

# Tarea 3

## Manejo dinámico de estructuras lineales y arborescentes

### Curso 2019

## Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Materiales</b>	<b>2</b>
<b>3. ¿Qué se pide?</b>	<b>2</b>
3.1. Verificación de la implementación . . . . .	3
<b>4. Descripción de los módulos y algunas funciones</b>	<b>3</b>
4.1. Módulo <i>Binario</i> . . . . .	3
4.1.1. <i>insertar_en_binario</i> (info_t i, binario_t b) . . . . .	3
4.1.2. <i>remover_de_binario</i> (texto_t t, binario_t b) . . . . .	3
4.1.3. <i>altura_binario</i> (binario_t b) . . . . .	4
4.1.4. <i>buscar_subarbol</i> (texto_t t, binario_t b) . . . . .	4
4.1.5. <i>imprimir_binario</i> (binario_t b) . . . . .	4
4.1.6. <i>menores</i> (int clave, binario_t b) . . . . .	5
4.1.7. <i>suma_ultimos_pares</i> (nat i, binario_t b) . . . . .	6
4.1.8. <i>es_AVL</i> (binario_t b) . . . . .	6
4.1.9. <i>es_camino</i> (binario_t b) . . . . .	7
4.1.10. <i>cadena_a_binario</i> (cadena_t c) . . . . .	8
<b>5. Entrega</b>	<b>9</b>
5.1. Plazos de entrega . . . . .	9
5.2. Identificación de los archivos de las entregas . . . . .	9
<b>6. Funciones auxiliares</b>	<b>10</b>
<b>7. Material Complementario</b>	<b>10</b>

## 1. Introducción

En la presente tarea continuaremos trabajando sobre el manejo dinámico de memoria, incorporando estructuras arborescentes. Trabajaremos con **árboles binarios de búsqueda**. Los elementos de los árboles son de tipo `info_t`, el cual ya fue utilizado en la tarea anterior.

Se agregaron restricciones en el tiempo de ejecución para algunas operaciones de los diferentes módulos. Esto puede requerir modificaciones en las implementaciones realizadas en tareas anteriores, y así garantizar que la solución a entregar cumpla con dichas restricciones.

Como corresponde al concepto de TAD no se puede usar la representación de un tipo fuera de su propio módulo. Por ejemplo, en `uso_tads.cpp` no se puede usar `rep_cadena` sino las operaciones declaradas en `cadena.h`.

El correcto uso de la memoria y el cumplimiento de las restricciones de tiempo será parte de la evaluación.

El resto del presente documento se organiza de la siguiente forma. En la Sección 2 se presenta una descripción de los materiales disponibles para realizar la presente tarea, y en la Sección 3 se detalla el trabajo a realizar. Luego, en la Sección 4 se explica mediante ejemplos el comportamiento esperado de algunas de las funciones a implementar. Por último, la Sección 5 describe el formato y mecanismo de entrega, así como los plazos para realizar la misma.

## 2. Materiales

Los materiales para realizar esta tarea se encuentran en el archivo *MaterialesTarea3.tar.gz* que se obtiene en la carpeta *Materiales* de la sección *Laboratorio* del sitio EVA del curso <sup>1</sup>.

A continuación se detallan los archivos que se entregan. La estructura de directorios es igual a la de las tareas anteriores.

- **info.h**: módulo de definición del tipo `info_t`
  - **cadena.h**: módulo de definición de cadenas
  - **binario.h**: módulo de definición de árbol binario de búsqueda
  - **uso\_cadena.h**: módulo de definición de funciones sobre cadenas
  - Casos de prueba en el directorio **test**
  - **Makefile**: archivo para usar con el comando `make`, que provee las reglas `principal`, `testing`, `clean` y entrega como en la tarea anterior.
  - **principal.cpp**: módulo principal.
  - En el directorio *src* se entrega además el archivo *ejemplo\_binario\_cpp.png* que contiene la imagen de la implementación de algunos tipos y funciones que se piden.
- El contenido de este archivos **puede** copiarse en los archivos a implementar.

