición de elementos para formar otros de mayor complejidad y prestaciones, como los arreglos y matrices. En la misma tónica, en esta unidad seguiremos analizando los diversos usos de los arreglos, especialmente los vectores, para crear y usar distintas estructuras de datos, particularmente útiles a la hora de realizar distintos tipos de procesamiento de datos en memoria.

### **1.1 Listas lineales**

Las listas son estructuras flexibles, ya que pueden crecer y reducirse, y sus elementos pueden ser accedidos o insertados en cualquier posición dentro de las mismas, como así también eliminados.

Matemáticamente, una lista es un conjunto de cero o más elementos de un tipo dado. Usualmente, una lista se representa como una sucesión de elementos separados por comas:

$$a(1), a(2), \dots, a(n) \text{ con } n \geq 0$$

El número  $n$  de elementos indica la longitud de la lista. Si suponemos que  $n \geq 1$ , decimos que  $a(1)$  es el primero elemento de la lista y  $a(n)$  es el último elemento. Si  $n = 0$ , se tiene la lista vacía, es decir, la lista no contiene elementos.

En una lista, los elementos están linealmente ordenados de acuerdo a su posición dentro de ella:

$$a(i) \text{ precede a } a(i+1) \text{ para } i = 1, 2, \dots, n-1$$

$$a(i) \text{ sigue a } a(i-1) \text{ para } i = 2, \dots, n$$





Como dijimos anteriormente, cada elemento de una lista está compuesto por uno o más campos. Cada campo puede ser un dato simple (primitivo) o estructurado.

Son ejemplos de listas:

- Los alumnos inscriptos en un curso de programación de la UTN,
- Los datos de los internados en la sala de un hospital,
- La lista de espera de pasajeros de un vuelo,
- Los coeficientes de un polinomio,
- La solución de un sistema de  $n$  ecuaciones lineales con  $n$  incógnitas,
- La lista de ejemplos de listas lineales

Entre las operaciones más comunes que se pueden realizar con listas lineales podemos citar las siguientes:

- Acceder a un  $k$ -ésimo elemento de una lista para examinar o cambiar el contenido de sus campos
- Insertar un nuevo elemento en la lista antes del  $k$ -ésimo elemento
- Eliminar el  $k$ -ésimo elemento de la lista
- Combinar dos o más listas para formar una nueva,
- Dividir o seccionar una lista en sublistas,
- Hacer una copia de la lista lineal,
- Determinar la longitud de una lista,



- Ordenar los elementos de una lista en forma ascendente o descendente, según los valores de uno o más de los campos que contiene cada elemento,
- Explorar una lista en busca de un elemento en particular que contiene un campo con un determinado valor.

Vale aclarar que raramente en una aplicación deberemos realizar todas las operaciones arriba mencionadas.

El término "k-ésimo" (se pronuncia "caésimo") se utiliza de la misma forma para nombre un elemento "x" o "n", cuando se representan una posición y se las numera en base a esa variable. Es análogo a término "enésimo", donde la posición está dada por "n".

