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

IIC2133 — Estructuras de Datos y Algoritmos 2'2022

Interrogación 2

24 de octubre de 2022

Condiciones de entrega. Debe entregar **solo 3** de las siguientes 4 preguntas.

Puntajes y nota. Cada pregunta tiene 6 puntos más un punto base. La nota de la interrogación será el promedio de las notas de las 3 preguntas entregadas.

Uso de algoritmos. De ser necesario, en sus diseños puede utilizar llamados a cualquiera de los algoritmos vistos en clase. No debe demostrar la correctitud o complejidad de estos llamados.

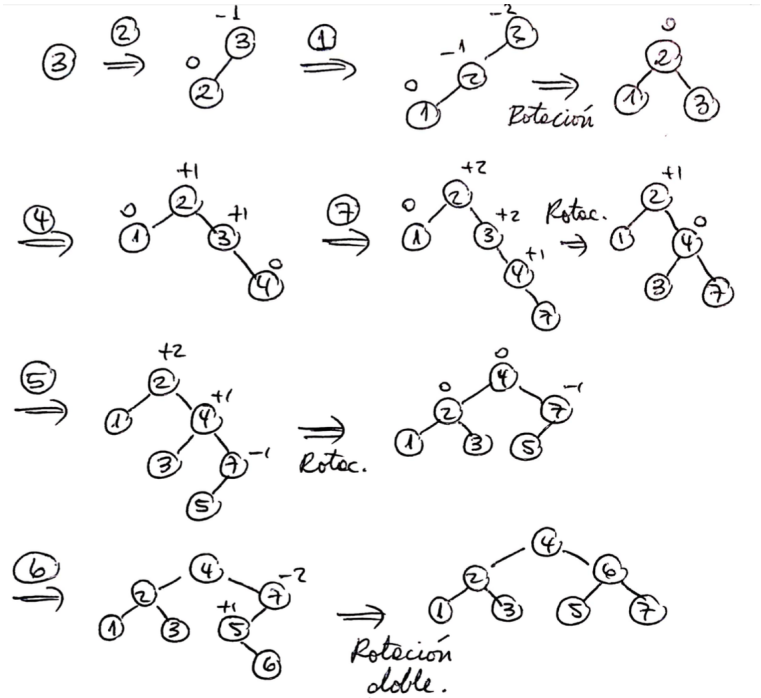
1. Árboles de búsqueda

En clases presentamos dos nociones de balance en árboles binarios de búsqueda: los árboles AVL y los rojo-negro. Argumentamos que ambos tipos de balance logran altura $\mathcal{O}(\log(n))$ cuando el árbol tiene n nodos.

(a) [1.5 ptos.] Muestre un diagrama del proceso de inserción y balance al insertar las siguientes llaves en un árbol AVL inicialmente vacío

