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

IIC2133 — Estructuras de Datos y Algoritmos 1'2023

Interrogación 2

5 de junio de 2023

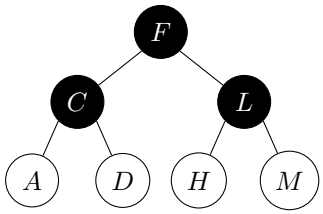
Condiciones de entrega. Debe entregar **solo** 3 de las siguientes 4 preguntas.

Nota. Cada pregunta tiene 6 puntos (+1 punto base). La nota es el promedio de las 3 preguntas entregadas.

Uso de algoritmos. En sus diseños puede utilizar llamados a cualquiera de los algoritmos vistos en clase. No debe demostrar la correctitud o complejidad de estos llamados, salvo que se especifique lo contrario.

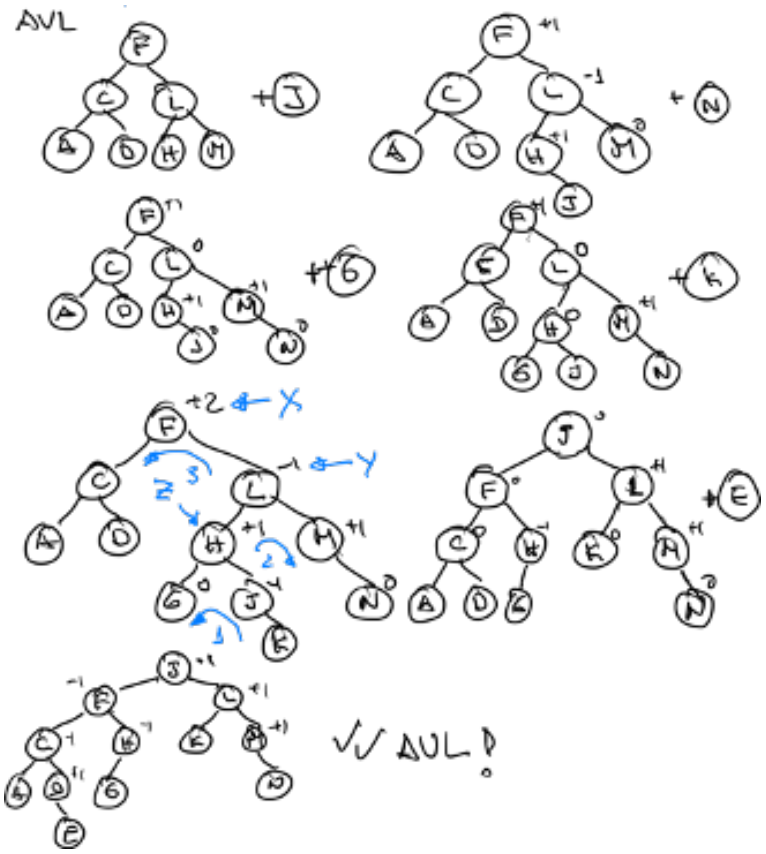
1. Árboles de búsqueda

(a) [4 ptos.] Considere el siguiente árbol $\mathcal{T}$ que es AVL y además rojo-negro (tomando las hojas A, D, H, M como rojos).



