Generalidades:

a) La prueba es individual y sin material.
b) La duración es 3hs.
c) Sólo se contestan dudas acerca de la letra.
d) Escriba las hojas de un solo lado y con letra clara.
e) Numere cada hoja, indicando en la primera el total.
f) Coloque número de cédula y nombre en cada hoja.

Ejercicio 1 (10 puntos)

Considere las siguientes definiciones de listas encadenadas de enteros y de pares de enteros:

Lista de enteros

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

typedef nodoLista *Lista;
```

Lista de pares enteros

```
struct nodoListaPares {
  int primero, segundo;
  nodoListaPares *sig;
};

typedef nodoListaPares *ListaPares;
```

Implemente una función iterativa (no recursiva) **combinar** que, dadas dos listas de enteros $L1 = [a1, a2, \ldots, an]$ y $L2 = [b1, b2, \ldots, bm]$, retorne la lista de pares $[(a1, b1), (a2, b2), \ldots, (ak, bk)]$, donde k es el mínimo entre n y m. Si una de las listas es vacía (`NULL`), el resultado debe ser la lista vacía (`NULL`). $L1$ y $L2$ no deben recorrerse más de una vez. El orden de tiempo de ejecución en el peor caso de combinar debe ser $O(\text{mín}(n, m))$.

ListaPares **combinar** (Lista L1, Lista L2);

Solución:

Se debe construir una lista de k nodos enlazados en tiempo $O(k)$. Esto implica que cada nuevo nodo se debe insertar en $O(1)$. Como no puede haber recursión, las listas $L1$ y $L2$ deben recorrerse desde el inicio hacia el final. Como para construir el $i-$ésimo nodo de la lista resultado se debe estar accediendo al $i-$ésimo nodo de $L1$ y al $i-$ésimo nodo de $L2$, la lista resultado también debe construirse desde el inicio hacia el final. Por lo tanto es necesario hacer la inserción en $O(1)$ al final de la lista que se está construyendo. Esto impide que para la inserción del nuevo nodo se recorra la lista resultado desde el inicio para llegar al final, porque demandaría tiempo $O(k)$. Entonces la solución es mantener una variable iteradora, de tipo *ListaPares*, que siempre mantenga la referencia del último elemento de la lista resultado. En cada paso, el nuevo nodo debe ser enlazado por el que era el último, y el nuevo debe pasar a ser el último. Por supuesto, también debe mantenerse la referencia al primer nodo, que es el que hay que devolver. No hacerlo y devolver la varibale iteradora implica perder le referencia a todos los nodos excepto el último.

En una solución alternativa, sin mantener la referencia al último elemento, se podría construir una lista auxiliar insertando siempre al inicio, quedando la lista en orden inverso al pedido. En una segunda pasada por la lista auxiliar se invierte, siempre insertando al inicio, obteniendo la lista pedida. Si bien este algoritmo es $O(k)$, recorre dos veces los nodos resultado, provocando ineficiencia.

Como la lista resultado no debe ser implementada con un arreglo, sino con nodos enlazados, no es necesario conocer antes de empezar la construcción la cantidad de elementos que va a contener. Además, conocer esa cantidad antes de empezar la construcción implicaría recorres dos veces $L1$ y $L2$, lo cual va en contra de lo requerido.

Debe notarse que como $L1$ y $L2$ son pasados por valor, dentro del cuerpo de la función se comportan como variables locales, por lo que pueden ser modificados sin que eso afecte a las variables que llaman a la función. Por lo tanto no es necesario definir otras variables para iterar en esas listas.

Se muestran dos soluciones. La que está en el listado 1 le da un trato diferenciado a la inserción del primer elemento. La que está en el listado 2 evita eso y el caso en que la lista resultado es vacía, creando una celda base (a veces llamada *dummy*), pero que debe ser eliminada al final.

En ambas soluciones se observa que en el bucle principal, al crear un nuevo nodo se asignan los valores de los campos *primero* y *segundo*, pero no se asigna `NULL` al campo *sig*. Esto último sería innecesario ya que en la siguiente pasada de la iteración ese valor va a ser asignado al siguiente nuevo nodo. Solamente en el último nodo, al terminar la iteración, se necesita asignar `NULL` al campo *sig*.

Listado 1: combinar - Versión con trato diferenciado para el primer elemento.