$$3, 2, 1, 4, 7, 5, 6$$

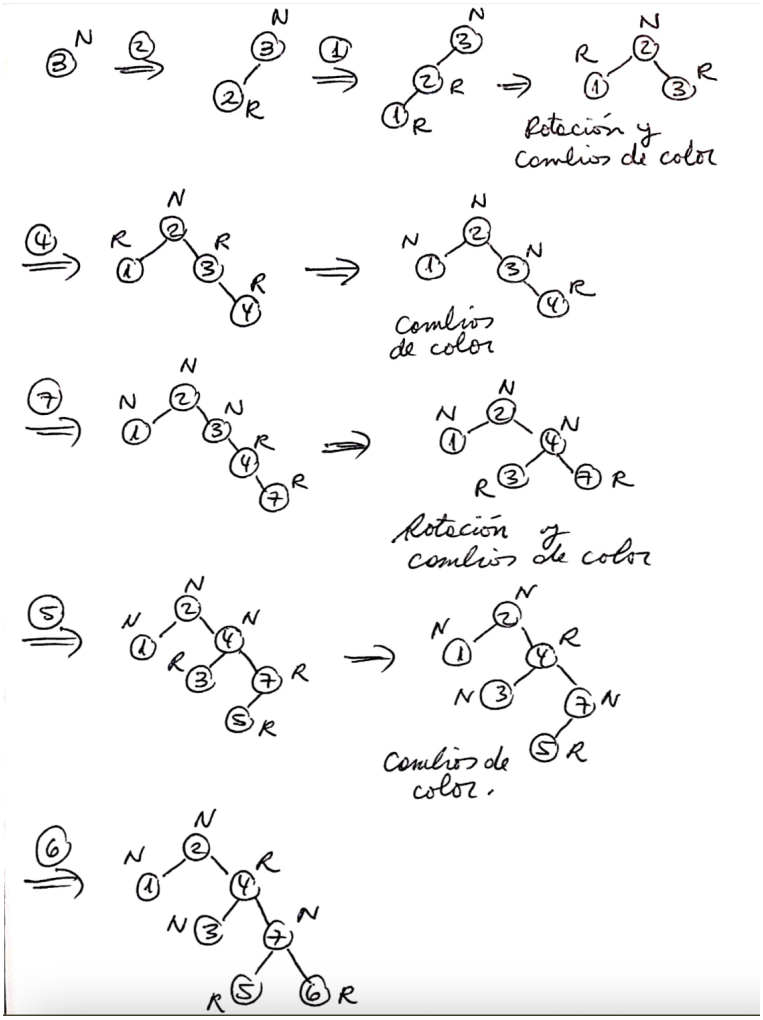
Solución.



Asignación de puntajes.
0.1 por inserción del 2
0.3 por inserción del 1 (rotación simple)
0.1 por inserción del 4
0.3 por inserción del 7 (rotación simple)
0.3 por inserción del 5 (rotación simple)
0.4 por inserción del 6 (rotación doble)

(b) [1.5 ptos.] Repita el proceso del inciso (a) para un árbol rojo-negro inicialmente vacío.

Solución.



Asignación de puntajes.
0.15 por inserción del 2
0.3 por inserción del 1 (rotación y cambio de color)
0.3 por inserción del 4 (cambio de color)
0.3 por inserción del 7 (rotación y cambio de color)
0.3 por inserción del 5 (cambio de color)
0.15 por inserción del 6

(c) [1.5 ptos.] Sin cambiar aristas, ¿es posible dar una coloración del árbol en (a) de forma que cumpla ser rojo-negro? ¿Es posible considerar el árbol en (b) como AVL? Justifique.

Solución.

El árbol obtenido en (a) admite la coloración en que cada nodo es negro, satisfaciendo las condiciones de rojo-negro. Esto debido a que es perfectamente balanceado y todos los caminos desde un nodo a sus hojas negras descendientes pasa por la misma cantidad de nodos negros.

El árbol obtenido en (b) no es AVL pues la raíz tiene un subárbol izquierdo con altura 1, mientras que el hijo derecho tiene arltura 3. La diferencia es mayor a 1 y por lo tanto no se cumple la propiedad AVL.

Asignación de puntajes.
0.3 por indicar que (a) admite una coloración
0.45 por entregar coloración y justificar su validez
0.3 por indicar que (b) no es AVL
0.45 por argumentar en base a alturas

(c) [1.5 ptos.] Si bien la complejidad asintótica de la búsqueda en ambos tipos de árboles es la misma, el tiempo en la práctica puede ser diferente. Dada una misma secuencia de inserciones iniciales, argumente en qué tipo de árbol (AVL o rojo-negro) es más probable que la búsqueda de una llave tarde más.

Solución.

Un árbol rojo-negro usa una noción de balance menos exigente que el criterio AVL, pues permite que los subárboles hijos de un nodo cualquiera tengan alturas que difieren en más de 1 unidad. Esto causa que la altura del árbol en la práctica pueda ser mayor en rojo-negro que en AVL. Por esto, la búsqueda en rojo-negro puede ser más lenta que en AVL en la práctica.