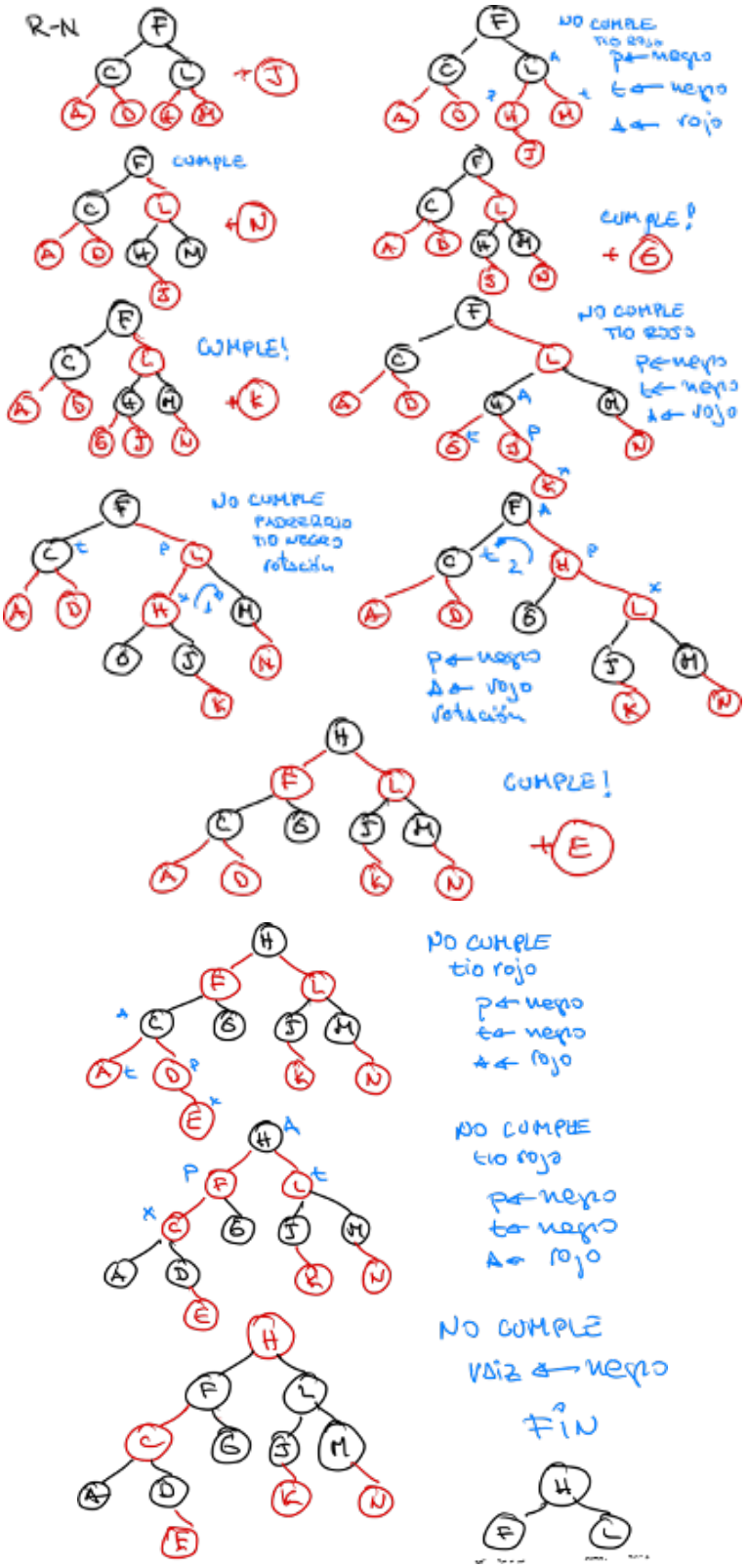
Muestre el resultado de insertar, en el mismo orden, la secuencia de llaves J, N, G, K, E cuando $\mathcal{T}$ se considera

(i) [2 ptos.] un árbol AVL.
Solución.



Puntajes.
0.4 por cada letra insertada correctamente en un árbol obtenido correctamente.

(ii) [2 ptos.] un árbol rojo-negro, donde las hojas de $\mathcal{T}$ son todas rojas inicialmente.
Solución.



Puntajes.
0.4 por cada letra insertada correctamente en un árbol obtenido correctamente.

(b) [2 ptos.] Suponga que deseamos implementar un árbol binario de búsqueda de gran tamaño (muchos

valores de llaves diferentes) y un volumen muy grande de inserciones. ¿Qué tipo de ABB elegiría entre AVL y rojo-negro? Justifique su respuesta.

Solución.

Esta respuesta depende del contexto y de supuestos que puedan considerarse. Si nos centramos solo en la diferencia comparativa de la inserción, los árboles rojo-negro son más rápidos en la práctica por cuanto requieren menos operaciones para rebalancear. Si la secuencia de inserciones produce un árbol más desbalanceado que el correspondiente AVL, entonces puede haber una desventaja al usar rojo-negro. Si asumimos que las inserciones son üniformes", entonces rojo-negro permite tener inserciones más rápidas de forma general.

Puntajes.

1.0 por referirse a la diferencia en las inserciones sin considerar contexto.

1.0 por justificar su decisión de acuerdo a supuestos pertinentes (esta parte es bien libre y depende de cada respuesta. Lo importante es que hayan justificado)

2. DFS y backtracking

Sea $G = (V, E)$ es un grafo dirigido. Un tour de largo n en G es una secuencia $u_0, \ldots, u_n$ tal que $u_i \in V$, cada $(u_i, u_{i+1}) \in E$ para todo $0 \leq i < n$ y $u_0 = u_n$ (todos los demás nodos deben ser distintos entre sí).

