Programación 2

La previa
Estructuras Arborescentes
(parte 2)
+
Repaso para el primer parcial

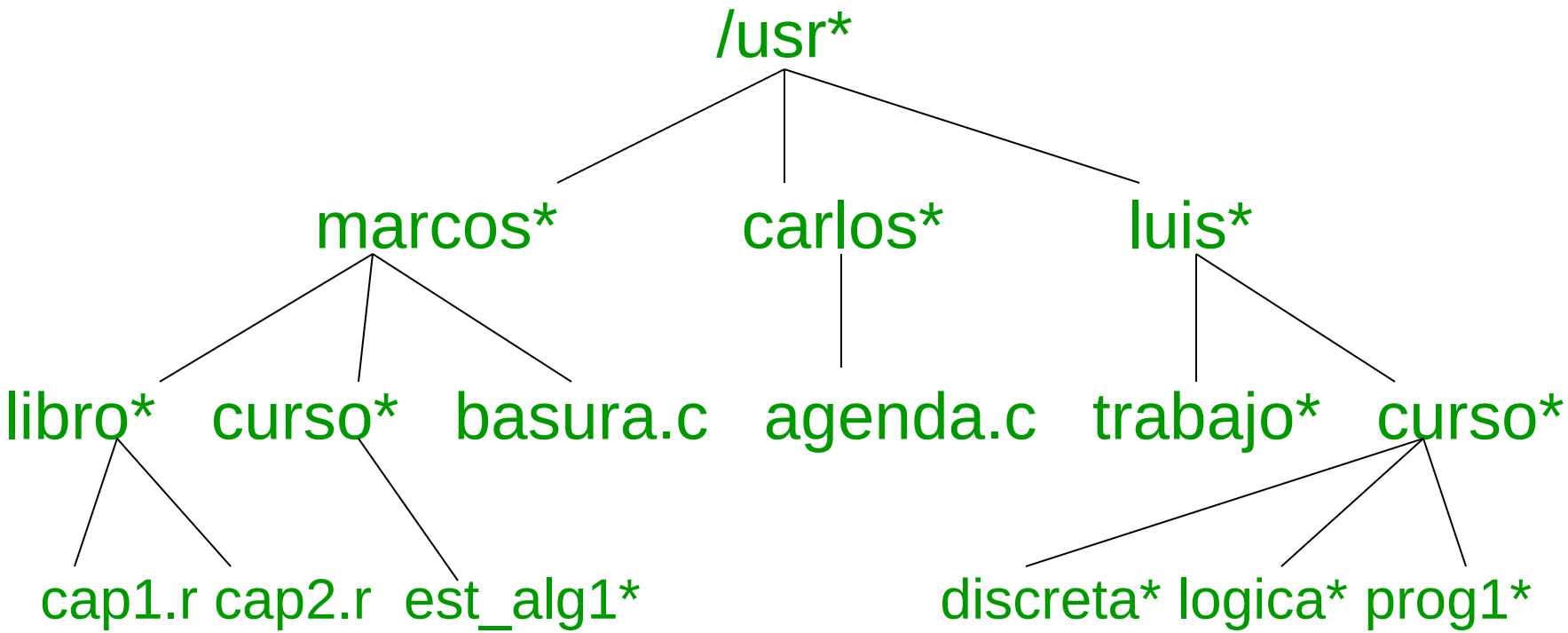
Arboles Generales.
Cómo implementarlos?

Una estructura *árbol* (*árbol general o finitario*) con tipo base T es,

1. O bien la estructura vacía
2. O bien un nodo de tipo T, llamado raíz del árbol, junto con un número finito de estructuras de árbol, de tipo base T, disjuntas, llamadas *subárboles*

¿cómo representamos árboles generales?

