Estructuras arborescentes

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.

(a) ¿Cómo representaría al árbol vacío con dicha representación?

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

I. función `consArbol`: que retorna un árbol no vacío a partir de un natural y otros dos árboles.

II. función `contarElems`: que recibe un árbol y retorna la cantidad de elementos del mismo.

III. función `contarHojas`: que recibe un árbol y retorna la cantidad de hojas (nodos cuyos ambos subárboles son vacíos) del mismo.

IV. función `altura`: que recibe un árbol y retorna la altura del mismo. Si el árbol es vacío su altura es 0.

V. función `copiar`: que recibe un árbol y retorna una *copia limpia* (que no comparte registros de memoria) del mismo.

VI. procedimiento `liberarArbol`: que recibe un árbol y elimina del mismo todos los nodos, liberando la memoria asociada a cada uno de ellos.

(c) 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 *b* y retorna una lista con los elementos de *b* ordenados según la recorrida en orden de *b*.

II. `preOrden`: que recibe un árbol *b* y retorna una lista con los elementos de *b* ordenados según la recorrida en pre orden de *b*.

III. `postOrden`: que recibe un árbol *b* y retorna una lista con los elementos de *b* ordenados según la recorrida en post orden de *b*.

IV. `esCamino`: que recibe un árbol *b* y una lista *l*, y retorna `TRUE` si y solo si *l* es igual a un camino desde la raíz a un hoja de *b*.

V. `caminoMasLargo`: que recibe un árbol *b* y retorna una lista con los elementos del camino más largo de *b* (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) Se dice que un árbol es perfecto si todas las hojas están en el mismo nivel y todos los nodos internos tiene dos subárboles no vacíos, o sea, si todos los niveles están completos. ¿Cuántos nodos tiene un árbol binario perfecto de altura *h*? Escriba una función booleana que dados un árbol binario *b* y un natural *h*, retorne `TRUE` si y solo si *b* es un árbol perfecto de altura *h*. Implemente dicha función sin usar operaciones auxiliares para calcular la cantidad de nodos o la altura de un árbol. Cada nodo se puede visitar a lo sumo una vez y no se deben visitar nodos innecesarios.

(c) ¿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.

(d) Se dice que un árbol es completo si todos los niveles están completos, excepto tal vez el más profundo al que le pueden faltar nodos a la derecha. Escriba una función booleana que dados un árbol binario *b* y un natural *h*, retorne `TRUE` si y solo si *b* es un árbol completo de altura *h*. Cada nodo se puede visitar a lo sumo una vez y no se deben visitar nodos innecesarios.

Segunda parte: árboles binarios de búsqueda

Ejercicio 3

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.

Utilice la representación dada para implementar las siguientes operaciones:

(a) procedimiento `insertarABB`: que recibe un natural *x*, y un ABB *b*, e inserta *x* en *b* manteniendo su cualidad de árbol binario de búsqueda. Si `x` pertenece al árbol la operación no tiene efecto.

(b) función `perteneceABB`: que recibe un natural *x* y un ABB *b* y devuelve true si y solo si *x* es un elemento del árbol *b*.

(c) función `maxABB`: que recibe un ABB no vacío *b* y devuelve el elemento de máximo valor en *b*.

(d) procedimiento `removerMaxABB`: que recibe un ABB no vacío *b* y elimina el elemento de máximo valor en *b*.

(e) procedimiento `removerABB`: que recibe un natural *x* y un ABB *b* y elimina el elemento de valor *x* de *b*, manteniendo su cualidad de ABB.