**Ninguno de estos archivos deben ser modificados.**

## 3. ¿Qué se pide?

Para cada archivo de encabezamiento **.h**, descrito en la Sección 2, se debe implementar un archivo **.cpp** que implemente **todas** las definiciones de tipo y funciones declaradas en el archivo de encabezamiento correspondiente. Los archivos **cpp** serán los entregables y deben quedar en el directorio **src**.

---

<sup>1</sup><https://eva.fing.edu.uy/course/view.php?id=132&section=6>

### 3.1. Verificación de la implementación

Para testear la implementación debe generar el ejecutable y cumplir con todas las pruebas utilizando los archivos **.in** y **.out**. Tenga en cuenta que el archivo **Makefile** proveerá las reglas **principal**, **testing** y **clean** como en las tareas anteriores.

## 4. Descripción de los módulos y algunas funciones


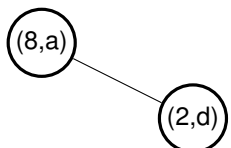
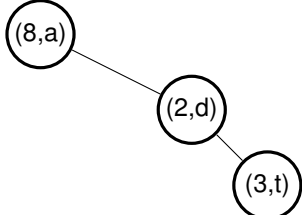
En esta sección se explica, mediante ejemplos, el comportamiento de algunas funciones representativas de los módulos a implementar. Se utilizará una notación gráfica para representar árboles, donde el árbol vacío se representa con el símbolo **●**. Los elementos de tipo **info\_t** se representan con el formato definido en la función **info\_a\_texto**.

### 4.1. Módulo *Binario*

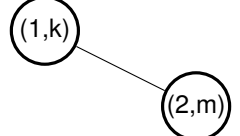

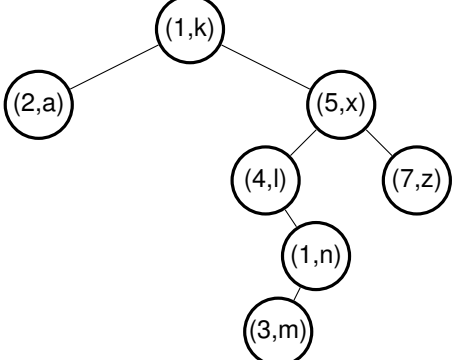
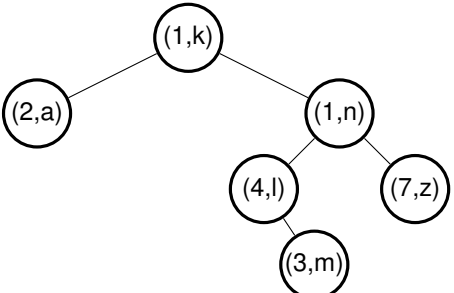
En este módulo se implementará un árbol binario de búsqueda (de tipo **binario\_t**) de elementos de tipo **info\_t**. La propiedad de orden de los árboles es definida por el campo **texto** de sus elementos.

A continuación presentaremos mediante ejemplos algunas funciones representativas del módulo **binario**.


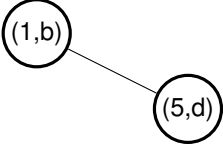
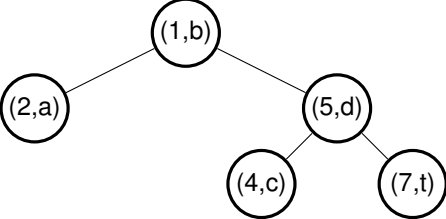
#### 4.1.1. *insertar\_en\_binario*(**info\_t i**, **binario\_t b**)

i	b	Resultado luego de la ejecución
(1,u)	●	
(3,t)		

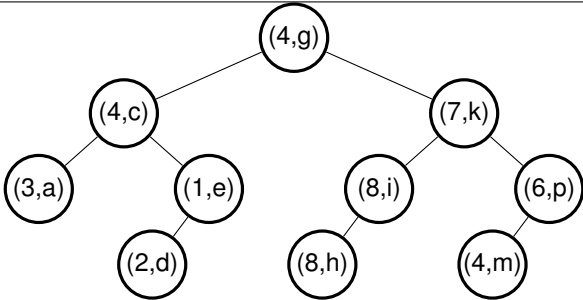
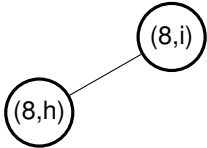
#### 4.1.2. *remover\_de\_binario*(**texto\_t t**, **binario\_t b**)