Un ejemplo de árbol general: “estructura de directorios”

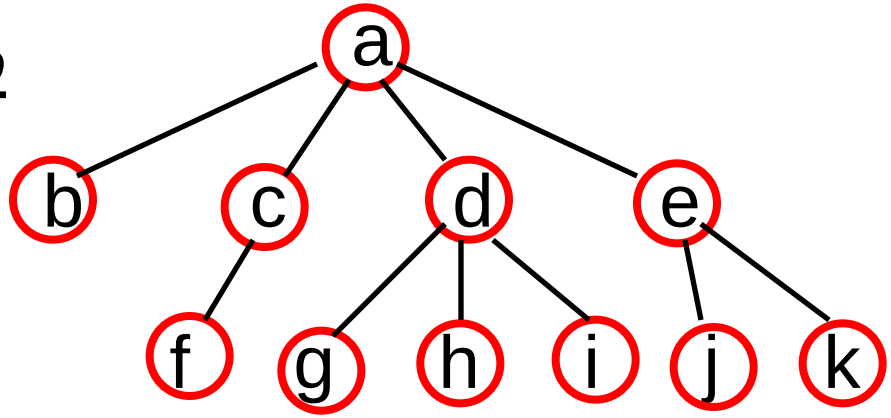


Una aplicación: listar un directorio

(recorridos de árboles)

Arbol general - Arbol binario

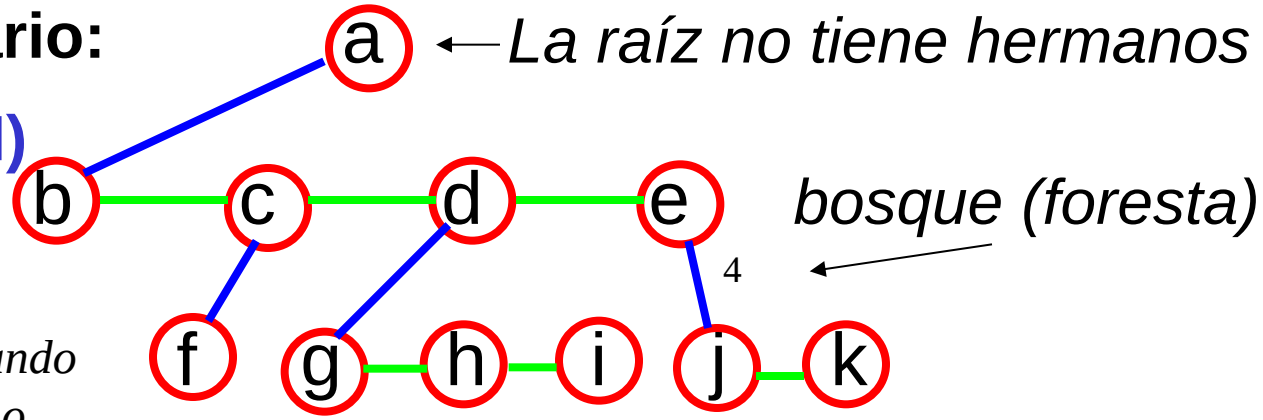
Cada nodo tiene un núnero *finitio* de subárboles



. . . . . .

Representación como árbol binario:

*** primer hijo (pH)**
*** siguiente hermano (sH)**

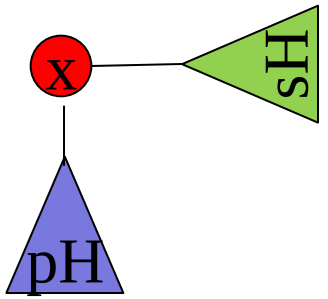


. . . . . .

Notar que estamos representando tanto bosques de árboles como árboles (bosques de un sólo árbol; la raíz).

Ejemplo 1

- La función contar nodos de un árbol general (recordar que la raíz no tiene hermanos).



