

Práctico 4

Objetivos

- Trabajar sobre estructuras arborescentes en memoria dinámica.
- Aplicar técnicas de recursión sobre árboles.
- Trabajar con funciones y procedimientos tanto totales, como parciales (que contemplan precondiciones).
- Fijar conceptos relacionados con árboles y algunas variantes, como ser: **árboles Binarios, árboles Binarios de Búsqueda y árboles Finitarios o Generales.**

En este práctico se asume definido el tipo para enteros sin signo (naturales):

```
typedef unsigned int uint;
```

Primera parte: árboles binarios

Ejercicio 1

Considere la representación para un árbol binario de Naturales de la Figura 1.

```
struct nodoAB {  
    uint elem;  
    nodoAB *izq, *der;  
};  
typedef nodoAB * AB;
```

Figura 1: Definición del tipo árbol binario de naturales.

- ¿Cómo representaría al árbol vacío con dicha representación?
- Utilice la representación dada para implementar las siguientes operaciones:
 - función `consArbol`: que crea un árbol no vacío a partir de un natural y otros dos árboles.
 - función `contarElems`: que recibe un árbol y retorna la cantidad de elementos que tiene el mismo.
 - función `altura`: que recibe un árbol y retorna la altura del mismo. Si el árbol es vacío su altura es 0.
 - función `contarHojas`: que recibe un árbol y retorna la cantidad de hojas (nodos cuyos ambos subárboles son vacíos) del mismo.
 - función `copiar`: que recibe un árbol y crea una *copia limpia* (que no comparte registros de memoria) del mismo.
 - procedimiento `liberarArbol`: que recibe un árbol y elimina del mismo todos los nodos, liberando la memoria asociada a cada uno de ellos. El árbol se transforma en un árbol vacío.
- La estructura que devuelve su solución de `consArbol`: ¿comparte memoria con los parámetros? En caso afirmativo: ¿qué problemas puede acarrear esto?

Ejercicio 2

Considere la representación para árbol binario de Naturales de la Figura 1 y la siguiente representación de Lista de Naturales:

```
struct nodoLista {  
    uint elem;  
    nodoLista *sig;  
};  
typedef nodoLista * Lista;
```

- (a) Utilícelas para implementar las siguientes funciones:
- I. `enOrden`: que recibe un árbol a y retorna una lista con los elementos de a ordenados según la recorrida en orden de a
 - II. `preOrden`: que recibe un árbol a y retorna una lista con los elementos de a ordenados según la recorrida en pre orden de a
 - III. `postOrden`: que recibe un árbol a y retorna una lista con los elementos de a ordenados según la recorrida en post orden de a
 - IV. `esCamino`: que recibe un árbol a y una lista l , y retorna TRUE si y sólo si l es un camino, desde la raíz a una hoja, de a
 - V. `caminoMasLargo`: que recibe un árbol a y retorna una lista con los elementos del camino más largo de a (desde la raíz a una hoja). En caso de haber más de un camino de igual longitud a la del camino más largo, retorna cualquiera de ellos.
- (b) ¿Cuántos nodos tiene como mínimo y cómo máximo el camino más largo desde la raíz a una hoja, para un árbol binario de n nodos? Justifique.
- (c) ¿Cuántos nodos tiene un árbol binario completo (perfectamente balanceado) de altura h ? Escriba una función booleana que dados un árbol binario a y un natural h , retorne TRUE si, y sólo si, a es un árbol completo de altura h . Implemente dicha función sin usar operaciones auxiliares para calcular la cantidad de nodos o la altura de un árbol.

Segunda parte: árboles binarios de búsqueda

Considere la representación para un árbol binario de búsqueda (ABB) de Naturales de la Figura 2 .

```
struct nodoABB {
    uint elem;
    nodoABB * izq, * der;
};
typedef nodoABB * ABB;
```

Figura 2: Definición del tipo árbol binario de búsqueda de naturales.