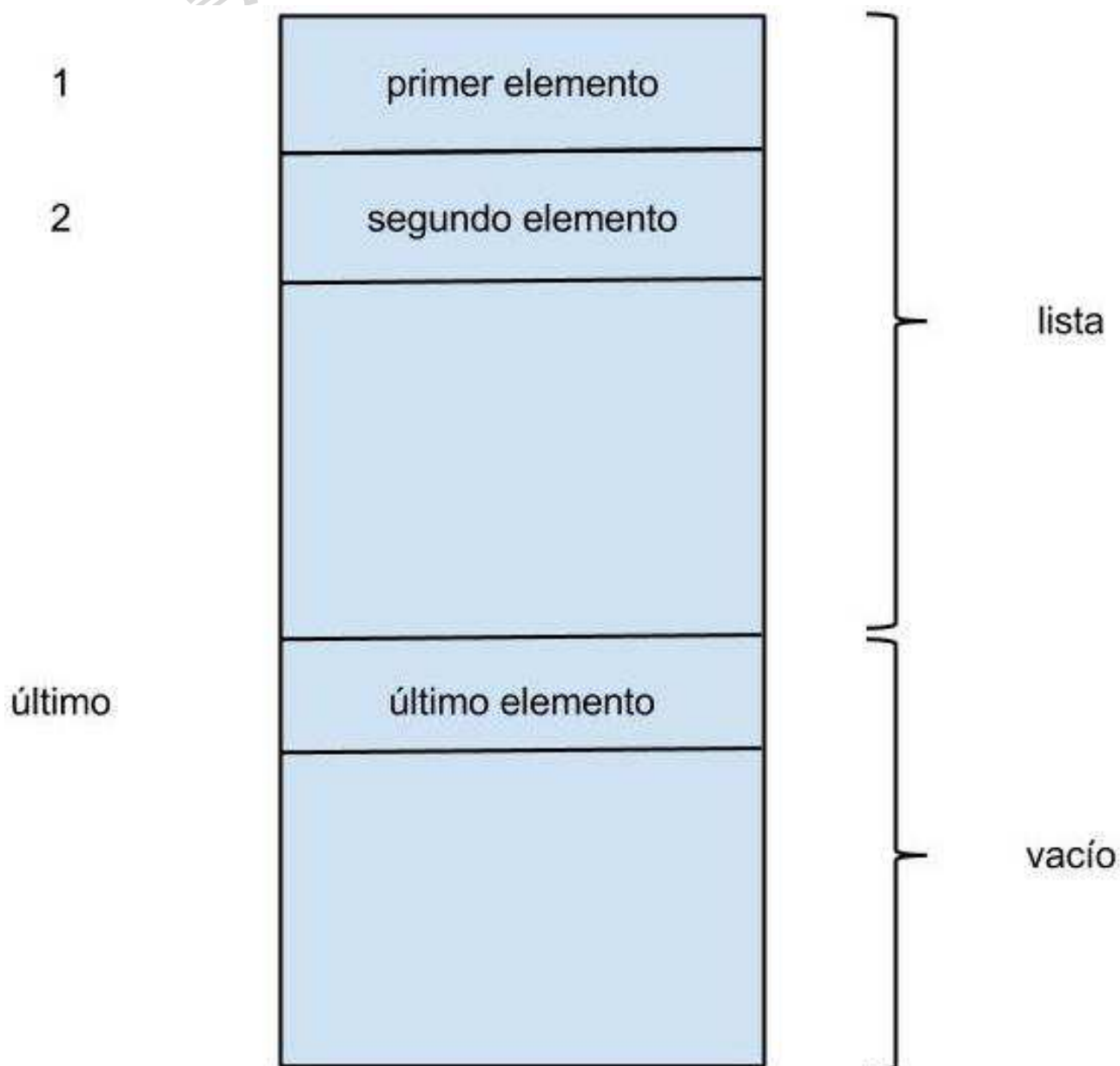
## 1.2 Listas simples

Existen varias formas comunes de implementar listas lineales en la memoria de la computadora. Una de ellas es el método de distribución de la memoria, que aprovecha las características unidimensionales de ésta, y se denomina almacenamiento secuencial.

La forma más simple y natural de almacenar una lista lineal en la memoria de la computadora es ubicar los elementos en la misma, de modo que sean lógicamente adyacentes, en posiciones físicamente adyacentes, es decir, un elemento detrás de otro. En este caso la lista lineal es implementada con un arreglo lineal. Con esta representación, se accede fácilmente a cualquier elemento de la lista y nuevos elementos pueden ser agregados rápidamente, al final de la misma.



Sin embargo, insertar un elemento de esta manera en el medio de la lista requiere un corrimiento de un lugar de los elementos que lo siguen con el fin de ubicarlo. Del mismo modo, eliminar cualquier elemento (excepto el último de la lista) también requiere el corrimiento de los mismos con el fin de evitar los espacios vacíos.





Para implementar una lista con un arreglo se debe declarar a éste la dimensión suficientemente grande como para contener todos los posibles elementos de la lista.

Por ejemplo, si se desea implementar la lista de enfermos internados en una sala de un hospital, la dimensión del arreglo debe ser el número máximo de camas de la sala. Además, es necesario definir una variable entera ("integer") para almacenar el índice del último elemento de la lista. Si este valor es  $m$  y la dimensión es  $n$ , debe verificarse la relación  $m \leq n$ .

A continuación desarrollaremos los algoritmos correspondientes a algunas de las operaciones más frecuentes que pueden realizarse con listas.

Acceso a un elemento:

```
...  
funcion string acceder (string l[ ], integer m, integer k)  
  var string x = ""  
  si m = 0 entonces  
    mostrar: "Lista vacía"  
  sino  
    si k < 1 o k > m entonces  
      mostrar: "Posición no válida"  
    sino  
      x = l [k]  
    fin si  
  fin si  
  retornar: x  
fin funcion  
...
```



En esta función se accede al k-ésimo elemento de una lista. Si la lista está vacía, se muestra un mensaje de error. Si el valor de k está fuera de rango del índice de la lista, se muestra otro mensaje de error. En cualquier otro caso, se retorna el valor solicitado.

Las variables utilizadas son:

- l: vector de strings de tamaño suficiente para almacenar todos los posibles elementos de la lista
- x: variable del mismo tipo que "l" en la cual retorna el k-ésimo elemento
- m: variable entera que indica la longitud de la lista
- k: variable entera cuyo valor indica la posición del elemento a retornar

Ingreso de un elemento nuevo en la lista lineal:

```
...
funcion ingresar (string l[ ], integer m, integer k, string x)
    var integer i = 0
    si m = n entonces          //"n" es global
        mostrar: "La lista está llena"
    sino
        si k < 1 o k >= m+1 entonces
            mostrar: "La posición no es válida"
        sino
            i = m
            mientras i >= k
                l [ i +1] = l [i]
                i = i - 1
            fin mientras
            l [k] = x
            m = m + 1
        fin si
    fin si
fin funcion
...
```



En esta función se inserta un nuevo elemento en el k-ésimo lugar de la lista. Nuevamente, se comprueba si la lista está llena y se muestra un mensaje de error. Posteriormente, se valida si el valor de k se encuentra fuera del rango del índice de la lista. En cualquier otro caso, se mueven los elementos de las posiciones: k, k+1..., último a las posiciones k+1, k+2..., último+1, respectivamente. Finalmente se inserta el nuevo elemento en la posición k y se incrementa en 1 la longitud de la lista.

Las variables utilizadas son:

- l: vector de strings de tamaño suficiente para almacenar todos los posibles elementos de la lista
- x: variable del mismo tipo que "l" que contiene el valor a insertar en la lista
- k: lugar de la lista en la que se ubicará al nuevo elemento "x"
- n: longitud del arreglo (variable global del programa)
- m: longitud de la lista
- i: variable entera utilizada como índice del arreglo "l"

A continuación, la comprobación con un caso práctico:

**Datos de prueba:**

n=9    tamaño del vector

k=3    lugar del vector donde quiero agregar mi elemento x

m=4    cantidad de posiciones ocupadas en el vector

x=J    letra "J" que quiero agregar en el vector

La explicación es: tengo un vector de 9 posiciones, de las cuales tengo 4 ocupadas. Lo que quiero hacer es agregar en la posición 3 la letra "J" sin perder los valores actuales.