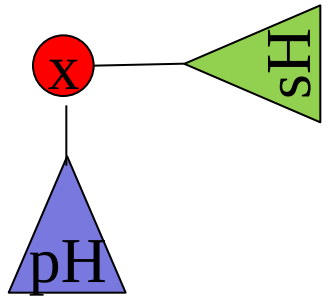

```
typedef NodoAG* AG;
struct NodoAG { T item; AG pH; AG sH; };

int nodos(AG t){
   if (t == NULL) return 0;
   else return nodos(t->pH)+nodos(t->sH)+1;
}
```

Notar que:

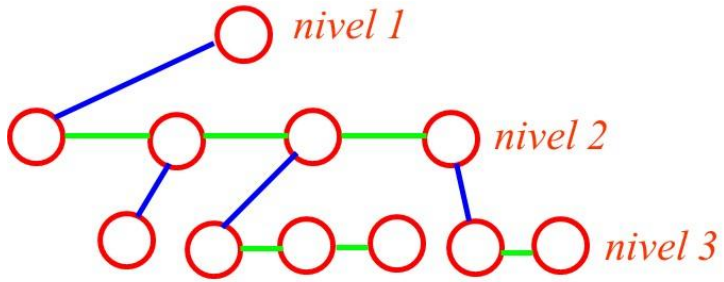
- **el código es idéntico al de contar nodos en un árbol binario tradicional.**
- **si se invoca a *nodos* con un bosque *t* (*t->sH!=NULL*), se tiene la cantidad de nodos del bosque.**

Ejemplo 2



- La función altura de un árbol general (recordar que la raíz no tiene hermanos).

```
int altura(AG t){
   if (t == NULL) return 0;
   else return MAX(1+altura(t->pH),
                   altura(t->sH));
}
```



Notar que el código NO es idéntico al de la altura de un árbol binario tradicional:

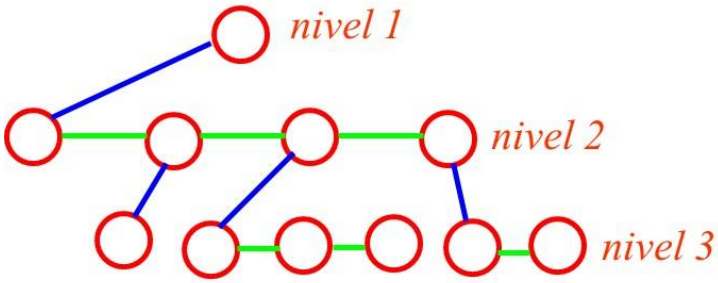
- "primer hijo" (**pH**) aumenta la altura.
- "siguiente hermano" (**sH**) no aumenta la altura, la mantiene (ver dibujo).

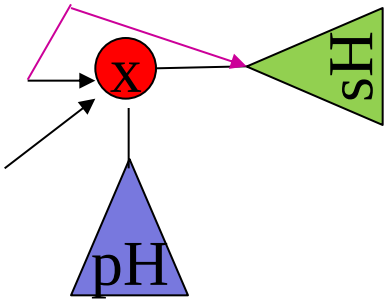
Si se invoca a la función *altura* con un bosque *t* (*t->sH!=NULL*), se tiene la altura del bosque.

Ejemplo 3

- Elimina cada subárbol que tiene a x como raíz de un árbol general (recordar que en la representación *pH-sH*, el nodo raíz principal no tiene hermanos).

...

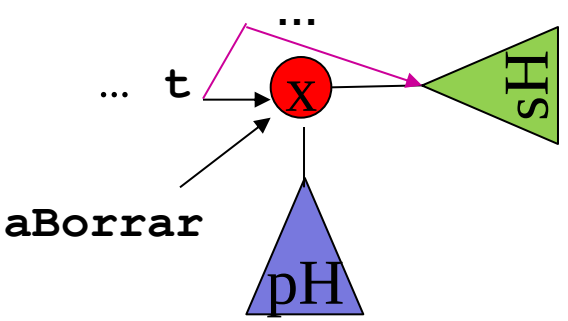




Ejemplo 3

- Elimina cada subárbol que tiene a x como raíz de un árbol general (recordar que en la representación *pH-sH*, el nodo raíz principal no tiene hermanos).

```
void elim(AG & t, T x){
   if (t != NULL){
        if (t->item != x){
               elim(t->pH, x);
               elim(t->sH, x);
        }else{ AG aBorrar = t;
               t = t->sH;
               elimTodo(aBorrar->pH);
               delete aBorrar;
               elim(t, x);
        }
   }
}
```




```
void elimTodo(AG & t){
  if (t != NULL){
    elimTodo(t->pH);
    elimTodo(t->sH);
    delete t;
    t = NULL;
  }
}
```