Ejercicio 3

(a) Utilice la representación dada para implementar las siguientes operaciones:

- I. procedimiento `insertarABB`: que recibe un natural x , y un ABB a , e inserta x en a manteniendo su cualidad de árbol binario de búsqueda. Si x pertenece al árbol, la operación no tiene efecto.
- II. función `perteneceABB`: que recibe un natural x , y un ABB a y devuelve `true` si y solo si x es un elemento del árbol a .
- III. función `maxABB`: que recibe un ABB no vacío a y devuelve el elemento de máximo valor en a .
- IV. procedimiento `removeMaxABB`: que recibe un ABB no vacío a y elimina el elemento de máximo valor en a .
- V. procedimiento `removeABB`: que recibe un natural x , y un ABB a , y elimina el elemento de valor x de a , manteniendo su cualidad de ABB.
- VI. función `k-esimo`: que recibe un natural k , y un ABB a y retorna el subárbol que tiene al k -ésimo menor elemento de a como raíz, si éste existe. Si no hay al menos k elementos en a o k es cero, la función debe retornar el árbol vacío. Si k es 1, se refiere al menor elemento del árbol, si k es 2 al 2do elemento más pequeño del árbol y así sucesivamente. La solución no puede visitar cada nodo mas de una vez.

Ejercicio 4 Filtrado

Sea ABB un tipo que representa árboles binarios de búsqueda cuyos elementos son del tipo `EstInfo`. `EstInfo` representa a un estudiante a partir de una identificación (`ci`) y la nota obtenida en un curso (`nota`). Los nodos de ABB están ordenados según el campo `ci`. En la figura siguiente se presenta a continuación la estructura de `EstInfo` y de ABB.

```
struct EstInfo {
    uint nota; //dato
    int ci;    //clave
};

struct nodoABB {
    EstInfo info ;
    nodoABB * izq , * der ;
};

typedef nodoABB * ABB ;
```

Se dispone de las operaciones `maxABB` y `removeMaxABB` definidas en el ejercicio anterior.

Se debe implementar la función `filtrado`, sin definir procedimientos auxiliares, para obtener un nuevo árbol solo con los estudiantes que superen una determinada nota:

```
/* Devuelve un árbol con los elementos de "a" en los que "nota" es ←  
   mayor que "cota". */  
ABB filtrado (ABB a; uint cota);
```

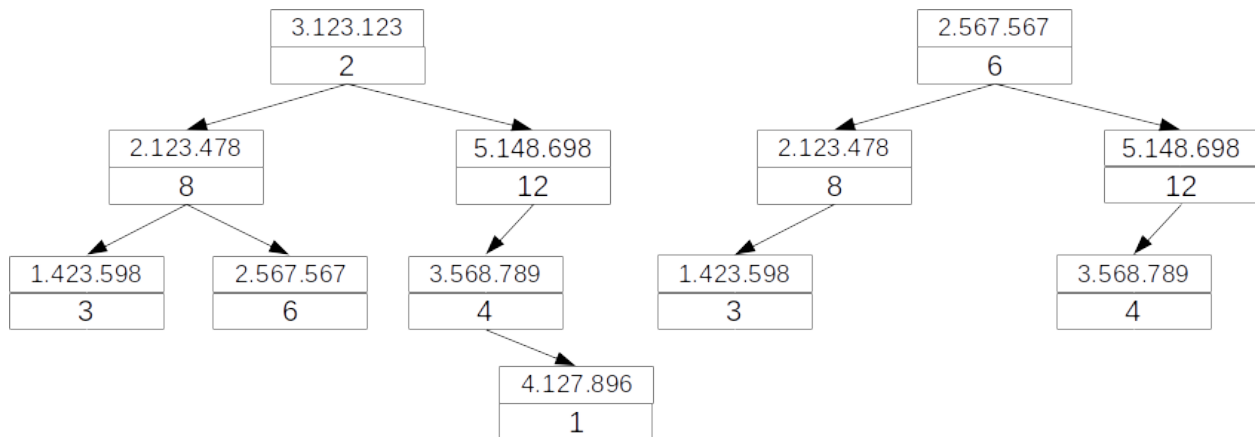


Figura 7: A modo de ejemplo, se presenta un árbol *a* (sobre la izquierda) y el resultado de filtrarlo con una cota igual a la nota 2 (sobre la derecha).

Tercera parte: árboles generales

Considere la siguiente definición del tipo AG de árboles generales o finitarios de enteros representados con árboles binarios con la semántica: primer hijo (pH) – siguiente hermano (sH):

```
struct nodoAG {  
    int dato;  
    nodoAG * pH;  
    nodoAG * sH;  
};  
typedef nodoAG* AG;
```

Ejercicio 5

Se quieren implementar las siguientes operaciones sobre árboles generales (representados con la semántica primer hijo, siguiente hermano) no vacíos y sin elementos repetidos de enteros:

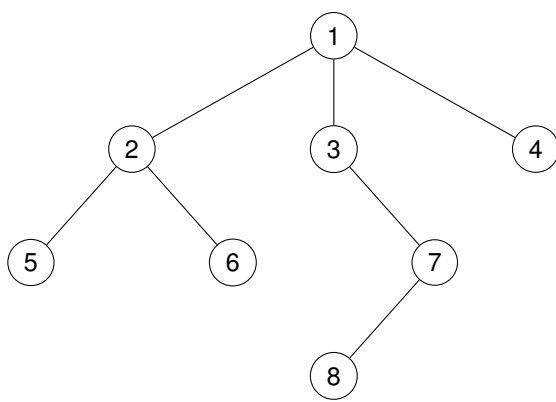
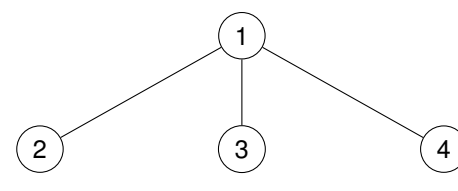
- (a) **arbolHoja**: Dado un entero x genera un árbol que sólo contiene a x (como una hoja).
- (b) **insertar**: Dados un árbol y dos enteros h y p , inserta a h como el primer hijo de p en el árbol (hijo más a la izquierda), siempre que p pertenezca al árbol y h no pertenezca al árbol. En caso contrario, la operación no tiene efecto.
- (c) **esArbolHoja**: Dado un árbol, retorna true si, y sólo si, el árbol es un árbol hoja (con un solo elemento).
- (d) **pertenece**: Dados un árbol y un entero x , retorna true si y sólo si x pertenece al árbol.
- (e) **borrar**: Dados un árbol y un entero x , elimina a x del árbol siempre que éste pertenezca al árbol, no sea la raíz del mismo y no tenga ningún hijo. En caso contrario, la operación no tiene efecto.
- (f) **borrarSub**: Dados un árbol y un entero x , elimina a x del árbol, siempre que éste pertenezca al árbol y no sea la raíz del mismo. En caso contrario, la operación no tiene efecto. Al eliminar el elemento deberán borrarse todos los elementos que están en el subárbol dependiente de éste.

Ejercicio 6 Primer Parcial 2017

Defina una función recursiva **copiaParcial** que dados un árbol a de tipo AG y un entero positivo k , retorne una copia de a , sin compartir memoria con éste, con todos los nodos que están en un nivel menor o igual a k . Si a es vacío o k es cero, el resultado debe ser el árbol vacío. Tenga en cuenta que en un árbol no vacío la raíz está en el nivel 1 y asuma que $a \rightarrow sH$ es NULL. No use operaciones auxiliares propias en la implementación de **copiaParcial**.

```
AG copiaParcial (AG a, uint k);
```

Ejemplo:

Entrada	Resultado
<p>$k = 2$</p> <p>a</p>  <pre> graph TD 1((1)) --- 2((2)) 1 --- 3((3)) 1 --- 4((4)) 2 --- 5((5)) 2 --- 6((6)) 3 --- 7((7)) 3 --- 8((8)) </pre>	 <pre> graph TD 1((1)) --- 2((2)) 1 --- 3((3)) 1 --- 4((4)) </pre>

Ejercicio 7

Defina una función recursiva **esPrefijo** que dada una lista de enteros y un árbol general de enteros, retorne true si y sólo si la lista es un prefijo de algún camino del árbol general, comenzando desde la raíz. La lista vacía es prefijo de cualquier camino (incluso del camino vacío, si el árbol es vacío). No se permite usar funciones o procedimientos auxiliares en este ejercicio. Utilice la definición de lista presentada en el ejercicio 2.

Ejercicio 8

Se quiere representar una estructura de directorios (carpetas) de un sistema operativo, donde cada directorio tiene un nombre único que lo identifica y posee una lista de archivos. Cada directorio puede contener un número finito de subdirectorios y una lista de archivos, donde cada archivo tiene un nombre y un contenido (ambos de tipo char*).