Mi vector actual:	Vector al que quiero llegar:
l[1]=A	l[1]=A
l[2]=B	l[2]=B
l[3]=C	l[3]=J



$I[4]=D$	$I[4]=C$ $I[5]=D$
----------	----------------------

**Comprobación:**

```
si m = n entonces                // 4 = 9? FALSO
    mostrar: "La lista está llena" // NO ENTRA ACA
sino
    si k < 1 o k >= m+1 entonces  // 3 < 1 (FALSO) o 3 >= 4+1 (FALSO)
        mostrar: "La posición no es válida" // NO ENTRA ACA
    sino
        i = m                    // i = 4
        mientras i >= k          // mientras 4 >= 3 (PRIMERA VEZ DE 2)
            I [ i +1] = I [i]    // I[5] = I[4] (MUEVO LA "D" A LA POSICION 5)
            i = i - 1            // i = 3
        fin mientras

                                // mientras 3 >= 3 (SEGUNDA VEZ DE 2)
                                // I [4] = I [3] (MUEVO LA "C" A LA POSICION 4)
                                // i = 2

        I [k] = x                // INSERTO EN I[3] LA LETRA "J"
        m = m + 1 // INCREMENTO LA CANTIDAD DE ELEMENTOS DE LA LISTA
    fin si
fin si
```



Eliminar un elemento de la lista lineal:

```
funcion eliminar (string l[ ], integer m, integer k)
    var integer i = 0
    si m = 0 entonces
        mostrar: "La lista está vacía"
    sino
        si k < 1 o k > m entonces
            mostrar: "La posición no existe"
        sino
            i = k
            mientras i <= m - 1
                l [i] = l [i+1]
                i = i + 1
            fin mientras
            m = m - 1
        fin si
    fin si
fin funcion
```

En esta función se elimina el elemento del k-ésimo lugar de la lista.

Se hacen las validaciones de rigor y en caso de no producirse ningún error, se mueven los elementos de la posiciones: k+1, k+2..., último a las posiciones: k, k+1..., último-1 respectivamente. Finalmente se reduce el tamaño de la lista en 1.

Las variables utilizadas son similares al del caso anterior.

A continuación, la comprobación con el primer caso de prueba práctico:





- Lote de prueba
  - $I[3] = "A[1]"; "B[2]"; "C[3]"$
  - $k=2$  (posición a eliminar, conteniendo el valor "B")
- Caso de prueba:
  - si  $m = 0 \rightarrow$  Falso! Continúa al "sino"
  - si  $2 < 1$  o  $2 > 3 \rightarrow$  Falso! Continúa al "sino"
  - $i=2$

1er mientras  $2 \leq (3-1)$  (V!)

$I[2] = I[3]$  //muevo la "C" al lugar de la "B"

$i=2+1=3$

fin mientras

2do mientras  $3 \leq (3-1)$  (F!)

fin mientras

$m=m-1 \rightarrow 3-1 \rightarrow m=2$  //el tamaño de la lista (m) pasa a ser 2

La comprobación con el segundo caso de prueba práctico:

- Lote de prueba
  - $I[3] = "A[1]"; "B[2]"; "C[3]"$
  - $k=3$  (posición a eliminar, conteniendo el valor "C")
- Caso de prueba:
  - $i=3$
  - 1er mientras  $3 \leq (3-1)$  (F!)
  - fin mientras
  - $m=m-1 \rightarrow 3-1 \rightarrow m=2$



Localizar un elemento en la lista lineal:

```
funcion integer localizar (string l[ ], integer m, string x)
  var integer i = 0
  var integer k = 0
  si m = 0 entonces
    mostrar: "La lista está vacía"
  sino
    i = 1
    // el símbolo "<>" aquí debajo indica "distinto de"
    mientras i <= m y l [i] <> x
      i = i + 1
    fin mientras
    si l [i] = x entonces
      k = i
    sino
      mostrar: "Elemento no encontrado"
    fin si
  fin si
  retornar: k
fin funcion
```

En esta función se localiza la posición de un elemento en la lista, según un valor dado a buscar.

Inicialmente, se valida el tamaño de la lista. En caso de no producirse error, se recorre la lista (más adelante formalizaremos este tipo de búsqueda) comparando cada posición con el valor dado. En caso de encontrarse el dato buscado, se retorna la posición en la que se encuentra.



Las variables utilizadas son:

- l: vector de strings de tamaño suficiente para almacenar todos los posibles elementos de la lista
- x: variable que contiene el string a buscar en la lista de strings "l"
- k: variable entera en la que se guarda la posición del elemento encontrado (si se encuentra). Es el valor devuelto por la función
- i: índice del arreglo "l"
- m: variable entera que indica la longitud de la lista

## 1.3 Colas

Una Cola (del inglés queue) es una lista lineal en la cual todas las inserciones se realizan desde uno de los extremos de la lista y todas las eliminaciones y los accesos se realizan desde el otro extremo.

Un ejemplo podría ser una "cola" de personas esperando un servicio. En una cola el elemento más "viejo" es el primero en eliminarse, esto quiere decir, que los elementos abandonan la lista en el mismo orden en que han entrado. Se los suele identificar como FIFO, por el inglés "first in, first out" (primero en entrar, primero en salir).

En la primera representación de una cola debemos definir dos variables "primero" (indica la posición anterior al primer elemento) y "ultimo" (indica la posición del último elemento).

Con:

```
...  
primero = ultimo = 0  
...
```

Indicamos que la cola está vacía.



Para agregar un elemento al final de la cola, hacemos:

```
...  
ultimo = ultimo + 1  
cola [ultimo] = x  
...
```

Para eliminar un elemento y retornar su valor hacemos:

```
...  
primero = primero + 1  
x = cola [primero]  
si primero = ultimo entonces  
    primero = 0  
    ultimo = 0  
fin si  
...
```

¿Qué sucede si “ultimo” se mantiene siempre detrás de “primero”? Esto implica que la cola siempre tiene, al menos, un elemento y el arreglo consiste entonces de

cola [1],cola [2]...,cola [1000]...,cola [100000]

y, por lo tanto, se pierde un espacio grande de la memoria. Por ejemplo, la cola puede estar constituida por un solo elemento que ocupa la posición 945 y por lo tanto estaríamos perdiendo 944 espacio previos. La implementación anterior serviría en el caso de que regularmente “primero” alcance a “ultimo”. Para circunscribir este problema, evitando el desperdicio de espacio en memoria, definimos un arreglo de “m” elementos de manera circular, de tal forma que cola[1] sigue a cola[m]. Redefinimos entonces los algoritmos anteriores, declarando primero = ultimo = m, para indicar la cola vacía.



