**Algoritmos y Programación II (75.41, 95.15) – Curso Buchwald**

**2.do parcialito – 06/07/2020**

Resolvé los siguientes problemas en forma clara y legible. Podés incluir tantas funciones auxiliares como creas necesarias.

Debés resolver 3 de los siguientes ejercicios. Uno entre el 1 y 3, otro entre el 4 y 6, otro entre el 7 y 9. Los ejercicios a resolver están definidos por tu padrón. Podés revisar en esta planilla los que te corresponden.

1. Implementar para un árbol binario una primitiva `bool ab_completo(const ab_t* ab)` que determine si se trata de un árbol completo . Un árbol binario es completo si todos los niveles en los que hayan nodos se encuentran completos. Indicar y justificar la complejidad de la primitiva implementada. A efectos del ejercicio, la estructura del árbol es:

```
typedef struct ab {
    struct ab* izq;
    struct ab* der;
    void* dato;
} ab_t;
```

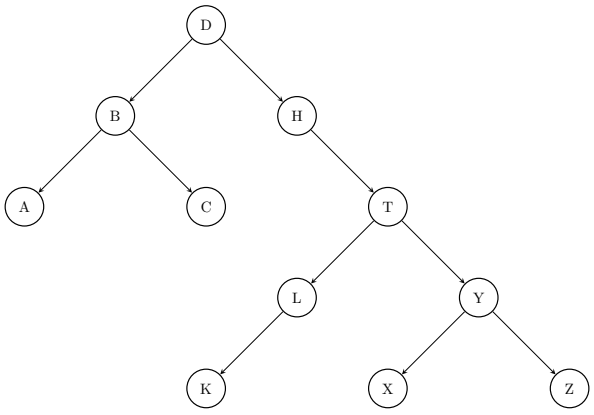
2. Implementar para un árbol binario una primitiva `lista_t* ab_primos_datos(const ab_t* ab, void* dato_buscado)` que devuelva **los datos** de los nodos primos a uno que se esté buscando. Dos nodos son primos si se encuentran en el mismo nivel dentro del árbol. Por simplificación, considerar que dos nodos *hermanos* (i.e. tienen mismo padre) son también primos. Suponer que en el árbol no hay elementos repetidos, y que se puede comparar los punteros por igualdad de dirección de memoria apuntada. En caso de no encontrarse el elemento, devolver una lista vacía. Indicar y justificar la complejidad de la primitiva implementada. A efectos del ejercicio, la estructura del árbol es:

```
typedef struct ab {
    struct ab* izq;
    struct ab* der;
    void* dato;
} ab_t;
```

3. Implementar para el ABB una primitiva `lista_t* abb_claves_debajo(const abb_t* abb, const char* clave)` que obtenga todas las claves del sub-arbol en el que dicha clave es raíz. Las lista devuelta debe quedar ordenada. En caso de no encontrarse la clave, devolver una lista vacía. Por simplicidad, suponer que las claves se comparan con `strcmp`[1]. Indicar y justificar la complejidad de la primitiva. En el árbol del ejemplo, invocando con clave `T`, debería devolverse `[K, L, T, X, Y, Z]`. A efectos del ejercicio, la estructura del árbol es:

```
typedef struct nodo_abb {
    struct nodo_abb* izq;
    struct nodo_abb* der;
    char* clave;
    void* dato;
} nodo_abb_t;
```

```
typedef struct abb {
     nodo_abb_t* raiz;
     size_t cantidad;
 } abb_t;
```




[1] Recordar que la función `strcmp(A, B)` (como cualquier función de comparación) devuelve un valor negativo cuando `A < B`, devuelve un valor positivo cuando `A > B`, y devuelve 0 cuando son equivalentes. Una forma fácil de entenderlo es que la expresión `strcmp(A, B) < 0` sería "equivalente" a `A < B`.

4. Implementar una función que dado un arreglo de $n$ números enteros positivos, encuentre en tiempo lineal el **menor** número posible (también entero positivo) que **no** se encuentre en el arreglo. Justificar la complejidad de la función implementada. Ejemplos:

```
[7, 8, 14, 13, 5]   --> 1
[9, 1, 2, 7, 9, 14] --> 3
[1, 2, 3, 4, 5, 6]  --> 7
```

5. Implementar una función que reciba un arreglo de $n$ números enteros y un número $K$ y determine, en tiempo lineal de $n$, si existe un par de elementos en el arreglo que sumen exáctamente $K$. Justificar la complejidad de la función implementada.

6. José trabaja en una tienda de ropa. Tiene una pila enorme de medias que tiene que juntar de a pares por colores (asumir talle único). Dado un arreglo de cadenas donde cada cadena representa un color, donde cada elemento del arreglo representa una media de dicho color, implementar una función de tiempo lineal (en la cantidad de medias) que nos indique cuántos pares se pueden armar para cada color. Esta información se puede devolver en un diccionario (clave: color, valor: cantidad de pares). Justificar la complejidad de la función implementada.

7. Definir si las siguientes afirmaciones son verdaderas o falsas, y justificar.

   a. Siempre que se trabaje con datos discretos me conviene ordenar usando Counting Sort.
   b. Para que RadixSort funcione correctamente, el algoritmo auxiliar a utilizar debe ser in-place.
   c. En Hopscotch hashing puede darse el caso que una búsqueda no sea en tiempo constante, según cómo sea la función de hashing.

8. Definir si las siguientes afirmaciones son verdaderas o falsas, y justificar.

   a. Para ordenar correctamente utilizando RadixSort, se debe ordenar los datos de la *cifra* menos significativa a la más significativa.
   b. BucketSort será lineal en tanto y en cuanto el algoritmo auxiliar para ordenar los buckets sea de tiempo lineal.
   c. En un AVL, tras una inserción, es necesario buscar en todo el árbol si se generó un desbalanceo.

9. Definir si las siguientes afirmaciones son verdaderas o falsas, y justificar.

   a. Si queremos ordenar por año todos los sucesos importantes ocurridos en el mundo desde el año 0 hasta la actualidad, CountingSort es una buena alternativa. Consideremos que todos los años hay sucesos importantes, y en este en particular sólo faltó la invasión zoombie (aúnque todavía queda diciembre...).
   b. RadixSort logra una complejidad lineal (dadas las condiciones adecuadas) reduciendo la cantidad de comparaciones que realiza entre los datos.
   c. Un potencial problema de Hash & Displace es que requerimos de varias funciones de hashing, y ni así podemos estar seguros que todo funcionará (salvo que sean realmente muchas).