(a) [2 ptos.] Describa cómo modificar el algoritmo DFS estudiado en clases, para responder si G tiene algún tour de largo 3, es decir, si es que existen nodos a, b, c diferentes tales que forman un triángulo dirigido $a \to b \to c \to a$. No necesita dar el pseudocódigo, basta con explicar qué cambios son necesarios.

Solución.

Se puede modificar la versión de DFS vista con tiempos, de forma que se recorre el grafo de forma usual, pero se añade un atributo adicional, e.g. `nodo.attr` que indica un orden correlativo de nodos a medida que se descubren. La diferencia con el tiempo es que este atributo debe actualizarse también para el nodo de inicio de un llamado, en caso de que no se encuentre un ciclo en un cierto llamado de `DFSVisit`. El valor inicial debe ser al menos 3 unidades mayor que el último valor usado, para no inducir a error en la detección. Cuando se inicia un nuevo `DFSVisit` desde un nodo u, se van incrementando los atributos `.attr` de sus descendientes. Si se llega desde u a un nodo v que ya tiene seteado este valor y es $v.\texttt{attr} = u.\texttt{attr} - 2$, significa que es un ancestro en un triángulo.

Puntajes.

1.0 por justificar que se usa un atributo nuevo y que se va incrementando como los tiempos.

1.0 por mencionar la diferencia con los tiempos usuales, asegurándose de que cada nuevo llamado de `DFSVisit` tiene valores estrictamente más grandes que los anteriores, para detectar correctamente el triángulo

Observación: se aceptan soluciones alternativas que resuelvan el problema. No se requiere una explicación en pseudocódigo.

Un tour es hamiltoniano si visita a todos los nodos de G. Si tal tour existe, decimos que G es hamiltoniano.

(b) [1 pto.] Para responder si G es hamiltoniano usando backtracking proponga variables, dominios y restricciones adecuadas. *Pista:* visualice un vector $\langle u_1, \ldots, u_n \rangle$ que indica en qué orden visitar nodos. ¿Cuántos elementos tiene el vector? ¿Cuál es el dominio de sus coordenadas?

Solución.

Variables $X = \{x_1, \ldots, x_n\}$, donde $n = |V| + 1$ pues se quiere codificar un ciclo que tiene a todos los nodos de G y repite el inicio para ser cerrado.

Dominio V para cada x_i, pues representan el i-ésimo nodo escogido en el camino

Restricción de que todo $x_i \neq x_j$ salvo por el primer y último elemento

Puntajes.

0.5 por especificar cantidad y tipo de variables.

0.5 por incluir correctamente las restricciones en su esquema

Observación: se aceptan otras propuestas acordes a la solución que hagan en (c).

(c) [3 ptos.] Plantee un algoritmo de backtracking para responder si G es hamiltoniano.
Solución.
Asumimos que el algoritmo tiene acceso a un arreglo global $H[0 \dots n-1]$ donde se almacenan los nodos visitados en orden según se visitan. De esta forma, $H[i]$ es el nombres o etiqueta del nodo que se visita en la posición i en un posible tour hamiltoniano.

input : Número $k \in \{0, \dots, n+1\}$ que indica el k-nodo a asignar

```
Ham(k):
1   if k = n + 1 :
2       return True
3   for v ∈ Vecinos(H[k − 1]) :
4       if (v no ha sido visitado) OR (k = n AND H[0] = v) :
5           H[k] ← v
6           if Ham(k + 1) :
7               return True
8   return False
```

El llamado inicial al algoritmo es $\texttt{Ham}(0)$ y su retorno indica si existe o no un tour hamiltoniano. Adicionalmente, en tal caso queda almacenado en el arreglo H.

Puntajes.
1.0 por estructura recursiva, necesaria en backtracking.
0.5 por detección correcta de caso base (éxito)
0.5 por condición de nodo no visitado como chequeo de factibilidad
0.5 por condición de nodo visitado que coincide con el inicial en caso de ser el último como chequeo de factibilidad
0.5 por asignación y retorno recursivo
Observación: Otras versiones deben cumplir con el objetivo para estar correctas. Se puede asumir la existencia de métodos que entregan vecinos u otros datos del grafo, tal como en esta pauta.

3. Algoritmos codiciosos y programación dinámica

Sea $A = \{a_1, \dots, a_n\}$ un conjunto de naturales (distintos) y K una cota natural. El problema de *subset sum* responde si acaso existe un subconjunto $S \subseteq A$ de forma que la suma de los elementos de S sea exactamente igual a K. Para resolver este problema, consideremos su versión como problema de optimización: determinar un $S \subseteq A$ tal que la suma de sus elementos es máxima y además no supera la cota K.