t	b	Resultado luego de la ejecución
m		
x		


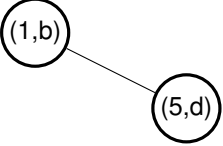
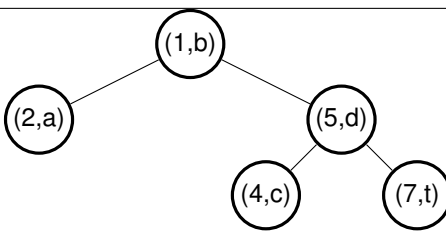
4.1.3. altura\_binario(binario\_t b)

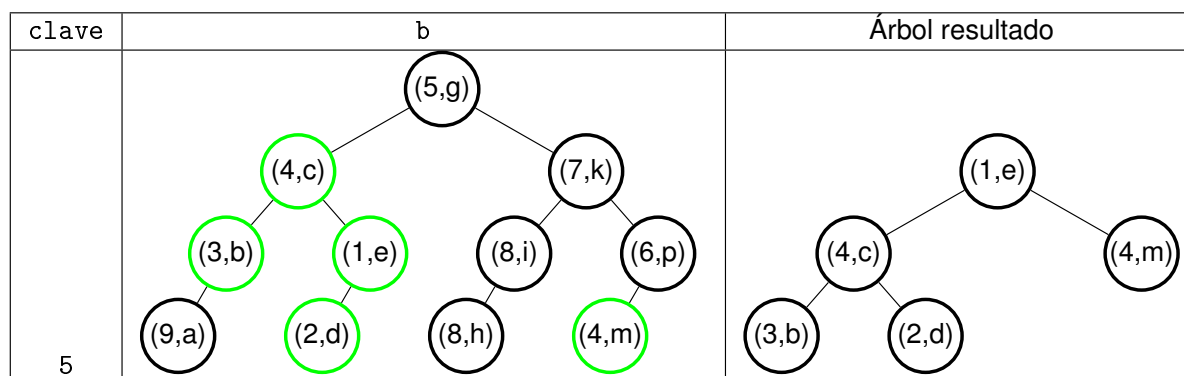
b	Altura de b
	1
	2
	3

4.1.4. buscar\_subarbol(texto\_t t, binario\_t b)

t	b	Árbol resultado
i		

4.1.5. imprimir\_binario(binario\_t b)

b	Salida esperada
	( 1 , b )
	- ( 5 , d ) ( 1 , b )
	-- ( 7 , t ) - ( 5 , d ) -- ( 4 , c ) ( 1 , b ) - ( 2 , a )

4.1.6. *menores(int clave, binario\_t b)*

Cuadro 1: Ejemplo de llamada a la función menores

En el Cuadro 1 se muestra un ejemplo de llamada a la función `menores`. En dicho ejemplo se destacan en verde los nodos en donde se cumple la condición, cuyas copias serán los nodos del árbol resultado.

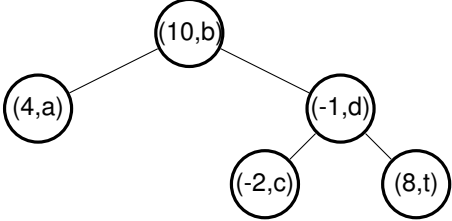
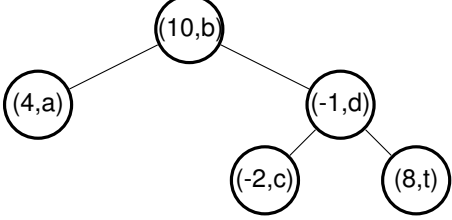
El resultado de la operación en el árbol que tenga como raíz uno de los nodos que cumplen con la condición, es un árbol que en cuya raíz está una copia de ese nodo. Los dos subárboles son el resultado de la operación aplicada en los hijos de ese nodo.

En los nodos cuyos elementos son (9,a) y (8,h) no se cumple la condición y no tienen descendientes. En (8,i) no se cumple y tampoco se cumple en sus descendientes. Por lo tanto el resultado del filtrado en cualquiera de esos nodos es el árbol vacío.