(f) función `k-esimo`: que recibe un natural *k* y un ABB *b* y retorna el subárbol que tiene al k-ésimo menor elemento de *b* como raíz. Si en *b* hay menos de *k* elementos 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 ningún nodo más 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 `removerMaxABB` 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 'b' en los que "nota" es ↩
   mayor que "cota". */
ABB filtrado (ABB b; uint cota);
```

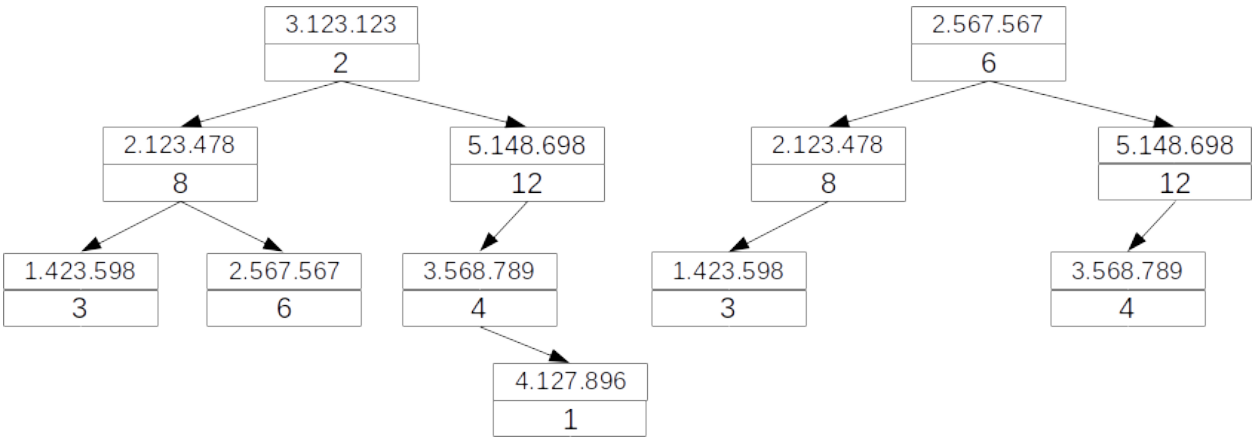


Figura 3: A modo de ejemplo, se presenta un árbol 'b' (sobre la izquierda) y el resultado de filtrarlo con una cota igual a la nota 2 (sobre la derecha).

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 de enteros (representados con la semántica primer hijo, siguiente hermano) no vacíos y sin elementos repetidos:

(a) `arbolHoja`: Dado un entero *x* retorna un árbol que sólo contiene a *x* (como una hoja).

(b) `esArbolHoja`: Dado un árbol, retorna true si y solo si el árbol es un árbol hoja (tiene un solo elemento).

(c) `pertenece`: Dados un árbol y un entero *x*, retorna true si y solo si *x* pertenece al árbol.

(d) `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) si *p* pertenece al árbol y *h* no pertenece al árbol. En caso contrario la operación no tiene efecto.

(e) `borrar`: Dados un árbol y un entero *x*, elimina a *x* del árbol si es una hoja del árbol y no es la raíz del mismo. En caso contrario la operación no tiene efecto. Al eliminar el elemento se debe liberar la memoria asignada a él.

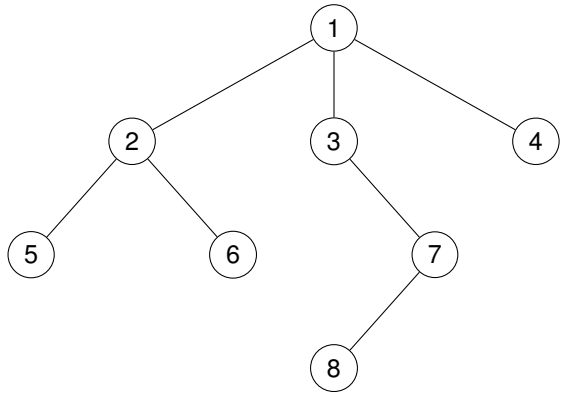
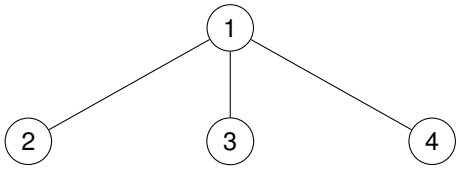
(f) `borrarSub`: Dados un árbol y un entero *x*, elimina a *x* del árbol, si pertenece al árbol y no es la raíz del mismo. En caso contrario la operación no tiene efecto. Al eliminar el elemento se deberá liberar la memoria asignada a los elementos que están en el subárbol dependiente de éste.

Ejercicio 6 Primer Parcial 2017

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

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

Ejemplo:

Entrada	Resultado
$k = 2$ a 	

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. Se dice que la secuencia x es prefijo de la secuencia z si z es xy, esto es, la concatenación de x e y. Cualquiera de las secuencias x, y, z pueden ser vacías. En particular si x es vacía es prefijo de cualquier secuencia, y toda secuencia es prfijo de de sí misma. No se deben usar funciones o procedimientos auxiliares en este ejercicio. Utilice la definición de lista presentada en el ejercicio 2.

Ejercicio 8

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 es vacío o la raíz no tiene hijos, la función debe retornar 0.