En los casos anteriores, no hemos tenido en cuenta la presencia de, al menos, un elemento cuando realizamos la operación de eliminación, ni tampoco al efectuar la inserción de un elemento verificamos la existencia de lugares disponibles en el arreglo.

Los siguientes procedimientos toman en cuenta el hecho de que estas restricciones no sean satisfechas automáticamente.

```
...
funcion agregar(string cola[], integer primero, integer ultimo, string nuevo, integer tamanoMaximo)
si ultimo+1 > tamanoMaximo entonces
    mostrar: "Cola llena, no se puede agrega más elementos"
sino
    si primero = ultimo = 0 entonces
        //está vacía! coloco la posición inicial de la cola en 1
        primero = 1
    fin si
    ultimo=ultimo+1
    cola[ultimo]=nuevo
    mostrar: "Elemento " + nuevo + " agregado con éxito en la posición " + ultimo
fin si
fin funcion
...
```

A continuación, la comprobación con la prueba práctica para ingresar un elemento en una cosa que puede estar vacía o ya tener algún elemento, usando el método visto arriba:

- Lote de prueba 1
  - cola[]=vacía
  - nuevo="A"
  - primero=0
  - ultimo=0
  - tamanoMaximo=3 (definimos que la cola de ejemplo va a poder almacenar hasta 3 elementos)



- Caso de prueba:

si  $0 + 1 > 3$  entonces

Falso! NO ENTRA!

sino

si  $0 = 0 = 0$  entonces

Verdadero! Es el primer elemento de la cola

primero=1

fin si

ultimo =  $0 + 1 = 1$

cola[1] = nuevo = "A"

mostrar:...

fin si

...

- Lote de prueba 2

- nuevo="B"
- primero=1
- ultimo=1
- tamañoMaximo=3

- Caso de prueba:

si  $1 + 1 > 3$  entonces

Falso! NO ENTRA!

sino

si  $1 = 1 = 0$  entonces



Falso! NO ENTRA!

fin si

ultimo = 1 + 1 = 2

cola[2] = nuevo = "B"

mostrar:...

fin si

...

- Lote de prueba 3
  - nuevo="C"
  - primero=1
  - ultimo=2
  - tamañoMaximo=3

- Caso de prueba:

si  $2 + 1 > 3$  entonces

Falso! NO ENTRA!

sino

si  $2 = 1 = 0$  entonces

Falso! NO ENTRA!

fin si

ultimo = 2 + 1 = 3

cola[3] = nuevo = "C"

mostrar:...

fin si

...



Quedando el arreglo:

cola[1] = "A"

cola[2] = "B"

cola[3] = "C"

Y las referencias de la cola:

ultimo = 3

primero = 1

Para ir vaciando la cola, se podría utilizar la siguiente función:

```
...  
funcion eliminar(integer ultimo, integer primero)  
  si ultimo = primero = 0 entonces  
    mostrar: "La cola no se llenó"  
  sino si primero <= ultimo entonces  
    primero = primero + 1  
  sino  
    mostrar: "La cola ya fue vaciada"  
  fin si  
fin funcion  
...
```





Lo que hacemos en este caso es comprobar en el primer "si" si la cola ya se encuentra vacía. Esto ocurre cuando el primero y el último valen cero (ambas referencias se encuentran apuntando a la mismo lugar o posición dentro del arreglo).

Adicionalmente se verifica que se esté intentando sacar un elemento válido de la cola. Si es así, no se modifica el contenido del arreglo (no eliminar "físicamente" el valor), sino que se modifica la referencia al "primero", por lo que deja de apuntar a posición para pasar a apuntar a otra.

A continuación, la comprobación con la prueba práctica para vaciar la cola llenada anteriormente. Para esto, reemplazaremos los valores de la función "eliminar".

Lote de prueba 1

- ultimo = 3
- primero = 1

si  $3 = 1 = 0$  entonces

//FALSO

sino si  $1 \leq 3$  entonces

//VERDADERO

$2 = 1 + 1$  //elimino referencia a cola[1] = "A"

sino

//FALSO

fin si

Lote de prueba 2

- ultimo = 3
- primero = 2



si  $3 = 2 = 0$  entonces

//FALSO

sino si  $2 \leq 3$  entonces

//VERDADERO

$3 = 2 + 1$  //elimino referencia a cola[2] = "B"

sino

//FALSO

fin si

Lote de prueba 3

- ultimo = 3
- primero = 3

si  $3 = 3 = 0$  entonces

//FALSO

sino si  $3 \leq 3$  entonces

//VERDADERO

$4 = 3 + 1$  // elimino referencia a cola[3] = "C"

sino

//FALSO

fin si



Lote de prueba 4

- ultimo = 3
- primero = 4

si 3 = 4 = 0 entonces

//FALSO

sino si 4 <= 3 entonces

//FALSO

sino

//VERDADERO

mostrar: "La cola ya fue vaciada"

fin si

Otra forma de vaciar toda la cola de una sola vez, sería utilizando un ciclo mientras, como se ve a continuación:

```
...
funcion eliminarMostrando(string cola[], integer ultimo, integer primero)
    si ultimo = primero = 0 entonces
        mostrar: "La cola no se llenó"
    sino
        mientras primero <= ultimo hacer
            mostrar: cola[primero]
            primero = primero + 1
        fin mientras
        mostrar: "La cola ya fue vaciada"
    fin si
fin funcion
...
```



A continuación, la comprobación con la prueba práctica para vaciar, con la nueva función, la cola llenada originalmente. Para esto, reemplazaremos los valores de la función "eliminarMostrando".