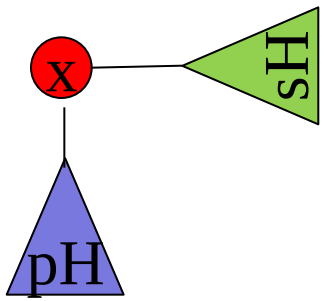
Buscar dato y dar su ubicación

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

Pensar en la especificación del problema...

AG buscar (AG t, int x)



Buscar dato y dar su ubicación

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

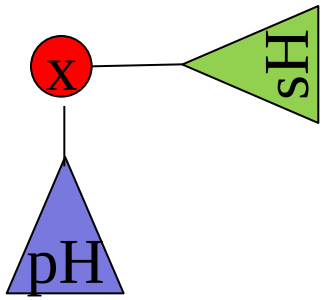
```
AG buscar (AG t, int x)
    if (t==NULL) return NULL;
    else if (t->dato==x) return t;
    else {      AG ret = buscar(t->pH, x);
                if (ret!=NULL) return ret;
                else return buscar(t->sH, x);
    }
}
```

Nivel dato

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

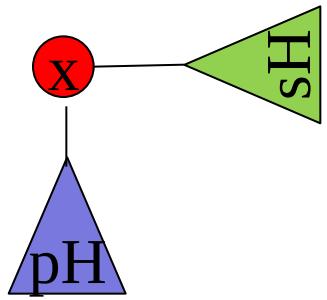
PRE: t no tiene elementos repetidos

int nivel(AG t, int x)



Nivel dato

```
int nivel(AG t, int x){
    if (t==NULL) return 0;
    else if (t->dato==x) return 1;
    else {      int ret = nivel(t->pH, x);
                if (ret>0) return 1+ret;
                else retun nivel (t->sH, x);
    }
}
```



Repaso de ABB

a) Considere la siguiente definición para árboles binarios de búsqueda de enteros (*ABB*):

```
typedef nodoABB * ABB
struct nodoABB { int dato; ABB izq; ABB der; }
```

Implemente una función recursiva (sin usar iteración) ***delMin*** que dado un árbol binario de búsqueda *t* de tipo *ABB* no vacío, elimine de *t* el mínimo elemento y retorne su valor. El árbol resultante deber ser también binario de búsqueda. No implemente ni asuma la existencia de operaciones auxiliares para implementar *delMin*.

```
PRE: t no vacío
int delMin(ABB & t)
```

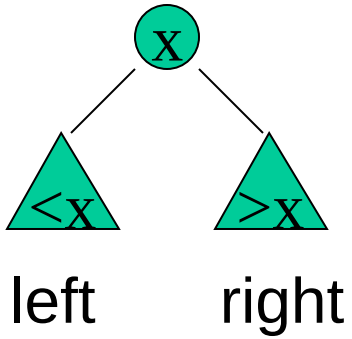
b) Explique muy brevemente el orden de tiempo de ejecución para el peor caso y para el caso promedio de ***delMin***.

