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

Pauta I1

IIC2133 — Estructuras de Datos y Algoritmos
Segundo semestre 2017

1 Pregunta 1

En el problema de la programación de máquinas, tenemos n tareas que realizar, cada una con su hora de inicio, s_i, y su hora de finalización, f_i, y tenemos un número suficientemente grande de máquinas para realizar las tareas; y lo que queremos es poder realizar todas las tareas ocupando el menor número posible de máquinas, respetando la restricción de que una máquina solo puede realizar una tarea a la vez (por lo tanto, dos tareas se pueden realizar en una misma máquina solo si la hora de finalización de una es menor o igual que la hora de inicio de la otra). Escribe un algoritmo de **backtracking** para resolver este problema.

Evaluación

Cabe destacar que backtracking por si solo no garantiza que la solución encontrada sea mínima en la cantidad de máquinas, por lo que los que no consideraran eso recibían solo **la mitad** del puntaje.

Se pensaron 3 formas sencillas de asegurar optimalidad:

- Limitar la cantidad de máquinas. Si no se encontraba una asignación, entonces se aumentaba el límite.
- Revisar todas las posibles asignaciones, y recordar siempre cual era la mejor.
- Ordenar las tareas por hora de inicio: luego de eso, la primera asignación que se encuentre es óptima.

Por otro lado, si el algoritmo propuesto no es de Backtracking, también se asignaba solo **la mitad** del puntaje.

En definitiva, se asignaron **3pts** a que la solución fuera de Backtracking, y otros **3pts** a que garantizara optimalidad.

Solución (recordando la mejor)

Dado un conjunto de tareas $T = \{t_1, \cdots, t_n\}$, y un conjunto de máquinas $M = \{m_1, \cdots, m_n\}$, se busca la mejor asignación S de tareas a esas máquinas, la cual usa sólo las primeras k máquinas.

Notar que en este algoritmo, tanto S como k son por referencia.

BuscarAsignación(T, M, S, k)

```
if todas las tareas en T han sido asignadas then
    return True
end
t ← una tarea sin asignar en T
foreach máquina m ∈ M do
    if t no tiene tope de horario con ninguna de las tareas en m then
        Se le asigna t a m
        if BuscarAsignación(T, M, S, k) then
            m ← cantidad de máquinas que usa la asignación actual
            if m < k then
                S ← asignación actual
                k ← m
            end
        end
        Se des-asigna t de m
    end
end
return False
```

Para encontrar la asignación óptima, primero se setea S como la asignación trivial: cada tarea a una máquina distinta, por lo que $k = n$.

Luego se llama:

BuscarAsignación(T, M, S, k)

Al finalizar, S contendrá la asignación óptima de tareas, usando k máquinas.

2 Pregunta 2

La pregunta consta de tres partes importantes:

- Encontrar la mediana
- Particionar según la mediana
- Que todo esto sea $\mathcal{O}(n)$ en promedio, siendo n la cantidad de elementos del arreglo

Para encontrar la mediana en tiempo $\mathcal{O}(n)$, podemos usar partition de manera inteligente, utilizando el algortimo quickselect.

medianPartition(A, medianPos, p, r)

```
m ← partition(A, p, r)
if m < medianPos then
    medianPartition(A, medianPos, m+1, r)
end
else if m > medianPos then
    medianPartition(A, medianPos, p, m-1)
end
return medianPos
```

Partition es el algoritmo visto en clases, *medianPos* corresponde a la posición en que se debería encontrar la mediana.

Si se utiliza quickselect no es necesario particionar los elementos según la mediana, ya que el algoritmo ya lo hace. En caso contrario de que se use otro tipo de algoritmo, es necesario dejar todos los elementos menores a la mediana a un lado, y los mayores al otro.

QuickSelect funciona en tiempo $\mathcal{O}(n)$ en promedio, la demostración de esto es parecida a la de QuickSort, pero no era necesario hacerla.