Asignación de puntajes.
0.5 por indicar el tipo de árbol con mejor/peor tiempo práctico de búsqueda
1.0 por justificar en base a sus criterios de construcción

2. Funciones y tablas de hash

Una función de hash $f : K \to \{0, \ldots, m-1\}$ se dice **incremental** si, dadas dos llaves $k, k' \in K$ tales que k' es ligeramente diferente a k, el valor $f(k')$ puede obtenerse rápidamente a partir de $f(k)$.

(a) [2 ptos.] Denotamos por $\{a, b\}^*$ el conjunto de las palabras formadas por los caracteres a y b. Considere la función de hash $h : \{a, b\}^* \to \{0, \ldots, 6\}$ dada por

$$h(s_0 s_1 \ldots s_p) = \left(\sum_{i=0}^{p} \#(s_i) \right) \textbf{ mod } 7$$

donde $\#(a) = 1$ y $\#(b) = 2$. Para probar que h es incremental, muestre cómo calcular $h(s_0 s_1 \ldots s_{p-1} s_p)$ a partir de $h(s_0 s_1 \ldots s_{p-1})$, justificando que toma tiempo $\mathcal{O}(1)$. *Hint:* utilice las propiedades de **mod** y asuma que las operaciones suma y **mod** son $\mathcal{O}(1)$.

Solución.

Tenemos que

$$
\begin{aligned}
h(s_0 s_1 \dots s_p) &= \left(\sum_{i=0}^{p} \#(s_i) \right) \textbf{ mod } 7 && \text{(por def. de } h\text{)} \\
&= \left(\sum_{i=0}^{p-1} \#(s_i) + \#(s_p) \right) \textbf{ mod } 7 && \text{(separando la suma)} \\
&= \left[\left(\sum_{i=0}^{p-1} \#(s_i) \right) \textbf{ mod } 7 + \#(s_p) \textbf{ mod } 7 \right] \textbf{ mod } 7 && \text{(módulo de la suma)} \\
&= [h(s_0 s_1 \dots s_{p-1}) + \#(s_p) \textbf{ mod } 7] \textbf{ mod } 7 && \text{(por def. de } h\text{)} \\
&= [h(s_0 s_1 \dots s_{p-1}) + \#(s_p)] \textbf{ mod } 7 && \text{(pues } \#(s_p) \in \{1, 2\}\text{)}
\end{aligned}
$$

con lo cual, sin importar el largo p del string que se quiere hashear, su valor de hash se calcula a partir de un valor ya conocido mediante operaciones $\mathcal{O}(1)$.

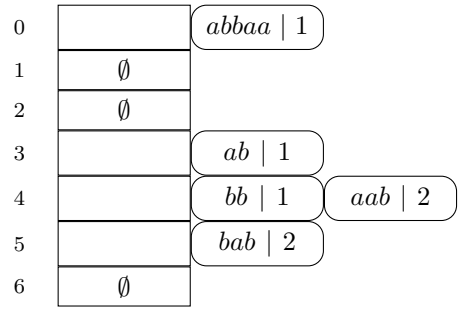
Asignación de puntajes.
0.3 por reescribir el valor como suma separada
0.5 por usar propiedad del módulo para separar
0.5 por llegar que depende de $h(s_0 s_1 \dots s_{p-1})$
0.2 por concluir respecto a la complejidad del proceso

(b) [2 ptos.] Suponga que en una ventana de tiempo recibimos varias palabras de $\{a, b\}^*$ y nos interesa contar cuántas veces recibimos cada una. Para esto, utilizaremos una tabla de hash T de tamaño 7 que almacene las apariciones por palabra y que usa la función de hash h del inciso (a). Si T usa encadenamiento para resolver colisiones, muestre un diagrama con el estado final de T luego de recibir

$$aab,\ ab,\ bb,\ bab,\ aab,\ bab,\ abbaa$$

Solución.