```
ListaPares combinar(Lista L1, Lista L2) {
  ListaPares res;
  if ((L1 == NULL) || (L2 == NULL)) {
    res = NULL;
  } else {
    res = new nodoListaPares;
    res->primero = L1->dato;
    res->segundo = L2->dato;
    ListaPares iter = res;
    while ((L1->sig != NULL) && (L2->sig != NULL)) {
      L1 = L1->sig;
      L2 = L2->sig;
      iter->sig = new nodoListaPares;
      iter = iter->sig;
      iter->primero = L1->dato;
      iter->segundo = L2->dato;
    }
    iter->sig = NULL;
  }
  return res;
}
```

Listado 2: combinar - Versión con celda base.

```
ListaPares combinar(Lista L1, Lista L2) {
  ListaPares iter, res = new nodoListaPares;
  iter = res;
  while ((L1 != NULL) && (L2 != NULL)) {
    iter->sig = new nodoListaPares;
    iter = iter->sig;
    iter->primero = L1->dato;
    iter->segundo = L2->dato;
    L1 = L1->sig;
    L2 = L2->sig;
  }
  iter->sig = NULL;
  iter = res;
  res = res->sig;
  delete iter;
  return res;
}
```

Ejercicio 2 (10 puntos)

Considere la siguiente definición del tipo **ABB** de los árboles binarios de búsqueda de enteros, en memoria dinámica:

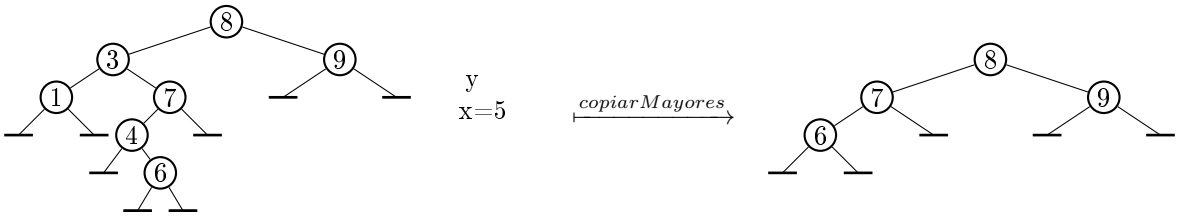
ABB

```
struct nodoABB {
  int dato;
  nodoABB *izq, *der;
};

typedef nodoABB *ABB;
```

Defina una función recursiva **copiarMayores** que dados un árbol binario de búsqueda de enteros A de tipo **ABB** (sin elementos repetidos) y un entero x, retorne una copia de A que contenga a todos los elementos del árbol cuyos datos son mayores estrictos a x. Si A es vacío (`NULL`) o no hay elementos estrictamente mayores a x en A, el resultado debe ser el árbol vacío (`NULL`). La función debe retornar un árbol binario de búsqueda que no comparta memoria con el árbol parámetro. El procedimiento debe tener $O(n)$ de tiempo de ejecución en el peor caso (siendo n la cantidad de nodos de A) y debe evitar recorrer todo el árbol, de ser posible. No defina funciones o procedimientos auxiliares.

Ejemplo:



Solución:

Al implementar un algoritno recursivo se deben determinar casos base. En esta solución, que está en el listado 3, se toma el árbol vacío como caso base. En ese caso, de manera trivial no hay elementos mayores que x, por lo que se devuelve el árbol vacío. Las llamadas recursivas se hacen en las líneas 6 y 10, en subárboles más pequeños, que se acercan al caso base.

Listado 3: copiarMayores