En (6,p) no se cumple la condición y no hay nodos en su subárbol derecho. El resultado en este nodo es, entonces, el resultado en su hijo izquierdo, que es el árbol cuyo único nodo es una copia de (4,m), por ser el único nodo que cumple con la condición en el subárbol izquierdo. De manera análoga, ni en (7,k) ni en su subárbol izquierdo hay nodos que cumplan la condición, por lo que el resultado en (7,k) es el resultado en su hijo derecho.

En la raíz del árbol, (5,g), no se cumple la condición pero en sus dos subárboles hay nodos que la cumplen. En el resultado la raíz debe ser la copia de algún otro nodo y el árbol devuelto debe mantener la propiedad de orden. Se elige como raíz una copia del nodo mayor (según el orden definido) de los que cumplen la condición en el subárbol izquierdo, que es (1,e). Este nodo debe removerse del resultado en (4,c) para pasar a ser la raíz del árbol resultado.

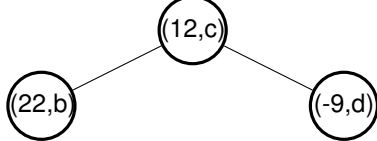
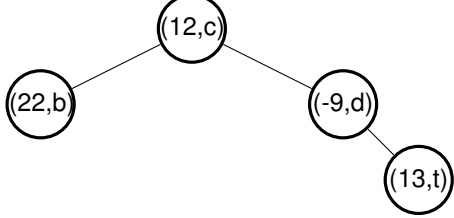
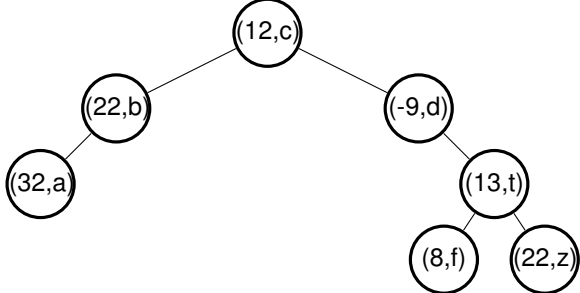
**4.1.7. suma\_ultimos\_pares(nat i, binario\_t b)**

i	b	Resultado esperado
1	●	0
0	(10,b)	0
1	(10,b)	10
2	(10,b)	10
2		6
4		20

**4.1.8. es\_AVL(binario\_t b)**

Los árboles AVL son un tipo particular de ABB que están siempre equilibrados en el sentido de que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha. Debido a esta propiedad, el orden de búsqueda en el árbol es siempre  $O(\log(n))$ .

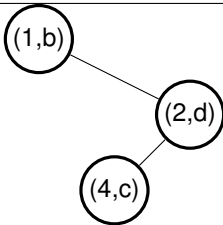
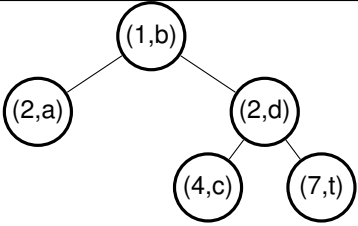
En esta ocasión se les solicita evaluar si dado un ABB, el mismo cumple con la propiedad AVL, es decir, verificar si el valor absoluto de la diferencia de las alturas de los subárboles izquierdo y derecho de **cada** nodo es menor o igual a 1.

b	Resultado esperado
●	true
	true
	true
	false

Se debe notar que:

- El árbol vacío se considera balanceado.
- En el cuarto ejemplo de la tabla, a pesar de que la raíz está balanceada, el resultado de la llamada a la función es `false` porque el nodo `(-9,d)` no está balanceado ya que su subárbol izquierdo tiene altura 0 mientras que su subárbol derecho tiene altura 2.

#### 4.1.9. `es_camino(binario_t b)`

l	b	Resultado esperado
[]	●	true
[(4,b)]	(1,b)	true
[(4,b), (5,d)]	(1,b)	false
[(4,b), (5,d)]		false
[(4,b), (5,d), (3,c)]		true

#### 4.1.10. *cadena\_a\_binario(cadena\_t c)*

Con esta operación se crea un árbol balanceado. Se debe notar que el criterio para decir que un árbol está balanceado es diferente al usado en `es_AVL`.