Considere el tipo Directorios, definido como árboles generales de Archivos e implementado como árboles binarios con la semántica primer hijo (pH) – siguiente hermano (sH):

```

struct nodoArchivo{
    char *nombreArchivo;
    char *contenidoArchivo;
    nodoArchivo *sig;
};
typedef nodoArchivo * Archivos;

struct nodoDirectorio{
    char *nombreDirectorio;
    Archivos listaArchivos;
    nodoDirectorio *pH, *sH;
};
typedef nodoDirectorio * Directorios;

```

Defina un procedimiento borrar que, dados un directorio D sin elementos repetidos y el nombre nom_dir de un directorio, elimine a nom_dir de D si nom_dir está en D y no tiene subdirectorios. En caso contrario, el procedimiento no tendrá efecto. En particular, si D es el directorio vacío (NULL) el procedimiento no tendrá efecto. Al eliminar un directorio deberá liberarse toda la memoria asociada a éste, incluyendo la correspondiente a los archivos que contenga. Utilice la función strEq (que se asume implementada) para comparar strings; strEq retorna true si y sólo si dos strings son iguales.

```
void borrar (Directorios &d, char *nom_dir);
```

Ejercicio 9

Implemente una función que retorne la amplitud del nodo del árbol de mayor amplitud. La amplitud de un nodo se define como la cantidad de hijos (directos) que tiene. Si el árbol *a* es vacío o la raíz no tiene hijos, la función debe retornar 0.

```
int mayorAmplitud(AG a)
```

Ejercicio Complementario

Ejercicio 10

Una forma común de almacenamiento de conjuntos de palabras consiste en la utilización de árboles generales de caracteres, ya que al evitar la duplicación de prefijos comunes permiten disminuir la cantidad de información almacenada. Esto se realiza de la siguiente forma:

1. Se deja la raíz del árbol sin información (nodo dummy).
2. Se colocan las palabras desde el segundo nivel hacia los niveles inferiores de a una letra por nivel, o sea, la primera letra en el segundo nivel del árbol, la segunda letra en el tercer nivel del árbol, ..., la última (*n*-ésima) letra en el nivel *n* + 1.
3. Se coloca el símbolo "\$" finalizando las palabras.
4. Adicionalmente, se considera que las listas de hermanos se mantienen ordenadas en forma lexicográfica y que el símbolo "\$" es el menor de todos.

A modo de ejemplo, si consideramos la estructura de la Figura 9a, las palabras almacenadas serían: *a*, *al*, *cal*, *can*, *la*, *las*, *lo*, *los*.

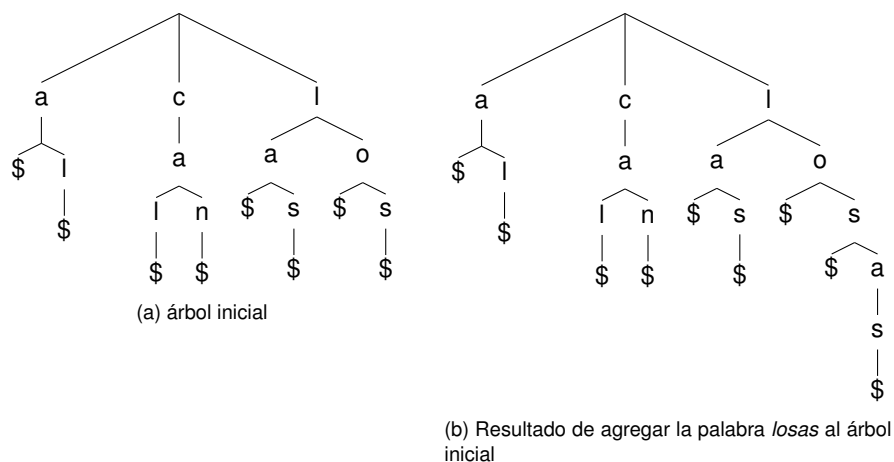


Figura 9: Ejemplos de árbol de caracteres.

Es conveniente destacar que cada camino de la raíz a una hoja (sin considerar el nodo dummy de la raíz) corresponde a una palabra del conjunto representado y que, como puede verse en el ejemplo, los nodos que tienen etiqueta "\$" necesariamente son hojas y solo pueden ser hojas, con lo cual no pueden tener hijos.

Para representar este tipo de árbol general se puede utilizar el tipo AG definido anteriormente pero donde el dato es de tipo *char* en lugar de *int*.

- (a) Escriba un procedimiento recursivo que dado un árbol general representado por el tipo AG agregue una palabra dada al diccionario. Dado el árbol del ejemplo anterior y la palabra *losas*, el árbol resultado debe ser el que se presenta en la Figura 9b. Considere que el árbol con el que se invocará la función posee al menos la raíz (nodo dummy).
- (b) Escriba un procedimiento llamado *palabras* que dado un árbol general representado por el tipo AG imprima todas las palabras del árbol. Por ejemplo, para el árbol de la Figura 9b, el procedimiento *Palabras* debe imprimir: *a, al, cal, can, la, las, lo, los, losas*.