Se valida que la cola no está vacía ("ultimo = primero = 0"), ya que:

ultimo = 3

primero = 1

//Primera vuelta del ciclo mientras

mientras 1 <= 3 hacer

    mostrar: cola[1]     //muestra "A"

    primero = 1 + 1 = 2

fin mientras

//Segunda vuelta del ciclo mientras

mientras 2 <= 3 hacer

    mostrar: cola[2]     //muestra "B"

    primero = 2 + 1 = 3

fin mientras

//Tercera vuelta del ciclo mientras

mientras 3 <= 3 hacer

    mostrar: cola[3]     //muestra "C"

    primero = 3 + 1 = 4



fin mientras

//Cuarta vuelta del ciclo mientras

mientras 4 <= 3 hacer

//NO ENTRA!

fin mientras

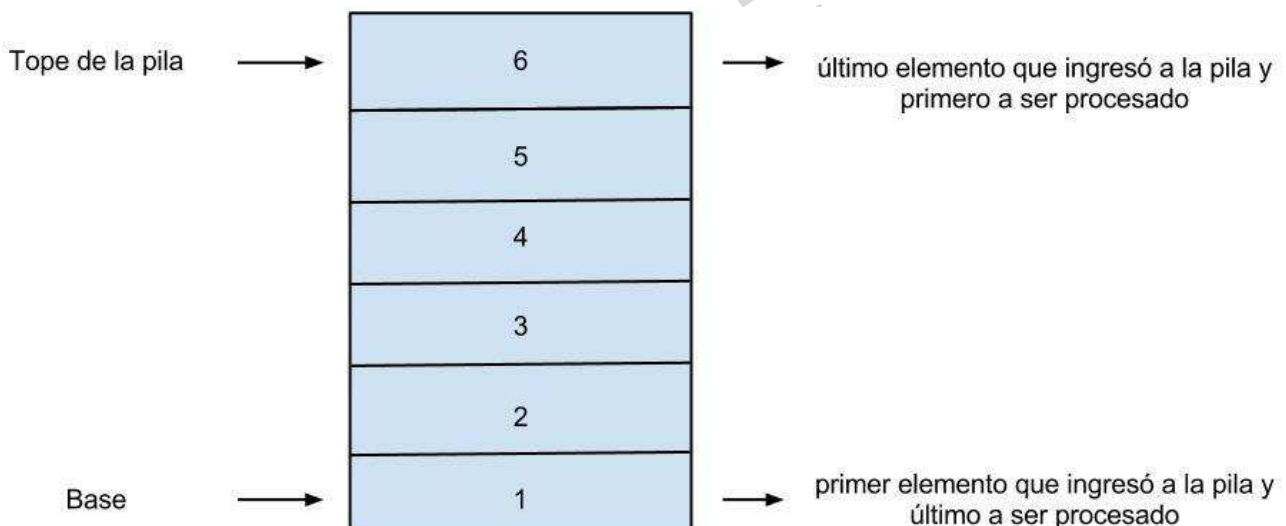
Finalmente, se ejecuta la última instrucción:

mostrar: "La cola ya fue vaciada"

## 1.4 Pilas

Una Pila (del inglés stack) es una lista lineal en la cual todas las inserciones y eliminaciones y accesos se realizan por uno de los extremos. Un ejemplo podría ser la "pila" de platos de un estante, donde siempre sale el elemento más "joven", es decir, el elemento que ha sido insertado último.

Se los suele identificar como LIFO, por el inglés "last in, first out" (último en entrar, primero en salir).





## **2. Ordenamientos**

Continuando con el análisis de estructuras de datos en memoria, encontramos que la ordenación y la intercalación proveen un medio para la organización de la información, facilitando, de esta manera, la recuperación de datos específicos. Imaginemos lo difícil que sería usar un diccionario si sus palabras no estuvieran ordenadas. Los siguientes ejemplos muestran un conjunto de datos numéricos ordenados de manera no descendente y otro de cadenas de caracteres ordenados según el orden lexicográfico.

**{7, 15, 18, 18, 29, 45}**

**{a, ábaco, abajo, ballena, de, dedo, manual}**

Debido a la importancia de estas operaciones, se han desarrollado una gran variedad de algoritmos para tratarlas, de los cuales analizaremos los más representativos.

La elección de un algoritmo debe tener en cuenta el uso eficiente de la memoria que se tiene disponible, como así también, el tiempo de ejecución del mismo. Una primera clasificación de los métodos de ordenación se efectúa entre aquellos que ordenan los ítems, transfiriéndolos desde un arreglo A a un arreglo B y los que los ordenan, en el mismo arreglo.

### **2.1 Ordenamiento por selección**

Dado un vector A de n elementos, el método de selección para ordenar el mismo en forma ascendente es el siguiente:

1. Encontrar el elemento más pequeño del arreglo (el proceso requerido se denomina “pasada”), transferirlo a la primera posición de un arreglo B, reemplazar el elemento en el arreglo A por un valor t, donde t se elige mayor que cualquier elemento del arreglo.



2. Buscar nuevamente en el arreglo A el elemento más pequeño (esta vez el segundo número más pequeño será seleccionado porque el más pequeño ha sido reemplazado por t), se transfiere este elemento a la segunda posición de B, se reemplaza el elemento (en A) por t.

