Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC 2133 — Estructuras de Datos y Algoritmos

Interrogación 2

Segundo Semestre, 2014

Duración: 2 hrs.

1. Radix sort, cuyo pseudocódigo presentado en clases interpreta a cada clave como un número en base 10, se puede modificar de tal forma que interprete a cada clave como un número en una base arbitraria, b.

   a) **(1/3)** Dé una expresión para el tiempo de ejecución de Radix Sort, que dependa tanto del valor elegido para la base b como del conjunto de claves a ordenar, k.

   Solución: Supongamos que el arreglo A que queremos ordenar tiene K claves para ordenar digamos que $A = \{a_i | i \in [0, k)\}$. Además, supongamos que cada a_i tiene un valor entero s_i, entonces podemos escribir el pseudocódigo de radix sort como sigue:

   Data: A, n, k
   Result: A ordenado
   1: **for** $i \leftarrow 0; i < k; i++$ **do**
   2: **for** $j \leftarrow 0; j < s_i; j++$ **do**
   3: $contenedor[j] \leftarrow \emptyset$
   4: **for** $j \leftarrow 0; j < n; j++$ **do**
   5: movemos A_i al final del $contenedor[A_i \rightarrow a_i]$
   6: **for** $j \leftarrow 0; j < s_i; j++$ **do**
   7: unimos el $contenedor[j]$ al final de A

   Notemos que el primer bloque del algoritmo toma tiempo $O(s_i)$, el segundo bloque toma tiempo $O(n)$ y el tercer bloque toma tiempo $O(s_i)$, si sumamos obtenemos

   $$\sum_{i=1}^{k} O(s_i + n) = O(kn + \sum_{i=1}^{k} s_i) = O(n + \sum_{i=1}^{k} s_i)$$

   Ahora por ejemplo si los números son enteros entre $(0, b^k - 1)$ podemos ver que para alguna constante k, las llaves son numeros en base b de k digitos, utilizando la ecuacion anterior obtenemos que la complejidad de radix sort es

   $$O(k(n + b)) = O(kN) = O(n)$$

   Notar que este resultado depende de que k sea constante.

   b) **(2/3)** Comente sobre la veracidad de la afirmación: "dado un conjunto de claves K siempre existe una forma de hacer que Radix Sort funcione esencialmente igual a Counting Sort".

   Solución: Si ocupamos como base el valor más grande del arreglo de input, radix sort generara el mismo numero de colas que counting sort de contadores, luego es fácil ver que cada vez que radix sort agrega los datos a la cola correspondiente, counting sort suma en el contador respectivo. Y,

en genral, cada vez que Radix Sort opere con una de sus colas counting sort hará lo mismo con su contador. Luego, es posible intercambiar las operaciones sobre contadores y colas haciendo que esencialmente estos algoritmos funcionen de la misma manera.

2. *a*) **(1/4)** ¿De qué tamaño es la rama más corta que puede tener un árbol AVL de altura h? Justifique.
Solución: Vista en clases, $\lfloor \frac{h}{2} \rfloor$

b) **(3/4)** Considere la afirmación:

> Dado un entero m correspondiente al tamaño de la tabla de hash, es posible demostrar que $h_{a,b}(k) = ((ak + b) \mod p) \mod m$, donde $a, b \in \{0, \ldots, p-1\}$ y p es un primo "grande" define una familia de funciones de hash universales.

¿Cómo debe estar definida la propiedad de ser "grande" arriba para que la afirmación sea verdadera? Muestre cómo es que cuando se viola esa propiedad la familia de funciones deja de ser universal.

Solución: Sea el universo $U = \{0, ..., u-1\}$, sea p un número primo tal que $p \leq u$, y definamos $h_{a,b}(x) = ((ax+b) \mod p) \mod m$ donde a, b son elegidos aleatoreamente módulo p tal que $a \neq 0$, primero debemos ver para que $H = \{h_{a,b}\}$ es una familia de funciones universales notemos que $h(x) = h(y)$ solamente cuando $ax + b \equiv_{mod(p)} ay + b + im$, para algun entero i entre 0 y $\frac{p}{m}$. Si $x \neq y$, su diferencia es distinta de 0 y por tanto existe una función inversa para obtener a tal que $a \equiv_{mod(p)} im(x-y)^{-1}$. Luego, hay $p-1$ posibles elecciones para a ya que $a \neq 0$ y, variando i en el rango permitido, existen $\lfloor \frac{p}{m} \rfloor$ valores posibles, luego la probabilidad de colisión es:

$$\frac{\lfloor \frac{p}{m} \rfloor}{p-1}$$

Luego tenemos que para m constante

$$\lim_{p \to \infty} \frac{\lfloor \frac{p}{m} \rfloor}{p-1} = \frac{1}{m}$$

, es decir limita la probabilidad de colisión al tamaño de la tabla.