Repaso de ABB

```
typedef nodoABB * ABB;
struct nodoABB { int dato; ABB izq; ABB der; };
```

PRE: t no vacío

int ***delMin****(ABB & t)*

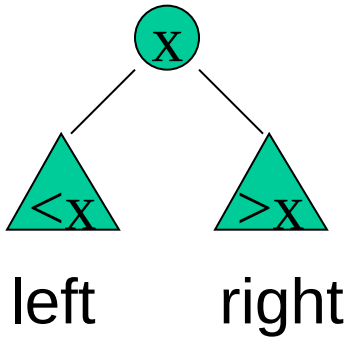


Repaso de ABB

```
typedef nodoABB * ABB;
struct nodoABB { int dato; ABB izq; ABB der; };
```

PRE: t no vacío

```
int delMin(ABB & t){
    if (t->izq==NULL){
        int min = t->dato;
        ABB aBorrar = t;
        t = t->der;
        delete aBorrar;
        return min;
    }
    else return delMin(t->izq);
}
```



Repaso de ABB

Para trabajar con una agenda de contactos telefónicos, considere la siguiente definición del tipo ***ABB*** de árboles binarios de búsqueda, que contienen números telefónicos (de tipo ***int***) y nombres completos de contactos (de un tipo ***T***):

```
struct nodoABB{
   unsigned int telefono; // número telefónico de un contacto
   T nombre; // nombre completo del contacto
   nodoABB * izq, * der;
}
typedef nodoABB * ABB;
```

Una agenda de contactos está modelada con un árbol de tipo ***ABB***, organizado (ordenado) por números telefónicos, que NO se pueden repetir. Los nombres completos asociados a los teléfonos en el árbol si podrían repetirse. Asuma que los nombres de tipo ***T*** pueden asignarse y compararse con los operadores habituales (=, ==).

a) Implemente un procedimiento recursivo ***void agendar(ABB & t, int num, T nom)*** que, dado un árbol *t* de tipo ***ABB***, un número de teléfono *num* y un nombre completo *nom*, inserta el contacto *num* con nombre *nom* en *t*, si *num* no estaba en *t*. En caso contrario, reemplaza el nombre completo asociado a *num* en *t* con *nom*. No defina operaciones auxiliares.

Indique el orden de tiempo de ejecución en el peor caso y en el caso promedio de ***agendar***.

Repaso de ABB

```
void agendar(ABB & t, int num, T nom){
    if (t == NULL){ // ingreso de un nuevo contacto
            t = new nodoABB;
            t->telefono = num;
            t->nombre = nom;
            t->izq = t->der = NULL;
    }
    else{
    if (num < t->telefono)
            agendar(t->izq, num, nom);
    else (num > t->telefono)
            agendar(t->der, num, nom);
    else t->nombre = nom; // reemplaza (actualiza) el nombre asociado al contacto num
    }
}
```

Indique el orden de tiempo de ejecución en el peor caso y en el caso promedio de **`agendar`**.

Repaso de ABB

Para trabajar con una agenda de contactos telefónicos, considere la siguiente definición del tipo ***ABB*** de árboles binarios de búsqueda, que contienen números telefónicos (de tipo ***int***) y nombres completos de contactos (de un tipo ***T***):

```
struct nodoABB{
   unsigned int telefono; // número telefónico de un contacto
   T nombre; // nombre completo del contacto
   nodoABB * izq, * der;
}
typedef nodoABB * ABB;
```

Una agenda de contactos está modelada con un árbol de tipo ***ABB***, organizado (ordenado) por números telefónicos, que NO se pueden repetir. Los nombres completos asociados a los teléfonos en el árbol si podrían repetirse. Asuma que los nombres de tipo ***T*** pueden asignarse y compararse con los operadores habituales (=, ==).

b) Implemente una función recursiva ***int contactos(ABB t, T nom)*** que, dado un árbol *t* de tipo ***ABB*** y un nombre completo *nom*, retorna la cantidad de contactos (números telefónicos) en *t* que coinciden con *nom*. Si *nom* no está en *t*, el resultado debe ser 0. No defina operaciones auxiliares.

Indique el orden de tiempo de ejecución en el peor caso de ***contactos***.