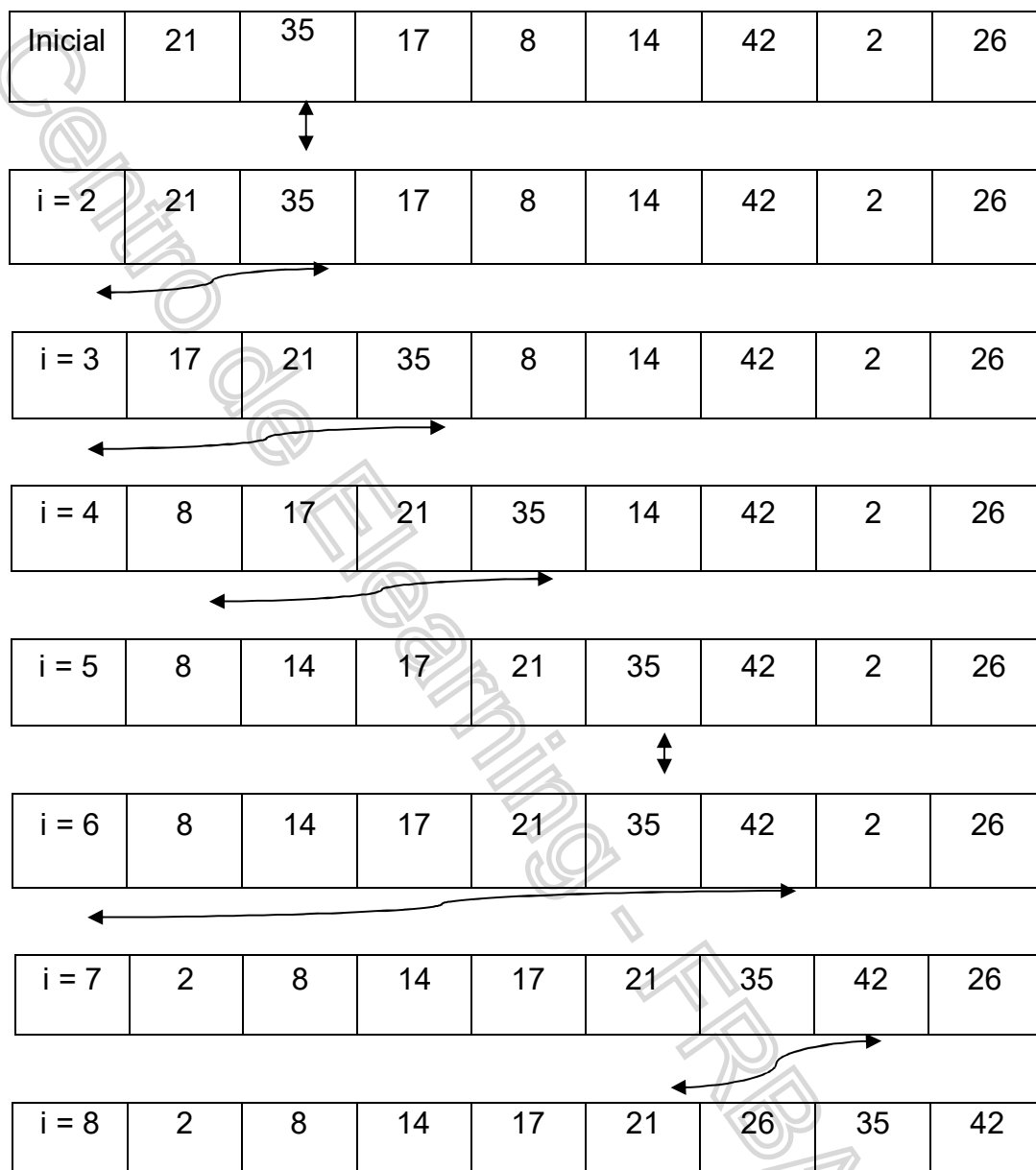
Se continúa repitiendo el mismo procedimiento hasta haber agotado los n elementos del arreglo A.

La desventaja de este método reside en que utiliza el doble de espacio de memoria que otros (por el hecho de tener un vector principal y otro del mismo tamaño como auxiliar). Si se tienen restricciones de memoria, se puede volver un problema insalvable a cierta escala.

## 2.2 Ordenamiento por inserción

Este método para ordenar arreglos en forma ascendente consiste en:

1. Ordenar  $a[1]$  y  $a[2]$
2. Comparar  $a[3]$  con  $a[2]$ , si  $a[3]$  es mayor o igual que  $a[2]$ , seguir con el próximo elemento del arreglo, sino, comparar  $a[3]$  con  $a[1]$  y si  $a[3]$  es mayor o igual que  $a[1]$ , insertar  $a[3]$  entre  $a[1]$  y  $a[2]$ . Si  $a[3]$  es menor que  $a[1]$ , entonces transferir  $a[3]$  a la posición de  $a[1]$ ,  $a[1]$  a la de  $a[2]$  y  $a[2]$  a la de  $a[3]$
3. Supongamos que de esta manera se han programado los m-1 primeros elementos del arreglo y corresponde insertar en el lugar apropiado el m-ésimo elemento. Si, por ejemplo,  $a[m]$  es mayor que  $a[k]$  (con  $k = 1, 2, \dots, m-1$ ), se deben correr una posición  $a[k+1], \dots, a[m-1]$  y almacenar  $a[m]$  en la posición  $k+1$ .

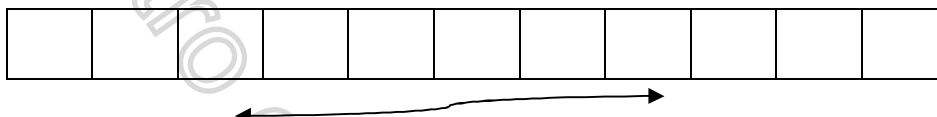




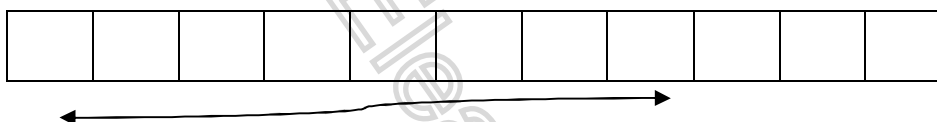


En el ejemplo se observa que cada pasada puede finalizar por dos condiciones distintas:

- Se encontró la posición donde debe insertarse el elemento que se está tratando, entre el primero y el anterior a él



- El arreglo debe insertarse delante del primero del arreglo



Esta terminación con dos condiciones puede evitarse usando la “técnica del centinela” que, en este caso, consiste en extender hacia la izquierda el arreglo definiendo un elemento  $A[0]$  para almacenar el que se está tratando.

## 2.3 Ordenamiento por intercambio (burbujeo)

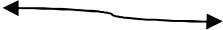





Estos métodos están basados en el principio de comparar e intercambiar pares de elementos adyacentes. Uno de los métodos de ordenación por intercambio consiste en comparar  $a[1]$  y  $a[2]$ , intercambiarlos si están desordenados, y luego hacer lo mismo con  $a[2]$  y  $a[3]$ ,  $a[3]$  y  $a[4]$  y así sucesivamente. Durante esta secuencia de operaciones, los elementos mayores tienden a moverse hacia la derecha, colocándose el elemento mayor en  $a[n]$ . La repetición de este proceso tiene como consecuencia que los elementos se colocan en las posiciones correctas  $a[n-1]$ ,  $a[n-2]$ , etc. Este método conocido como método de la burbuja o burbujeo consiste en:

Comparar el primer elemento con el segundo, intercambiarlos si están desordenados, luego se compara el segundo con el tercero, intercambiándose si fuera necesario, hasta llegar al último elemento. De esta manera en la última posición del arreglo ha quedado el elemento mayor.