3. Los árboles $(2,3)$ son árboles binarios de búsqueda *multiway* que son como los árboles $(2,4)$, pero no permiten la existencia de 4-nodos.

a) Demuestre que si K es un conjunto de objetos, entonces es posible construir un árbol $(2,3)$ conteniendo exactamente los objetos de K que esté *perfectamente balanceado*, es decir, en donde cada rama tiene el mismo largo.

Solución: Procederemos por inducción.

- Caso Base $K = 1$, si el árbol tiene un solo elemento, entonces tiene un sólo nodo, luego todas sus hojas tendran la misma altura por lo que la propiedad se cumple.
- Hipoótesis de induccion, supongamos que para todo $0 \leq K \leq n$, se cumple que el árbol siempre se encuentra balanceado.
- Tesis de inducción: Por demostrar la propiedad para $n+1$ elementos, consideremos un árbol $(2,3)$ de n elementos, el cual se encuentra balanceado por hipótesis de inducción. Ahora si insertamos un elemento obtenemos 2 casos:
  - Si al agregar un elemento no ocurre overflow entonces no hay problemas ya que no se crearon nodos nuevos y por ende el arbol sigue balanceado.
  - Si al agregar un elemento ocurre un overflow se toma el elemento central del nodo y se sube al padre, si en el padre no ocurre overflow, terminamos y el arbol sigue balanceado, si ocurre overflow se sube el nodo central al padre de este nodo, se juntan los dos hijos centrales del primer nodo y se separa los extremos de este nodo en dos, este procedimiento

se repite hasta que no ocurra overflow, cuando terminamos nos damos cuenta que no creamos ninguna hoja extra por lo que el árbol resultante debe estar balanceado. Luego como la hipótesis implica la tesis la propiedad debe ser cierta.

b) Todo árbol $(2,3)$ es un árbol $(2,4)$. Este hecho por si solo, sin embargo, no prueba que posible usar el algoritmo de eliminación de los árboles $(2,4)$ directamente sobre los árboles $(2,3)$. Argumente convincentemente por qué sí es posible usarlo.

Solución: Supongamos que no es posible utilizar el algoritmo de eliminación de los árboles $(2,4)$ en los árboles $(2,3)$, sabemos que el algoritmo de los árboles $(2,4)$ es correcto, luego debe poder eliminar cualquier caso de un árbol $(2,4)$. Por otra parte, notemos que un árbol $(2,4)$ también contiene casos de eliminación de los arboles $(2,3)$, pero nuestro algoritmo no funcióna para la eliminación de los casos que corresponden a árboles $(2,3)$, pero esto contradice el hecho de que funciona para todos los casos de los árboles $(2,4)$ por tanto nuestra suposición de que el algoritmo no funciona para los árboles $(2,3)$ es incorrecta.

4. *a*) Escriba un pseudocódigo detallado de la operación *restructure*, que es la base de las operaciones de balanceo de los árboles AVL y Rojo-Negro. Suponga que la operación toma como argumento un árbol T y tres nodos en T: u, v y w tales que v es hijo de u y w es hijo de v.

Respuesta: lo importante de este algoritmo es que logre identificar los casos que pueden darse y actualize correctamente las referencias en el árbol.

Existen muchos algoritmos correctos, a continuación se plantea uno.

```
1: function RESTRUCTURE(T, u, v, w)
2:     if left[u] = v then
3:         if right[v] = w then
4:             LEFT-ROTATE(v)
5:         RIGHT-ROTATE(u)
6:     else                          ▷ casos simétricos
7:         if left[v] = w then
8:             RIGHT-ROTATE(v)
9:         LEFT-ROTATE(u)
```

b) Dé un algoritmo que haga $O(\mathit{altura}[T])$ llamados a *reestructure* para implementar la operación $\mathsf{split}(T,k)$, que retorna un par $\langle T_{\leq k}, T_{\geq k}\rangle$, donde

- T es un ABB y k es una clave.
- $T_{\leq k}$ es un ABB que contiene elementos en T cuya clave es menor o igual a k,
- $T_{\geq k}$ es un ABB que contiene elementos en T cuya clave es mayor o igual a k,
- Todo elemento de T está en $T_{\leq k}$ o en $T_{\geq k}$.

Respuesta: en esta pregunta no era necesario escribir un pseudocódigo detallado, bastaba con describir el algoritmo de forma precisa. Un algoritmo que funciona es el siguiente:

1) Insertamos un nodo u con clave k en el árbol.
2) Llamamos *restructure* sobre u, su padre y su abuelo.
3) Repetimos (2) hasta que u sea la raíz o hijo de la raíz.
4) Si u es hijo de la raíz, aplicamos la rotación adecuada para dejar a u como raíz.
5) Los árboles buscados corresponderán a $left[u]$ y $right[u]$.

Este algoritmo hará $O(\mathit{altura}[T])$ llamados a *restructure* por las siguientes razones. El paso (1) hará $O(\mathit{altura}[T])$ llamados a *restructure*. Además, cada vez que hacemos (2), u sube por lo menos un nivel y, por lo tanto, (2) se ejecutará menos de $\mathit{altura}[T]$ veces. Luego, el algoritmo cumple la cota mencionada.