Repaso de ABB

```
int contactos(ABB t, T nom)
    if (t == NULL)
        return 0;
    else if (t->nombre == nom)
        return 1 + contactos(t->izq, nom) + contactos(t->der, nom);
    else return contactos(t->izq, nom) + contactos(t->der, nom);
}
```

Indique el orden de tiempo de ejecución en el peor caso de **`contactos`**.

Repaso de ABB

Considere la siguiente definición del tipo ***ABB*** de árboles binarios de búsqueda de números reales (sin elementos repetidos):

```
struct nodoABB{ float dato; nodoABB * izq, * der; };
typedef nodoABB * ABB;
```

Implemente una operación recursiva ***borrarMayores*** que, dados un número real *x* y un árbol binario de búsqueda *t* de tipo *ABB*, elimine de *t* todos los elementos mayores estrictos que *x* y retorne la cantidad de elementos suprimidos. El árbol resultante debe ser también binario de búsqueda. Si no hay elementos mayores que *x* en *t* (en particular si *t* es el árbol vacío, NULL), *borrarMayores* no tendrá efecto en *t* y el retorno será 0. La operación deberá evitar recorrer nodos innecesarios de *t*. No defina operaciones auxiliares para implementar *borrarMayores*.

```
int borrarMayores(float x, ABB & t)
```

Repaso de ABB

```
int borrarMayores(float x, ABB & t){
    int result = 0;
    if (t != NULL){
        if (t->dato > x){
                result = 1 + borrarMayores(x, t->izq) +
                             borrarMayores(x, t->der);
                ABB aBorrar = t;
                t = t->izq;
                delete aBorrar;
        }
        else    result = borrarMayores(x, t->der);
    }
    return result;
}
```

Repaso de árboles

Se quieren definir procedimientos para eliminar hojas de árboles binarios (AB) y árboles generales (AG). Tener presente que un nodo es hoja si no tiene hijos, tanto en un AB como en un AG. Los procedimientos deberán tener O(n) en el peor caso, siendo n la cantidad de nodos del árbol, en cada caso.

a) Considere la definición del tipo AB de árboles binarios de enteros:

```
typedef nodoAB * AB;
struct nodoAB { int dato; AB izq, der; };
```

Implemente un procedimiento recursivo **elimHojasAB** que dado un AB t, elimine de t todos sus nodos hojas, liberando la memoria que corresponda. Si el árbol es vacío (t es NULL), el procedimiento no tendrá efecto. No se pueden implementar funciones o procedimientos auxiliares.

```
void elimHojasAB (AB & t)
```

Repaso de árboles

```
void elimHojasAB (AB & t){
  if (t != NULL){
     if (t->izq == NULL && t->der == NULL) {
          delete t;
          t = NULL;
     } else{
          elimHojasAB (t->izq);
          elimHojasAB (t->der);
     }
  }
}
```

Repaso de árboles

b) Considere ahora la siguiente definición del tipo ***AG*** de árboles generales de enteros, representados con árboles binarios mediante la semántica: primer hijo (pH) – siguiente hermano (sH).

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

Implemente un procedimiento recursivo ***elimHojasAG*** que dado un AG *t*, elimine de *t* todos sus nodos hojas, liberando la memoria que corresponda. Si el árbol es vacío (*t* es NULL), el procedimiento no tendrá efecto. No se pueden implementar funciones o procedimientos auxiliares, ni usar iteración (while/for).

```
void elimHojasAG (AG & t)
```

Repaso de árboles

```
void elimHojasAG (AG & t){
  if (t != NULL){
      if (t->pH == NULL) {
            AG aBorrar = t;
            t = t->sH;
            delete aBorrar;
            elimHojasAG (t);
      } else{
            elimHojasAG (t->pH);
            elimHojasAG (t->sH);
      }
  }
}
```

Repaso de Listas

a) Se desea trabajar con listas ordenadas de identificadores de productos de un negocio. Considere la siguiente definición de listas de enteros (identificadores de productos):

```
struct nodoLista { int dato; nodoLista * sig; };
typedef nodoLista * Lista;
```

Implemente una función iterativa ***Lista comunes(Lista l1, Lista l2)*** que dadas dos listas de enteros ordenadas de menor a mayor y sin elementos repetidos, construya y retorne una nueva lista ordenada de menor a mayor que contenga solamente a los elementos (identificadores de productos) comunes; es decir, que están presentes en ambas listas. Si no hay elementos comunes, el resultado debe ser la lista vacía, NULL.