- Si durante el paso anterior hubo cambio, se vuelve a repetir el proceso pero solo hasta una posición anterior a la del último elemento ubicado en su lugar correcto.
- Si durante una pasada no hubo cambios, el algoritmo se da por finalizado.

En el siguiente ejemplo, mostramos la forma en la cual los elementos se van ubicando en su posición correcta:

Inicial	21	35	17	8	14	42	2	26
								
Pasada 1	21	17	8	14	35	2	26	42
								
Pasada 2	17	8	14	21	2	26	35	42
								
Pasada 3	8	14	17	2	21	26	35	42
								
Pasada 4	8	14	2	17	21	26	35	42
								
Pasada 5	8	2	14	17	21	26	35	42
								
Pasada 6	2	8	14	17	21	26	35	42



```
programa OrdenBurbuja
inicio

    funcion principal ()
        var integer a[3]
        // relleno arreglo con valores fijos
        a[1] = 21
        a[2] = 35
        a[3] = 17
        burbuja (a[], 3)    // llamo a función burbuja y le envío
        // parámetros: el arreglo y la longitud
    fin funcion

    // se le delega el algoritmo de orden a esta función debajo
    funcion burbuja(integer a[], integer n)
        var integer cota = n
        var integer k = 1
        var integer i = 1
        var integer aux = 0
        mientras k <> 0
            k = 0
            mientras i <= cota - 1
                si a[i] > a[i+1] entonces
                    aux = a[i]
                    a[i] = a[i+1]
                    a[i+1] = aux
                    k = i
                fin si
                i = i + 1
            fin mientras
            cota = k
            i=1
        fin mientras

        // mostrar arreglo a ordenado

    fin funcion
fin
```



Las variables utilizadas fueron:

- "a": arreglo de tipo numérico entero que en las posiciones 1, 2, 3 y 4 contiene los elementos a ordenar
- "n": variable de tipo entero que representa la dimensión de "a"
- "cota": variable de tipo entero cuyo valor es el índice hasta donde se debe llegar en determinada pasada
- "i": variable entera, índice de "a"
- "aux": variable del mismo tipo que los elementos del arreglo "a", que se usa para realizar el intercambio
- "k": variable entera que contiene el valor del índice del elemento intercambiado en la pasada anterior a la actual. Si el valor de "k" es cero, significa que no se ha hecho intercambio en esa pasada

Este algoritmo en particular invierte mucho tiempo en la búsqueda del elemento mínimo entre los elementos que aún están desordenados en el vector, lo que implicaría un tiempo de proceso elevado si se ordenan varios arreglos de tamaño considerable. Por otro lado, es uno de los métodos más difundidos por su simpleza.



A continuación, realizaremos la comprobación de método visto:

- **Vuelta 1:**

El vector contiene los siguientes elementos, en el siguiente orden:

**21,35,17**

Con  $k = 1$ , cota = 3

mientras  $1 < 0$  Verdadero!

$k = 0$

mientras  $1 \leq 3-1$

si  $21 > 35$  entonces

Falso!

fin si

$i = 2$

fin mientras

mientras  $2 \leq 3-1$

si  $35 > 17$  entonces Verdadero!!

$aux = 35$

$a[2] = 17$

$a[3] = 35$

$k = 2$

fin si

$i = 3$

fin mientras

cota = 2

$i = 1$

fin mientras



- **Vuelta 2:**

El vector contiene los siguientes elementos, en el siguiente orden:

**21,17,35**

Con  $k = 2$ ,  $cota = 2$

mientras  $2 \neq 0$

$k = 0$

    mientras  $1 \leq 2-1$

        si  $21 > 17$  entonces

$aux = 21$

$a[1] = 17$

$a[2] = 21$

$k = 1$

        fin si

$i = 2$

fin mientras

$cota = 1$

fin mientras

- **Vuelta 3:**

El vector contiene los siguientes elementos, en el siguiente orden:

**17,25,35**

Con  $k = 1$ ,  $cota = 1$

mientras  $1 \neq 0$       Verdadero!

$k=0$

    mientras  $1 \leq 1-1$       Falso!

fin mientras

$cota = 0$

$i = 1$

fin mientras



- **Vuelta 4:**

El vector sigue con los siguientes elementos, en el siguiente orden:

**17,25,35**

mientras 0 <> 0

Falso!! El vector ya está ordenado

fin mientras

## **2.4 Intercalación**

Desarrollaremos un método de intercalación de dos arreglos ordenados, los cuales se van a combinar para producir un único arreglo también ordenando. Este proceso se realiza seleccionando sucesivamente los elementos con el mínimo valor de cada uno de los dos vectores, situándolos en un nuevo arreglo. De esta forma, el nuevo arreglo tiene todos sus elementos ordenados.

Por ejemplo, teniendo los dos arreglos siguientes:

**Arreglo 1: 9, 35, 41**

**Arreglo 2: 7, 20, 25, 43**

El procedimiento a seguir es el siguiente:



7	{	9	35	41		
		20	25	43		
7	9	{	35	41		
		20	25	43		
7	9	20	{	35	41	
			25	43		
7	9	20	25	{	35	41
				43		
7	9	20	25	35	{	41
					43	
7	9	20	25	35	41	43





### **3. Búsquedas**

La búsqueda se refiere a localizar un dato en particular dentro de un conjunto de datos. El dato a localizar puede o no estar entre los elementos del conjunto. La búsqueda de un elemento dado es una tarea muy frecuente, como por ejemplo, localizar por nombre y apellido un alumno de un curso. Debido a la importancia del tema, se han desarrollado varios algoritmos, en los cuales la eficiencia está dada por la velocidad con que un dato es encontrado.