```
1  ABB copiarMayores(ABB A, int x) {
2    ABB mayores;
3    if (A == NULL) {
4      mayores = NULL;
5    } else {
6      ABB mayDer = copiarMayores(A->der, x);
7      if (A->dato <= x) {
8        mayores = mayDer;
9      } else {
10       ABB mayIzq = copiarMayores(A->izq, x);
11       mayores = new nodoABB;
12       mayores->dato = A->dato;
13       mayores->izq = mayIzq;
14       mayores->der = mayDer;
15     }
16   }
17   return mayores;
18 }
```

De manera independiente de la relación entre x y `A->dato` (por la propiedad de orden de los árboles binarios de búsqueda) en `A->der` puede haber elementos mayores que x, por lo que la llamada de la línea 6 debe hacerse siempre.

En el ejemplo mostrado se ve que la recorrida no debe llegar al nodo cuyo dato es 1, ya que se sabe que en el subárbol izquierdo del nodo con dato 3 no puede haber elementos mayores a 5. Cuando esto pasa, la solución es el resultado de la llamada en el subárbol derecho.

Ejercicio 3 (10 puntos)

Se quiere representar una estructura simplificada de directorios (carpetas) de un sistema operativo, donde cada directorio tiene un nombre único que lo identifica. Cada directorio puede contener un número finito de subdirectorios. No se contemplan archivos en esta versión simplificada del sistema.

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

Directorios

```
struct nodoDir {
  char *nombre;
  nodoDir *pH, *sH;
};

typedef nodoDir *Directorios;
```

Defina un procedimiento **borrar** que, dados un **Directorios** 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 la memoria asociada a este. Utilice la función *strEq* (que se asume implementada) para comparar strings; *strEq* retorna `true` si y sólo si dos strings son iguales. No defina funciones o procedimientos auxiliares.

void **borrar** (Directorios & D, char * nom_dir);

Solución:

Como la representación de *Directorios* es un árbol binario, una implementación recursiva de *borrar* debería producir un código sencillo. Esto se muestra en el listado 4.

Se asume que la llamada inicial es en la raíz de todo el árbol, y que este está bien formado (esto es, `A->sH==NULL`). Debido a la implementación con la semántica primer hijo - siguiente hermano, en las sucesivas llamadas desde el parámetro A se puede acceder a sus siguientes hermanos.

Listado 4: borrar

```
1  void borrar(Directorios &A, char *nom_dir) {
2    if (A != NULL) {
3      if (strEq(A->nombre, nom_dir)) {
4        if (A->pH == NULL) {
5          Directorios borrar = A;
6          A = A->sH;
7          delete[] borrar->nombre;
8          delete borrar;
9        }
10     } else {
11       borrar(A->pH, nom_dir);
12       borrar(A->sH, nom_dir);
13     }
14   }
15 }
```

Como es habitual se puede tratar el árbol vacío como caso base. En esta implementación esto se maneja de manera implícita ya que si el árbol es vacío no se hace nada.

La recursión también debe detenerse cuando se llega a un nodo cuyo nombre sea *nom_dir*, ya que se sabe que los nombres de los directorios son únicos. Hacer llamadas recursivas en ese caso es innecesariamente ineficiente. Las llamadas recursivas sólo se hacen en el otro caso, lo cual se muestra

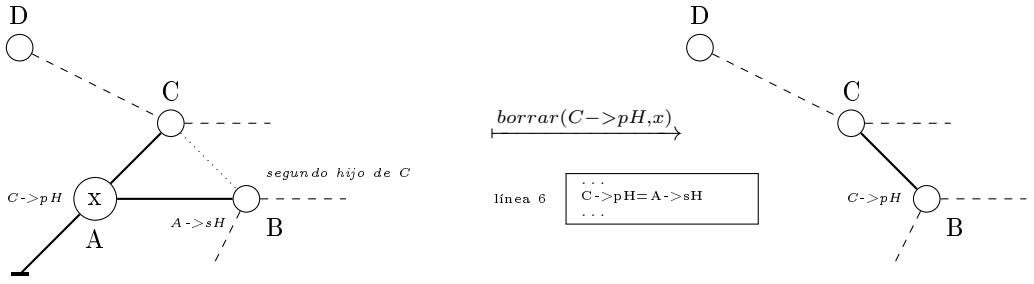
en las líneas 11 y 12. Las llamadas deben hacerse siempre para los dos subárboles de A, aunque en la primera de las llamadas se encuetre el nodo buscado, ya que *borrar* no devuelve información acerca de si se encontró el nodo o no.

Si se encuetra el nodo y no tiene hijos hay que borrarlo. Pero debe mantenerse el resto de la estructura del árbol. Esto implica que los siguientes hermanos de A deben seguir siendo hijos del padre de A. Para lograrlo se debe modificar el valor de A, lo cual se hace en la línea 6, que enlaza al nodo que llama con el siguiente hermano de A (a menos que A sea la raíz de todo el árbol, en cuyo caso pasa a ser el árbol vacío). Este cambio es efectivo porque A es pasado por referencia.

Se pueden distinguir tres casos en los que A debe ser borrado, de acuerdo a su relación con el resto de la estructura.

Raíz En este caso, A no tiene hijos y su siguiente hermano es `NULL`, por lo que el resultado debe ser el árbol vacío, lo cuál es asignado correctamente en la línea 6.

Primer hijo A es el primer hijo de un nodo, llamémosle C, y por lo tanto se llega a él con la llamada de la línea 11. Supongamos que el siguiente hermano de A es B (que puede ser vacío). Como resultado del borrado, el primer hijo de C debe pasar a ser B (esto se indica con línea punteada en el diagrama de la izquierda). Esto se logra ya que, en la línea 11 lo que es pasado por referencia, y por lo tanto modificado en la línea 6 (`A=A->sH`) de la invocación, es el valor del primer hijo de C. Antes de la llamada se cumplía `C->pH==A, A->sH==B`, por lo que la modificación es `C->pH=A->sH`, y queda `C->pH==B`.



Siguiente hermano De manera similar, si A no es primer hijo de su padre es el siguiente hermano del nodo llamante, C, y es accedido con la llamada de la línea 12. Lo que se modifica en esa llamada es el siguiente hermano de C que pasará a ser B, que era el siguiente hermano de A. Se tiene `C->sH==A, A=A->sH==B`, por lo que se obtiene `C->sH==B`.

Para borrar el nodo se debe obtener una referencia a él con una variable auxiliar antes de la modificación.

Ejercicio 4 (10 puntos)

Considere la siguiente función que recibe un arreglo A de enteros de tamaño n:

```
bool F(int *A, unsigned int n) {
  bool res = true;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      if (i != j)
        res = res && (A[i] != A[j]);
  return res;
}
```

a) Explique brevemente qué retorna la función F, dado un arreglo de enteros de tamaño n.

b) ¿Cuál es el orden (O) de tiempo de ejecución para el peor caso de la función F? Justifique brevemente.

c) Si se sabe que el arreglo A sólo puede contener valores enteros en el rango $[0 : 2n]$, el problema que resuelve F ¿podría resolverse en un menor orden de tiempo de ejecución en el peor caso? Justifique en caso negativo y en caso afirmativo escriba una solución más eficiente que F en el peor caso, indicando su orden (O).

Solución:

a) F calcula si todos los elementos de A son diferentes. Devuelve *true* si son diferentes o *false* si hay al menos un par de elementos iguales.

b) El tiempo de ejecución es $O(n^2)$ ya que cada uno de los n índices de A se compara con los n índices de A en la sentencia `if (i !=j)`, y también se calcula n^2 veces las operaciones `j<n` y `j++` del bucle interno. Además, $n^2 - n$ veces se recalcula el valor de *res* en la operación de asignación. En el peor caso, cuando todos los elementos son diferentes, todas esas veces se hace también la comparación `A[i] != A[j]`.

c) Se puede resolver en tiempo $O(n)$ creando un arreglo de booleanos, B, con índices desde 0 hasta $2n$, el cual se inicializa en *false*. Se recorre A marcando *true* los índices de B que corresponden a los valores de A. Si se intenta marcar un elemento de B que ya fue marcado, entonces hay elementos repetidos. El tiempo es $O(n)$ porque hay que hacer $2n+1$ asignaciones *false* en B y en el peor caso una recorrida por los n elementos de A.

Listado 5: Solución en O(n)

```
1  bool F (int * A, unsigned int n) {
2    bool distintos = true;
3
4    // arreglo de booleanos inicializados en false
5    bool *B = new bool[2 * n + 1];
6    for (unsigned int i = 0; i <= 2 * n; i++)
7      B[i] = false;
8
9    unsigned int i = 0;
10   while ((i < n) && distintos) {
11     if (B[A[i]]) {
12       distintos = false;
13     } else {
14       B[A[i]] = true;
15       i++;
16     }
17   }
18   delete[] B;
19   return distintos;
20 }
```