(a) [1 pto.] Considere un algoritmo "codicioso" que revisa los elementos $a_1, \dots, a_n$ en ese orden y determina si se debe incluir a_i con la siguiente estrategia codiciosa: *se le incluye si, y solo si, hacerlo no supera la suma permitida por la cota.* Demuestre que esta estrategia no es correcta.
Solución.
Consideramos el conjunto $A = \{2, 3, 6\}$ y la cota $K = 6$. En este caso, el algoritmo codicioso selecciona el subconjunto $S_{\text{greedy}} = \{2, 3\}$ con suma 5. Pero el óptimo es $S_{\text{opt}} = \{6\}$. Vemos que la estrategia codiciosa no es correcta.
Puntajes.
0.5 por construir conjunto y cota.
0.5 por entregar el resultado de la estrategia codiciosa y el óptimo

(b) [2 ptos.] Dado un conjunto $A = \{a_1, \dots, a_n\}$ y cota K, sea $f(k, T)$ la suma máxima lograble cuando se considera el subproblema de optimización para $\{a_1, \dots, a_k\}$ con $k \leq n$ y tomando la cota $T \leq K$. Determine una ecuación de recurrencia para $f(k, T)$, especificando casos base y borde de ser necesario.
Solución.
Proponemos la siguiente función recursiva

$$f(k, T) = \text{máx}\{f(k-1, T), f(k-1, T-a_k) + a_k\}$$

donde el primer término recursivo denota el caso en que no se escoge el elemento a_k, y el segundo, el caso en que sí se toma y aporta a_k al total. Las condiciones de borde cuando se revisan todos los elementos ($k = 0$) depende de la suma disponible T según

$$f(0,T) = \begin{cases} 0 & \text{si } T \geq 0 \\ -\infty & \text{si } T < 0 \end{cases}$$

Puntajes.
0.3 por caso en que no se toma a_k.
0.3 por caso en que se toma a_k.
0.2 por comparación con máx para decidir
0.2 por casos borde

(c) [2 ptos.] Plantee el pseudocódigo de un algoritmo de programación dinámica que permita obtener la suma óptima para un conjunto $A = \{a_1, \ldots, a_n\}$ y cota K, e indicando en qué estructura de datos almacenará los óptimos de los subproblemas.
Solución.

input : Índice $k \in \{1, \ldots, n\}$ y cota T
output: Suma óptima

`SubsetSum`(k, T)**:**

```
1   if k = 0 AND T ≥ 0 :
2       return 0
3   if k = 0 AND T < 0 :
4       return −∞
5   else:
6       if M[k][T] ≠ ∅ :
7           return M[k][T]
8       else:
9           M[k][T] ← máx{SubsetSum(k − 1, T), SubsetSum(k − 1, T − a_k) + a_k}
10          return M[k][T]
```

donde M es un arreglo de arreglos global, donde se guardan los subóptimos.

Puntajes.
0.5 por casos base
1.0 por llamados a subproblemas.
0.5 por almacenar resultados parciales en alguna estructura de datos especificada

(d) [1 pto.] Describa cómo responder a la pregunta de si existe o no un subconjunto con suma exactamente K y explique cómo determinar el subconjunto a partir de la estructura usada en su solución en (c). No necesita entregar un pseudocódigo, sino que basta con una descripción a alto nivel.
Solución.
A partir del arreglo M, se comienza revisando $M[n][K]$. Este valor se compara con todos los valores $M[n-1][T]$ para los T almacenados. Si coincide con alguno, significa que el dato a_n no se incluye. Si coincide con alguno añadiendo a_n, significa que sí se incluye. Este proceso se repite revisando hacia atrás, deduciendo si cada elemento se incluye o no.
Puntajes.
1.0 por explicar cómo deducir si un dato se incluye o no.

4. Orden lineal y hashing

En el sistema de control de embarque de pasajeros en un aeropuerto se utiliza una tabla de hash T de tamaño M con encadenamiento para llevar un registro de los pasajeros autorizados para abordar el vuelo. Esta tabla almacena el identificador del pasajero (el RUT utilizado como llave), el código del vuelo y el

código de equipaje asociado a cada pasajero. Así, dado el RUT es posible recuperar de manera muy eficiente los demás datos asociados.