Considerando que para una celda de la tabla, la lista ligada contiene elementos de la forma $(e.\texttt{string}, e.\texttt{count})$ que contienen respectivamente, el string y el conteo de apariciones del mismo, el diagrama es



Asignación de puntajes.
0.3 por tamaño correcto de la tabla
0.2 por usar listas ligadas (obligatorio)
1.0 por ubicar correctamente los elementos en su celda
Observación. Se admiten otros modelos para almacenar los datos. Lo importante es que el número de apariciones se pueda deducir, ya sea directamente (como en la solución mostrada) o de forma implícita (por ej guardando todas las apariciones por palabra).

(c) [2 ptos.] Una palabra w' es prefijo de w si aparece al principio de w. Considerando la tabla T del inciso (b), proponga el pseudocódigo de un algoritmo que, dada una palabra $w \in \{a, b\}^*$, imprima las apariciones almacenadas en T de **cada prefijo** de w, imprimiendo 0 si no aparece.

(d) [**bonus 1 pto.**] Si su algoritmo en (c) aprovecha que h es incremental, obtiene 1 punto adicional.

Solución.

Usando el modelo planteado en la parte (b), el algoritmo es

input : Tabla de hash T, palabra $w \in \{a, b\}^*$

```
CountStrings (T, w):
1    n ← |w|
2    h ← 0
3    for i = 0 ... n − 1 :
4        h ← (h + w[i]) mod 7
5        found ← false
6        for e ∈ T[h] :
7            if e.string = w[0 ... i] :
8                found ← true
9                printf(e.string, e.count)
10               break
11       if not found :
12           printf(e.string, 0)
```

Asignación de puntajes.
0.5 por ubicar correctamente un prefijo en la celda de la tabla
0.5 por ubicar correctamente un prefijo en la celda de la lista correspondiente
0.5 por cubrir todos los prefijos
Bono 1.0: si el valor de hash se calcula como en la solución mostrada, es decir, se usa el valor anterior para calcular el próximo.

3. Ordenación lineal

Para procesar la información obtenida desde el telescopio espacial James Webb, los objetos del campo de observación son identificados utilizando el Guide Star Catalog (GSC) el cual asocia a cada objeto un identificador de la forma: `fffff0nnnnn` en que `fffff` es un valor alfanumérico (0-9,A-Z) que identifica la región del espacio en que se encuentra el objeto y `nnnnn` es un correlativo del objeto en esa región (el 0 se usa como separador). El procesamiento requiere ordenar muy eficientemente (con eficiencia lineal ya que se deben ordenar 100 mil objetos distintos cada vez) los datos obtenidos desde múltiples campos de observación distintos, utilizando como criterio de orden dicho identificador.

(a) [2 ptos.] Aplique RadixSort con CountingSort para realizar lo solicitado, detallando los ajustes necesarios al pseudo código visto en clases para que sean adecuados a las características del problema planteado.

Solución.

El orden es el "natural", menos significativo a la derecha y más significativo a la izquierda (de 0 a $d-1$ con $d = 11$). Modifica Radix sort para incorporar la cardinalidad del dominio del digito a ordenar en la llamada a counting sort, y modifica counting sort para utilizar dicho parámetro:

```
RadixSort (A, d):
    for j = 0 ... d − 1 :
        if j ≥ 0 ∧ j < 6 :        ▷ rango numérico, incluye al 0 central
            CountSort(A, j, 10)
        else:        ▷ rango alfanumérico
            CountSort(A, j, 37)
```

input : Arreglo $A[0 \ldots, n-1]$,j dígito a ordenar de A, k valores posibles

```
CountSort(A, j, k):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
    for m = 0 ... n − 1 :
        C[A[j][m]] ← C[A[j][m]] + 1        ▷ A[j][m] es el dígito j en la palabra m de A
    for p = 1 ... k :
        C[p] ← C[p] + C[p − 1]
    for r = n − 1 ... 0 :
        B[C[A[r]]] ← A[r]
        C[A[r]] ← C[A[r]] − 1
    return B
```

Asignación de puntajes.
1.0 por Radix Sort distinguiendo los dos escenarios
1.0 por CountSort modificado
Observación. Se puede incluir el manejo de los dos casos en el mismo CountingSort

(b) [2 ptos.] Utilizando el resultado de (a) proponga el pseudo código para permitir seleccionar si se desea ordenar en forma ascendente o descendente, manteniendo la complejidad de tiempo en $\mathcal{O}(n)$.