```
int mayorAmplitud(AG g)
```

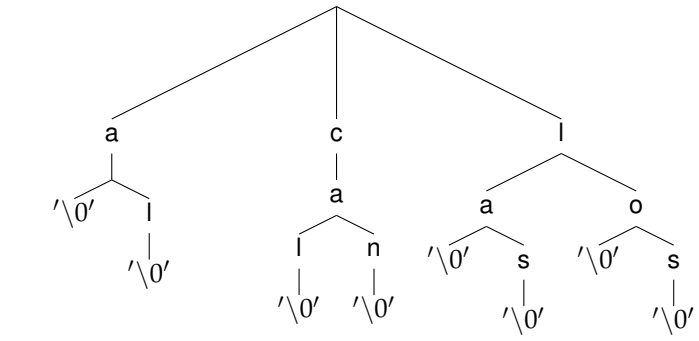
Ejercicios Complementarios

Ejercicio 9

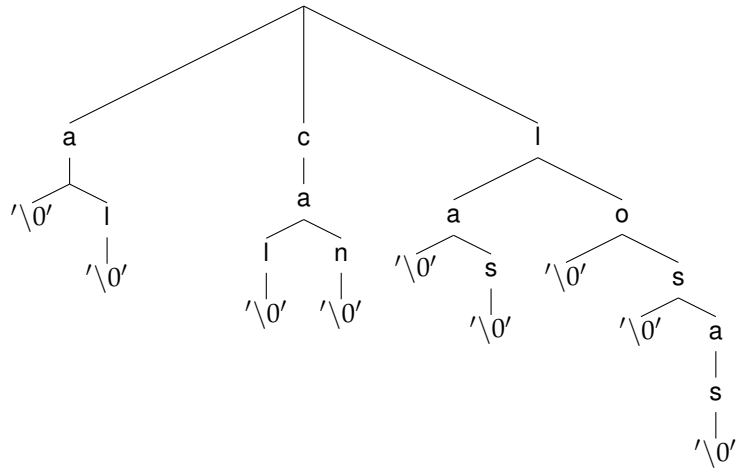
Una forma común de almacenamiento de conjuntos de palabras consiste en la utilización de árboles generales de caracteres, ya que al evitar la duplicación de prefijos comunes permiten disminuir la cantidad de información almacenada. Esto se realiza de la siguiente forma:

1. Se deja la raíz del árbol sin información (nodo dummy).
2. Se colocan las palabras desde el segundo nivel hacia los niveles más profundos 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 $'\backslash 0'$ finalizando las palabras.
4. Adicionalmente, se considera que las listas de hermanos se mantienen ordenadas en forma lexicográfica y que el símbolo $'\backslash 0'$ es el menor de todos.

A modo de ejemplo, si consideramos la estructura de la Figura 4a, las palabras almacenadas serían: *a, al, cal, can, la, las, lo, los.*



(a) árbol inicial



(b) Resultado de agregar la palabra *losas* al árbol inicial

Figura 4: 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, ni el $'\backslash 0'$ de la hoja) corresponde a una palabra del conjunto representado y que, como puede verse en el ejemplo, hay una correspondencia entre los nodos con etiqueta $'\backslash 0'$ y las hojas.

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 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 4b, el procedimiento `Palabras` debe imprimir: *a, al, cal, can, la, las, lo, los, losas.*

(b) Escriba una función que dada una palabra devuelva un `AG` que representa esa única palabra. Es decir, consiste en una celda dummy y un nodo por nivel para cada uno de los símbolos de la palabra, terminando en la hoja $'\backslash 0'$.

(c) Escriba un procedimiento recursivo que dado un árbol general representado por el tipo `AG` y una palabra, agregue la palabra al árbol. Dado el árbol del ejemplo anterior y la palabra *losas*, el árbol resultado debe ser el que se presenta en la Figura 4b. Considere que el árbol con el que se invocará la función posee al menos la raíz (nodo dummy).

Ejercicio 10 ExamenJulio2021

Considere un árbol general de enteros representado mediante un árbol binario de enteros con la semántica: puntero al primer hijo (pH), puntero al siguiente hermano (sH).

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

Implemente la función `AG padre(AG g, int x)` que retorne un puntero al nodo padre en `g` del nodo que tenga a `x` como dato. Asumimos que `g` no tiene elementos repetidos. Si `x` no está en `g` o si `x` es la raíz de `g`, la función `padre` deberá retornar `NULL`. No se pueden definir operaciones auxiliares para implementar padre.