Aquí se fundamenta (sin demostrar) la cota de tiempo  $O(n \cdot \log n)$  requerida en la ejecución de `cadena_a_binario` en `binario.h`. Esta fundamentación muestra que es posible alcanzar la cota requerida.

Es importante recordar que la entrada está ordenada y no tiene elementos repetidos, y que el árbol es binario de búsqueda.

Teniendo en cuenta que la cantidad de niveles de un árbol balanceado de  $n$  nodos es  $O(\log n)$  (ver expresión (1)), se puede llegar a la cota solicitada si en la implementación se logra que el tiempo de procesamiento dedicado a cada nivel sea  $O(n)$ . Esto no significa que la implementación debe hacerse “por niveles”, sino que, para cada nivel, la suma de los tiempos dedicados a sus nodos sea  $O(n)$ . Y, a su vez, esto se puede lograr si el tiempo dedicado al procesamiento de cada nodo es lineal respecto al tamaño de su entrada.

Para simplificar, supongamos que la cantidad de elementos de la entrada,  $n$ , genera un árbol perfecto de  $h$  niveles (un árbol es perfecto si todas las hojas están en el último nivel). En el nivel  $i$ -ésimo, la cantidad de nodos es  $2^{i-1}$  (se considera que la raíz está en el nivel 1). Esto implica que la cantidad de nodos es

$$n = 1 + 2 + 2^2 + \dots + 2^{h-1} = \sum_{i=1}^h 2^{i-1} = 2^h - 1, \quad (1)$$

de donde  $h = \log(n+1)$  y por lo tanto  $h = O(\log n)$ .

El tiempo de ejecución de un algoritmo que implementa de manera eficiente lo solicitado es:

$$\begin{aligned} & O(n) + 2 \cdot O(n/2) + 4 \cdot O(n/4) + \dots + 2^{h-1} \cdot O(n/2^{h-1}) \\ &= O(n(1 + 2^1 \cdot 1/2^1 + 2^2 \cdot 1/2^2 + \dots + 2^{h-1} \cdot 1/2^{h-1})) \\ &= O(n \cdot h) \\ &= O(n \cdot \log n). \end{aligned}$$

Esto se justifica gráficamente en la figura 1, en la que en cada nodo se muestra el tamaño de la entrada que debe procesar, que es igual a la cantidad de nodos del subárbol del que es raíz.

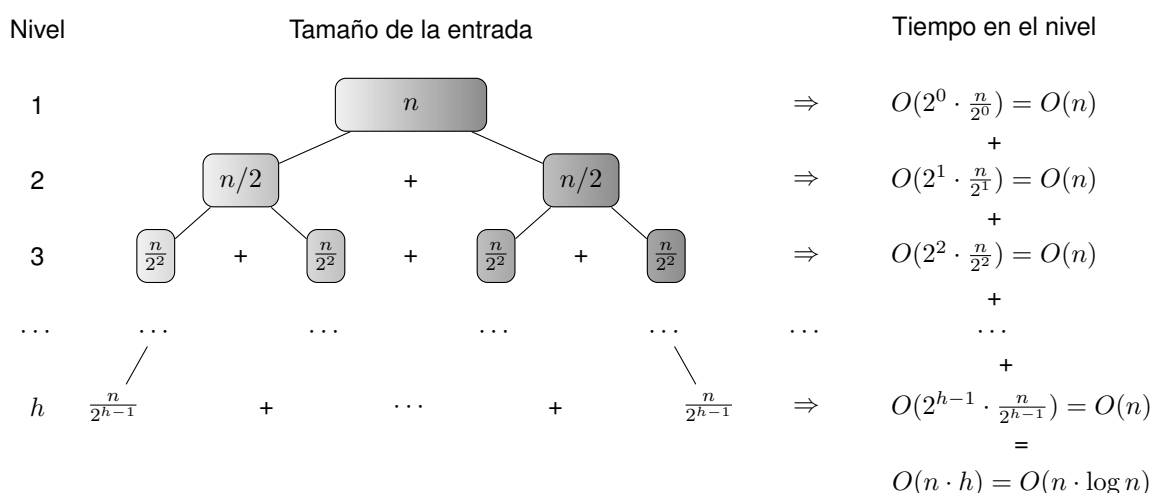


Figura 1: **Crear un árbol balanceado.** En cada nodo se muestra el tamaño de la entrada que se debe tratar. Si el procesamiento del nodo se hace en tiempo lineal respecto a ese tamaño, el procesamiento de todo el nivel se hace en tiempo  $O(n)$ , y el de todo el algoritmo en tiempo  $O(n \cdot \log n)$ .