3 Pregunta 3

- Counting sort crea un arreglo del tamaño del alfabeto (normalmente de tamaño 256) para ordenar. Considerando que muchas veces al ordenar se tienen pocos elementos (a veces 2 o 3 incluso), es excesivamente ineficiente hacer counting sort sobre estos pocos datos. En el caso de MSD string sort, puede darse el caso de que se separen los strings en muchos grupos pequeños, y haya que ordenarlos por separado. Usando counting sort se estarían haciendo al menos 256 operaciones por cada grupo pequeño. Si bien en orden de complejidad no parece ser muy terrible, en la práctica mejora mucho el tiempo de ejecución cambiar a un algoritmo como counting sort. Además, si se hace recursiva la ejecución de MSD string sort (lo cual es lo común), es probable que se produzca stack overflow.
- La gran gracia de MSD string sort es que no importa el largo de los strings ya que en la práctica al iterar sobre unos pocos caracteres de los strings, se logra ordenar el arreglo completo. El peor caso de MSD string sort se da cuando el algoritmo debe iterar sobre los strings completos, lo que hace que el tiempo de ordenación dependa directamente del largo de los strings.

4 Pregunta 4

a) Buscamos la clave en el nodo que estamos mirando; el primer nodo que miramos es obviamente la raíz del árbol. Si encontramos la clave en este nodo, entonces pasamos a b); de lo contrario, tenemos que descender a uno de los hijos del nodo y, recursivamente, buscar ahí. En este caso, usamos el hecho de que las claves del nodo están ordenadas para decidir a cuál hijo tenemos que descender: si la clave buscada es menor que la menor de las claves del nodo, entonces bajamos al hijo izquierdo del nodo; si la clave buscada es mayor que la mayor de las claves del nodo, entonces bajamos al hijo derecho del nodo; si la clave buscada está entre la menor y la mayor claves del nodo (en un nodo 3), entonces bajamos al hijo del medio.

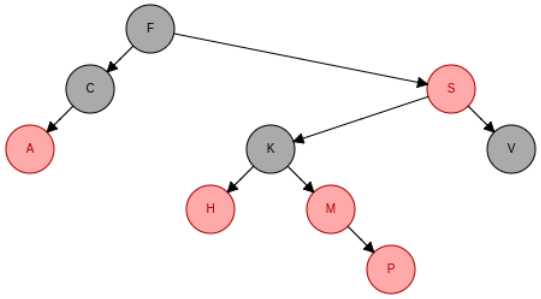
b) Como es un nodo interior, entonces la clave predecesora (o la sucesora, se puede elegir cualquiera) está siempre en una hoja, más abajo en el árbol: la clave predecesora/sucesora es la más grande/pequeña de las claves almacenadas en el subárbol izquierdo/derecho (respecto de la clave que estamos eliminando). Para encontrarla (supongamos que buscamos la predecesora), bajamos a la raíz del subárbol correspondiente; si es una hoja, entonces es la clave más a la derecha en la hoja; de lo contrario, bajamos por el puntero más a la derecha del nodo y buscamos recursivamente ahí. Esta clave predecesora la ponemos en el lugar de la clave que queremos eliminar, sacándola de la hoja en la que estaba.

c) Consideremos el agujero como un nodo ficticio, h, sin clave; h está inicialmente en una hoja del árbol. Como veremos, si h es hijo de un nodo-3, entonces podemos eliminarlo sin necesidad de hacerlo subir por el árbol. Esto se debe a que entre el padre y los hermanos de h hay suficientes claves para reestructurar estos nodos. Esto también es cierto si h es hijo de un nodo-2, y su hermano es un nodo-3. Sólo si h es hijo de un nodo-2 y su hermano es un nodo-2, debemos hacer subir h un nivel en el árbol y aplicar ahí de nuevo este algoritmo. Veamos.

- Si h es hijo izquierdo de un nodo-2 con clave X, y tiene como hermano un nodo-2 con clave Y ($X < Y$), entonces reestructuramos estos nodos de la siguiente manera: ponemos h en el lugar en que está X y hacemos que X y Y formen un nodo-3, hijo (único) de h. Con esto, h sube un nivel y hay que lidiar con él en este nuevo nivel.
- Si h es hijo izquierdo de un nodo-2 con clave X, y tiene como hermano un nodo-3, con claves Y y Z ($X < Y < Z$), entonces reestructuramos estos nodos así: Y sube y queda en el lugar de X, por lo que el nodo-3 se convierte en un nodo-2 con clave Z y X baja y queda en el lugar de h. En este caso, h es efectivamente eliminado (no sube).
- Ambos casos anteriores tienen casos simétricos, en que h es hijo derecho de un nodo-2.
- Si h es hijo izquierdo de un nodo-3 y su hermano inmediato —a su derecha o su izquierda— es un nodo 2 (el otro hermano puede ser un nodo 2 o un nodo 3), entonces cambiamos el padre por un nodo 2 (dejamos en el padre sólo una de las dos claves originales) y bajamos la otra clave, que pasa a formar un nodo 3 con el hermano inmediato de h. h es eliminado.
- Finalmente, si el hermano inmediato de h es un nodo 3, entonces separamos este hermano en dos nodos 2, de modo que uno de ellos reemplaza a h; en este proceso, hay que redistribuir las claves de los dos nodos 3. h es eliminado.