Solución.

Usando la parte (a) se obtiene el orden ascendente en $\mathcal{O}(n)$, si piden el orden descendente basta con recorrer el resultado de (a) y entregarlo en orden inverso, esto también es $\mathcal{O}(n)$ y como se "suma", el desempeño global sigue en $\mathcal{O}(n)$.

(c) [2 ptos.] Un compañero sugiere reemplazar CountingSort por QuickSort. Detalle el impacto que esto tendría desde el punto de la correctitud del algoritmo y su desempeño.

Solución.

QuickSort es $\mathcal{O}(n \log(n))$ y no es estable, luego el algoritmo si bien termina (al ordenar todos los dígitos) no cumple su propósito, ya que no entrega la salida ordenada correctamente. Además el desempeño pasa de ser $\mathcal{O}(n+d) = \mathcal{O}(n)$ a ser $\mathcal{O}(n \log(n) + d) = \mathcal{O}(n \log(n) + d)$.

4. Backtracking

Para asegurar la conectividad del trasporte en el extremo sur del país existen tramos en los cuales se utilizan barcazas para llevar vehículos (autos particulares y camiones) entre dos puntos que no tienen conectividad por tierra. La capacidad de la barcaza se define en función de los metros lineales de vehículos que puede acomodar (4 filas de vehículos de máximo 15 metros cada fila son 60 metros lineales de capacidad máxima) y el peso máximo total que puede transportar (por ejemplo 240.000 kilos de carga). Así una barcaza B se define como

(`B.n_filas`, `B.m_por_fila`, `B.max_carga`).

Los vehículos V que están a la espera de transporte están en una fila y tienen determinado su largo y peso (`V.largo`, `V.peso`) expresados en metros y kilogramos.

(a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.

Solución.

Variables las filas (1 o 4 según como lo aborden, con su capacidad en metros y vehículos cargados), Dominios vehículos a cargar en las filas y sus atributos, Restricciones la cantidad de metros lineales disponibles y carga total disponible en la barcaza (solo una barcaza).

(b) [3 ptos.] Diseñe un algoritmo para definir qué vehículos de la fila transportar de modo de maximizar la cantidad de vehículos sin superar la capacidad de la barcaza (en metros lineales totales y la carga máxima de la misma). **No considere** la capacidad de cada fila de la barcaza, sino la **capacidad total**.

Solución.

Dado que se pide maximizar, debe buscar todas las soluciones, evaluar que no se supera la capacidad de peso y largo para detener la recursión e ir guardando la solución que tiene la mayor cantidad de vehículos hasta el momento.

input : X, D, R parámetros de backtracking, M la mejor solución encontrada

`IsSolvableAll` (X, D, R, M)**:**
- **if** $X.metros\_utilizados \leq metros\_barcaza$ **:**
  - **if** $X.cantidad\_vehiculos > M.cantidad\_vehiculos$ **:**
    - $X \leftarrow M$
  - **return true**
- **for** $v \in D_x$ **:**
  - **if** *agregar vehiculo v a la fila no supera peso max* **:**
    - $X \leftarrow v$
    - **if** `IsSolvableAll` *(X,D,R,M)* **:**
      - Se marca M como solución
    - Sacar v de la fila para probar otro vehiculo
- **return false**

(c) [2 ptos.] Modifique su algoritmo anterior para que entregue en qué fila de la barcaza va cada vehículo a transportar, al maximizar la cantidad de vehículos sin superar la capacidad de la barcaza.

Solución.

Utilizando el resultado de (b) se tiene una lista de vehículos que cumple la restricción de peso y metros lineales totales. Con eso se requiere un algoritmo que encuentre un resultado que permita ubicar estos vehículos en las filas disponibles. Pueden terminar con "no hay solución" cuando no se puede fraccionar una fila, o quitar un vehículo para encontrar una sub solución.