## 5. Entrega

Se mantienen las consideraciones reglamentarias y de procedimiento de la tarea anterior.

En particular, la tarea otorga dos puntos a los grupos que la aprueben en la primera instancia de evaluación y uno o cero a los que aprueben en la segunda instancia. La adjudicación de los puntos de las tareas de laboratorio queda condicionada a la respuesta correcta de una pregunta sobre el laboratorio en el segundo parcial. La adjudicación se aplica de manera individual a cada estudiante del grupo.

La tarea es **eliminatória**.

Se debe entregar el siguiente archivo, que contiene los módulos implementados *info.cpp*, *cadena.cpp*, *uso\_cadena.cpp* y *binario.cpp*:

### ■ Entrega3.tar.gz

Este archivo se obtiene al ejecutar la regla entrega del archivo *Makefile*:

```
$ make entrega
tar zcvf Entrega3.tar.gz -C src info.cpp cadena.cpp uso_cadena.cpp binario.cpp
info.cpp
cadena.cpp
uso_cadena.cpp
binario.cpp
```

Con esto se empaquetan los módulos implementados y se los comprime.

**Nota:** En la estructura del archivo de entrega los módulos implementados deben quedar en la raíz, NO en el directorio *src* (y por ese motivo se usa la opción *-C src* en el comando *tar*).

**NO SE PUEDEN ENTREGAR MÓDULOS ADICIONALES A LOS SOLICITADOS**

### 5.1. Plazos de entrega

El plazo para la entrega es el **miércoles 24 de abril a las 14 horas**.

**NO SE ACEPTARÁN ENTREGAS DE TRABAJOS FUERA DE FECHA Y HORA. LA NO ENTREGA O LA ENTREGA FUERA DE LOS PLAZOS INDICADOS IMPLICA LA PÉRDIDA DEL CURSO.**

### 5.2. Identificación de los archivos de las entregas

Cada uno de los archivos a entrega debe contener, en la primera línea del archivo, un comentario con el número de cédula de el o los estudiantes, **sin el guión y sin dígito de verificación**. Ejemplo:

```
/* 1234567 - 2345678 */
```

## 6. Funciones auxiliares

Para la implementación de algunas de las funciones de esta tarea podrían necesitarse funciones auxiliares. En C no se puede definir una función anidada dentro de otra como se puede hacer en Pascal. Por lo tanto la función auxiliar debe definirse fuera de las funciones en las que se va a usar. Además, la declaración debe estar antes de que la función sea usada.

Esto puede llegar a generar un conflicto en el momento del enlazado si en otros archivos también hubiera funciones auxiliares con la misma firma ya que en C las funciones de manera predeterminada tienen ámbito global. Para evitar este problema se debe anteponer la palabra reservada `static` a la declaración de la función. Con esto queda como local al archivo. Su ámbito va desde la declaración hasta el fin del archivo.

```
// Declaración de 'auxiliar' usada en 'f' y 'g'.
// Debe preceder a la implementación de las funciones que la usan.
static int auxiliar(int p);
// La implementación se puede hacer junto con la declaración
// o después.

// f y g usan 'auxiliar'
void f() {
    ...
    int a = auxiliar(5);
    ...
}

void g() {
    ...
    int a = auxiliar(8);
    ...
}

// implementación de auxiliar
static int auxiliar(int p) {
    ...

    ...
    return p + ...;
}
```

## 7. Material Complementario

En la sección de laboratorio se encuentra el documento Material Complementario, el cual contiene la información referente a Makefile, Control de manejo de memoria, Assert y Método sugerido.