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

2.do parcialito – 14/06/2021

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 `size_t ab_sin_nietos(const ab_t* ab)` que devuelva la cantidad de nodos en el árbol que no tienen nietos . Se le considera nieto de un nodo a un hijo de cualquiera de sus hijos. Es decir, un nieto es un hijo del hijo. Indicar y justificar la complejidad de la primitiva implementada, y el tipo de recorrido utilizado. 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 `size_t ab_algun_hermano(const ab_t* ab)` que devuelva la cantidad de nodos en el árbol que tienen un hermano. Se considera que un nodo tiene hermano si su nodo padre tiene otro hijo. **Nota: Tener en cuenta que si A tiene de hermano a B, entonces B tiene de hermano a A. Es decir, tanto A como B son nodos con hermanos y tanto A como B deber ser contabilizados**. Indicar y justificar la complejidad de la primitiva implementada, y el tipo de recorrido utilizado. 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 un árbol binario una primitiva `size_t ab_sin_hermanos(const ab_t* ab)` que devuelva la cantidad de nodos en el árbol que no tienen hermanos . Se considera que un nodo no tiene hermano si su nodo padre no tiene otro hijo. Indicar y justificar la complejidad de la primitiva implementada, y el tipo de recorrido utilizado. A efectos del ejercicio, la estructura del árbol es:

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

4. Implementar, para un **Hash Cerrado**, una **primitiva** que dado un Hash, un número `k` y una función `bool debe_duplicar_dato(const char* clave)`, y suponiendo que en el Hash todos los datos guardados son de tipo puntero a entero, duplique el valor del dato guardado para las primeras `k` claves para las que la función `debe_duplicar_dato` devuelva `true`. En caso de que el Hash tenga menos de `k` elementos, multiplicar por 2 el dato guardado de todas las claves para las que `debe_duplicar_dato` devuelva `true`. Indicar y justificar la complejidad de la primitiva implementada. La firma de la primitiva deberá ser: `bool duplicar_selectivo(const hash_t* hash, int k, bool (*debe_duplicar_dato)(const char* clave))`

5. Implementar, para un **Hash Abierto**, una **primitiva** que dado un Hash, un número `k` y una función `void aplicar(void* dato)`, aplique la función pasada por parámetro a los primeros `k` datos en el Hash. En caso de que el Hash tenga menos de `k` elementos, aplicar la función `aplicar` a todos los datos del Hash. Indicar y justificar la complejidad de la primitiva implementada. La firma de la primitiva deberá ser: `bool aplicar_selectivo(const hash_t* hash, int k, void (*aplicar)(void* dato))`

6. Implementar, para un **Hash Abierto**, una **primitiva** que dado un Hash, un número `k` y una función `bool debe_devolver_dato(const char* clave)`, devuelva una lista con los datos asociados a las primeras `k` claves para las que la función `debe_devolver_dato` devuelva `true`. En caso de que la lista resultante tenga menos de `k` elementos, completar hasta `k` elementos con `NULL`. Indicar y justificar la complejidad de la primitiva implementada. La firma de la primitiva deberá ser: `lista_t* obtener_selectivo(const hash_t* hash, int k, bool (*debe_devolver_dato)(const char* clave))`

7. Realizar el seguimiento de ordenar por Radix Sort el siguiente arreglo de cadenas que representan versiones. Cada versión tiene el formato `a.b.c`, donde cada valor `a`, `b` y `c` puede tener un valor entre 0 y 99. Considerar que se quiere que quede ordenado primero por la primera componente (`a`), luego por la segunda (`b`) y finalmente por la tercera (`c`). Tener en cuenta que, por ejemplo, `1.1.3` < `1.1.20`, `2.20.8` < `3.0.0`.

```
["10.4.2", "6.3.2", "10.1.4", "5.10.20", "3.2.1", "4.6.3", "2.1.5", "8.1.2", "6.3.1", "10.0.23"]
```

¿Cuál es el orden del algoritmo? ¿Es estable?

8. Hacer el seguimiento de counting sort, mostrando todos los pasos intermedios, para ordenar por año las siguientes películas:
   2005 - El cadáver de la novia 2003 - El gran pez 1989 - Batman 1990 - El joven manos de tijera
   2005 - El planeta de los simios 1996 - Marcianos al ataque 2003 - Sweeney Todd 2005 - Alicia en el país de las maravillas

   ¿Cuál es el orden del algoritmo? ¿Qué sucede con el orden de los elementos de un mismo año, respecto al orden inicial, luego de finalizado el algoritmo? Justificar brevemente. ¿Es estable?

9. Ordenar las siguientes direcciones IP usando radix sort, mostrando los pasos intermedios. Una IP es utilizada para identificar dispositivos en internet. Está conformada por 4 números de 0 a 255, separados por punto: `w.x.y.z`. El orden resultante debe quedar ordenado primero por la primera componente (`w`, luego la segunda (`x`), luego la tercera (`y`) y por último la cuarta (`z`).

`132.115.152.108`	`218.115.121.82`
`65.97.8.57`	`193.97.8.57`
`244.76.152.108`	`76.97.121.82`
`218.97.121.82`	`238.115.152.57`

   ¿Cuál es el orden del algoritmo? ¿Es estable?