Formalmente el problema de la búsqueda es:

“Dado un conjunto de  $n$  elementos distintos y un dato  $k$  llamado argumento, determinar:

- si  $k$  pertenece al conjunto y en ese caso indicar cuál es su ubicación en el mismo;
- si  $k$  no pertenece al conjunto”.

#### **3.1 Búsqueda secuencial**

La manera más obvia para encontrar un argumento  $k$  dentro de un conjunto es comparar  $k$  con cada elemento hasta que éste sea encontrado, o bien, se haya recorrido todo el conjunto.

En arreglos desordenados se utiliza siempre  $n+1$  (donde  $n$  es la cantidad de elementos a comparar) comparaciones para una búsqueda sin éxito, y alrededor de  $n/2$  comparaciones (como término medio) para una búsqueda con éxito.

En arreglos ordenados se utiliza  $n/2$  comparaciones (por término medio) para búsquedas con o sin éxito.



```
...  
funcion busquedaSecuencial()  
    mientras i <= n  
        si k = a[i] entonces  
            mostrar: "Elemento encontrado en posición "  
            mostrar: i  
            i = n + 2    // fuerza el final del ciclo  
        sino  
            i = i + 1  
        fin si  
    fin mientras  
  
    si i = (n + 1) entonces  
        mostrar: "No se encontró el elemento"  
    fin si  
fin funcion  
...
```

En este ejemplo tenemos las siguientes variables definidas:

- a: arreglo de n posiciones
- k: variable del mismo tipo que los elementos de "a", cuyo valor es el elemento a buscar
- n: variable entera (integer), que representa el tamaño de "a"
- i: variable integer, utilizada como índice de "a"



## 3.2 Búsqueda binaria

Supongamos que todos los elementos del arreglo han sido ordenados previamente, por ejemplo, en orden ascendente.

El método de búsqueda binaria consiste en localizar aproximadamente la posición media del arreglo y examinar el valor allí encontrado.

Si ese valor es mayor que el buscado, entonces se busca el argumento en la primera mitad del arreglo, repitiéndose este proceso en la mitad correspondiente del mismo, hasta que se encuentre el elemento deseado. Si, por el contrario, el valor es menor que el buscado, se prosigue con la segunda mitad del arreglo, localizándose el elemento situado en el centro de ella y continuando con el procedimiento hasta haber encontrado el elemento buscado, o bien, hasta que el intervalo de búsqueda haya quedado vacío.

Después de comparar  $k$  con  $a[i]$  existen tres posibilidades:

- $k < a[i]$ : los elementos de  $a[i]$ ,  $a[i+1]$ ...,  $a[n]$  quedan eliminados de la búsqueda
- $k = a[i]$ : la búsqueda ha terminado (con éxito)
- $k > a[i]$ : los elementos  $a[1]$ ,  $a[2]$ ...,  $a[i]$  quedan eliminados de la búsqueda

Por ejemplo: teniendo el siguiente conjunto de ocho elementos ordenados en forma ascendente:

**2, 8, 14, 17, 21, 26, 35, 42**



Caso 1: el usuario ingresa para buscar el número 35.

- El elemento central de este arreglo es el número 17 (la posición se obtiene como la parte entera de  $(1+8)/2$ )
- Como 35 es mayor que 17, quedan eliminados los valores 2, 8, 14 y 17
- Se debe examinar la segunda parte del arreglo: desde  $a[5]$  a  $a[8]$
- El elemento central de este sub intervalo es 26 (la posición se obtiene como la parte entera de  $(5+8)/2$ )
- Como 35 es mayor a 26, deben examinarse los elementos que están a su derecha, es decir,  $a[7]$  y  $a[8]$
- El elemento central de  $a[7]$  y  $a[8]$  es  $a[7]$

Como el valor de  $a[7]$  es 35, la búsqueda terminó con éxito

Caso 2: el usuario ingresa para buscar el número 13

- Comenzamos comparando 13 con el elemento central del arreglo
- Como en este caso el elemento buscado resulta menor que  $a[4]$ , se procede a examinar los elementos  $a[1]$ ,  $a[2]$  y  $a[3]$ .
- El valor 13 es mayor que  $a[2]$  (elemento central de la primer mitad del arreglo) y por consiguiente, se busca en la porción del arreglo que consiste solo de  $a[3]$ .
- El valor buscado es distinto a  $a[3]$  y se concluye que el dato no pertenece al conjunto



```
...  
funcion busquedaBinaria()  
    var integer min = 1  
    var integer max = n  
    var integer medio = (n+1) / 2  
  
    mientras min <= max y k <> a[medio]  
        si k < a[medio] entonces  
            max = medio - 1  
        sino  
            min = medio + 1  
        fin si  
        medio = (min + max) / 2  
    fin mientras  
  
    si k = a[medio] entonces  
        mostrar: "El valor se encontró en "  
        mostrar: medio  
    sino  
        mostrar: "El valor no se encontró"  
    fin si  
fin funcion  
...
```



Las variables utilizadas son:

- a: arreglo de n posiciones
- k: variable del mismo tipo que los elementos de "a", cuyo valor es el elemento a buscar
- n: variable entera (integer), que representa el tamaño de "a"
- min: variable integer cuyo valor es el valor inferior del índice en un paso
- max: variable integer cuyo valor es el valor superior del índice en un paso
- medio: variable de tipo integer cuyo valor es la parte entera de  $(\min + \max) / 2$



## Lo que vimos

- Concepto de estructura de datos
- Estructuras de datos principales: listas (lineales y simples), pilas y colas
- Métodos de ordenamiento por selección, por inserción, por intercambio y por intercalación
- Métodos de búsqueda secuencial y búsqueda binaria



---

## Lo que viene:

- Los primeros conceptos del paradigma de Programación Orientada a Objetos (POO)
- Los conceptos de clase, objeto, métodos y atributos
- Los conceptos de los dos primeros principios que hacen a la Orientación a Objetos (OO): abstracción y encapsulamiento