5 Pregunta 5

El árbol de la figura es un **árbol rojo negro** al cual se le acaba de insertar la clave P.



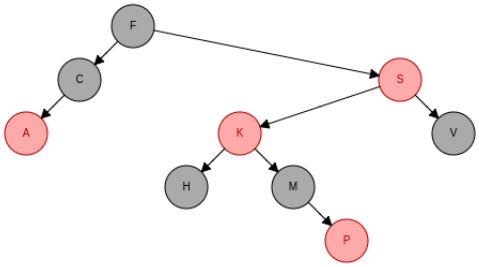
a) **[0.5 pts]** ¿De qué color está pintado el nodo con clave P? ¿Cómo sabemos que es el color?

[0.5 pts] Por decir que es rojo e indicar que es por las reglas de inserción (un nodo siempre se inserta pintado de rojo).
También se da puntaje si indican que, considerando que el árbol antes de la inserción de P estaba balanceado, la raíz siempre debe ser negra y como P tiene un color distinto a la raíz, entonces debe ser rojo.

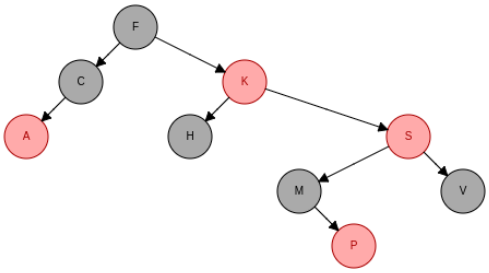
b) **[1 pts]** ¿Qué propiedad de árbol rojo-negro está siendo violada en estas condiciones?
[1 pts] Por indicar que es la propiedad que todo nodo de color rojo debe tener únicamente hijos de color rojo.l

c) **[1 pts]** ¿Cuáles dos tipos de operaciones se pueden ejecutar para restaurar la condición de árbol rojo-negro?
[1 pts] Por indicar que se puede hacer una recoloración y luego una rotación doble.

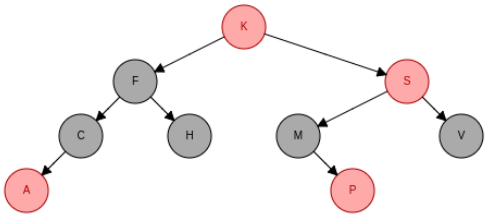
d) **[3.5 pts]** Ejecuta una a una las operaciones necesarias hasta restaurar la condición del árbol rojo-negro. A continuación se muestra el resultado de aplicar la operación indicada hasta llegar al árbol balanceado:



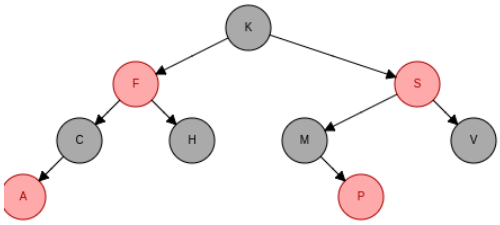
(a) Recoloración



(b) Rotación a la derecha



(c) Rotación a la Izquierda



(d) Recoloración

Figura 1: Aplicación de las operaciones necesarias para balancear el árbol

- **[0.75 pts]** Por indicar que se necesita una recoloración gracias al tío rojo de P.
- **[1 pts]** Por indicar que se necesita una rotación a la derecha ya que el tío de K es negro, con K rojo siendo el hijo derecho de un nodo rojo izquierdo.
- **[1 pts]** Por indicar que se necesita una rotación a la izquierda, ya que C es el tío negro de S y S es el hijo rojo derecho de K rojo.
- **[0.75 pts]** Por indicar que se necesita una recoloración por raíz roja.

Si hacen bien la primera recoloración, luego indican donde ocurre el nuevo conflicto y cómo solucionarlo, pero dibujan incorrectamente el árbol, entonces reciben **1.75 pts**.