(a) [2 ptos.] Proponga los algoritmos en pseudocódigo para los siguientes métodos:

(i) [1 pto.] `agregarPasajero(r,v,e)`: Recibe como entrada el RUT del pasajero, el código del vuelo y el código de equipaje. Agrega los datos a la tabla T correctamente.
Solución.

input : RUT r, código de vuelo v, código de equipaje e

`agregarPasajero`(r, v, e)**:**
1 $i \leftarrow$ `hash`(r)
2 `insertarEnLista`$(T[i], r, v, e)$

donde `insertarEnLista` inserta la información del pasajero en la lista ligada de colisiones de la celda $T[i]$.
Puntajes.
0.5 por obtención de índice de celda con la función de hash.
0.5 por insertar en la lista de colisiones

(ii) [1 pto.] `buscarPasajero(r)`: Recibe como entrada el RUT del pasajero. Si lo encuentra, devuelve el código del vuelo y el código de equipaje, y si no lo encuentra devuelve -1 indicando que el pasajero no está registrado.
Solución.

input : RUT r

`buscarPasajero`(r)**:**
1 $i \leftarrow$ `hash`(r)
2 $p \leftarrow$ `buscarEnLista`$(T[i], r)$
if $p \neq \varnothing$ **:**
return p
return -1

donde `BuscarEnLista` itera en la lista de la celda $T[i]$ hasta encontrar el rut r.
Puntajes.
0.5 por obtención del índice de la tabla.
0.5 por buscar y retornar en la lista.

(b) [2 ptos.] Proponga el pseudocódigo para el método `pasajerosRegistrados()` que, a partir de la tabla T, permite obtener en tiempo lineal $\mathcal{O}(n)$ la nómina (código de vuelo, RUT) de todos los pasajeros registrados en un momento dado, ordenada por código de vuelo. Asuma que el código del vuelo es de la forma AA9999 (dos letras seguidas de 4 números). *Pista:* Puede utilizar como apoyo una estructura de datos adicional, asumiendo que el factor de carga de T es siempre menor a 1, y aplicar/modificar algoritmos vistos en clases en su solución.
Solución.

`pasajerosRegistrados`()**:**
1 $R \leftarrow$ arreglo de M celdas
2 **for** *e elemento en alguna lista de la tabla de hash* **:**
3 Guardar en R los datos e.`vuelo`, e.`rut` de e
return `RadixSort`(R)

Observamos que R tiene el tamaño adecuado, dado que como el factor de carga es menos a 1, hay menos de M pasajeros en la tabla. Además, se asume que Radix ordena R en base al atributo código de vuelo.

Puntajes.
1.0 por contrucción de una estructura que contenga la información requerida.
1.0 por ordenar usando un algoritmo en tiempo lineal.

(c) [2 ptos.] Un problema habitual es el equipaje perdido. Proponga un algoritmo en pseudocódigo para el método `buscarEquipaje(e)`: Recibe como entrada un código de equipaje (un número de 15 dígitos) y si lo encuentra, retorna el RUT y el código de vuelo del pasajero al que pertenece, -1 si no lo encuentra. Esta función debe operar en tiempo $\mathcal{O}(1)$. *Pista:* considere modificar primero `agregarPasajero()` de la parte (a) de modo de contar con una tabla de hash auxiliar E de tamaño M con encadenamiento que utilice como llave el código de equipaje y lo relacione con el RUT del pasajero.
Solución.

Primero modificamos `agregarPasajero` según

input : RUT r, código de vuelo v, código de equipaje e

`agregarPasajero`(r, v, e)**:**
1 $i \leftarrow$ `hash`(r)
2 $j \leftarrow$ `hash`$'(e)$
3 $p \leftarrow$ `insertarEnLista`$(T[i], r, v, e)$
4 `insertarEnLista`$(E[j], e, p)$

donde E es una segunda tabla de hash para los equipajes como llave (con función de hash `hash`$'$), y donde cada elemento almacenado en las listas de colisiones contiene un puntero p a la información del pasajero en la primera tabla, el cual puede ser fácilmente retornado por el método que inserta en la tabla T.

input : Código de equipaje e

`buscarEquipaje`(e)**:**
1 $i \leftarrow$ `hash`$'(e)$
2 $p \leftarrow$ `buscarEnLista`$(E[i], e)$
3 **return** `obtenerInfo`(p)

donde `obtenerInfo` retorna la información contenida en el objeto apuntado por el puntero p.

Puntajes.
1.0 por modificaciones a `agregarPasajero`.
1.0 por entregar la información pedida sin buscar en las listas directamente