La lista resultado no deberá tener elementos repetidos ni compartir memoria con las listas parámetro, que no pueden modificarse. La función debe tener O(n+m) peor caso, siendo n y m los largos de las listas parámetro. No implemente funciones o procedimientos auxiliares, ni use arreglos/vectores.

Por ejemplo, si las listas son [1,**3**,**4**,8,**9**] y [2,**3**,**4**,6,**9**,10,17], el resultado debería ser: [**3**,**4**,**9**].

b) Justifique informal y brevemente el orden de tiempo de ejecución requerido para el peor caso de ***comunes***.

Repaso de Listas

```
Lista comunes(Lista l1, Lista l2){
    Lista res = NULL;
    Lista fin_res, nodo;
    while (l1!=NULL && l2!=NULL){
        if (l1->dato == l2->dato){
                nodo = new nodoLista;
                nodo->dato = l1->dato;
                nodo->sig = NULL;
                if (res == NULL) res = nodo;
                else fin_res->sig = nodo;
                fin_res = nodo;
                l1 = l1->sig;
                l2 = l2->sig;
        } else if (l1->dato < l2->dato) l1 = l1->sig;
        else l2 = l2->sig;
    }
    return res;
}
```

Repaso de Listas

La función **comunes** es O(n+m) en el peor caso, siendo n y m el lago de cada lista parámetro, ya que recorre ambas listas una única vez, haciendo operaciones de O(1).

Repaso de Listas

Considere las siguientes definiciones de listas de enteros (*Lista*) y de pares de enteros (*ListaPar)*:

```
typedef nodoLista * Lista
struct nodoLista{ int dato; Lista sig; }
typedef nodoListaPar * ListaPar
struct nodoListaPar{ int dato; int cantidad; ListaPar sig; }
```

a) Implemente una **<u>función iterativa</u>** ***ordenar*** que dada una lista *l* de tipo *Lista* que puede contener valores exclusivamente en el rango [0 : *m*] (entre 0 y *m* inclusive, con $m > 0$), retorne una nueva lista de tipo *ListaPar* ordenada de mayor a menor según los datos de *l*, que contenga para cada elemento de *l* la cantidad de sus repeticiones. Si *l* es vacía (NULL), el resultado debe ser NULL. Se pueden usar estructuras de datos auxiliares, manejando adecuadamente la memoria (pedido y liberación, si corresponde). **La función *ordenar* debe ser O(max(*n*,*m*)) en el peor caso**, siendo *n* el largo de *l*.

```
PRE: Cada elemento x de l cumple: 0 ≤ x ≤ m, con m > 0
```

```
ListaPar ordenar (Lista l, int m)
```

Por ejemplo, si *l* es [2, 4, 83, 4, 4, 99, 2, 83, 5] y *m* es 100, el resultado debe ser [(99,1), (83,2), (5,1), (4,3), (2,2)].

b) Justifique brevemente el orden exigido para ***ordenar***.

Repaso de Listas

```
Lista ordenar(Lista l, int m){
   ListaPar ret = NULL;
   int * elementos = new int[m+1];
   for (int i=0; i<=m; i++)
        elementos[i] = 0;
   while (l!=NULL){
        elementos[l->dato]++;
        l = l->sig;
   }
   for (int i=0; i<=m; i++){
        if(elementos[i]>0)
              insComienzo(i, elementos[i], ret);
   }
   delete [] elementos;
   return ret;
}
```