Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2133— Estructuras de Datos y Algoritmos 2020-2

FECHA: 5 de Octubre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

Interrogación 1

Pregunta 1

a) Supongamos que al ejecutar Quicksort sobre un arreglo particular, la subrutina partition hace siempre el mayor número posible de intercambios; ¿cuánto tiempo toma Quicksort en este caso? ¿Qué fracción del mayor número posible de intercambios se harían en el mejor caso? Justifica.

Observación: según las issues, también se considera intercambio cuando se ejecuta sobre sí mismo.

Cuando partition ejecuta la mayor cantidad de intercambios posibles, es porque se eligió uno de los extremos del array como pivote. Ese caso corresponde al menos eficiente posible, y según lo visto en clases, Quicksort tendría complejidad temporal de $O(n^2)$.

En el peor caso, se realizarían $n + (n-1) + (n-2) + ... + 2 + 1$ intercambios, lo cual corresponde a $\frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$.

En el mejor caso, se realizarían $\frac{n}{2} + 2 \cdot \frac{n}{2} \cdot \frac{1}{2} + 4 \cdot \frac{n}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} + ...$ intercambios. La serie se repite $\log_2(n)$ veces, porque la array se divide continuamente en 2. De esta manera, se obtiene que el número total de intercambios corresponde a $\frac{n}{2}\log_2(n)$

Por lo tanto, la fracción equivaldría a $\frac{\frac{n}{2}\log_2(n)}{\frac{n(n+1)}{2}} = \frac{\log_2(n)}{(n+1)}$

El algoritmo *quicker-sort* llama a la subrutina *pq-partition*, que utiliza dos pivotes p y q (p < q) para particionar el arreglo en 5 partes: los elementos menores que p, el pivote p, los elementos entre p y q, el pivote q, y los elementos mayores que q. Escribe el pseudocódigo de pq-partition. ¿Es *quicker-sort* más eficiente que *quick-sort*? Justifica.

```
function pq_partition(A, inicio, final)
    // Asegurar que p < q
    if (A[inicio] > A[final]) then
        swap(A[inicio], A[final])
    end if
    let p = A[inicio]
    let q = A[final]

    let a = inicio
    let b = inicio
    let c = final - 1
    while a < c do
        a++
```

```
        if (A[a] < p) then
            b++
            swap(A[a], A[b])
        else if (A[a] > q) then
            while (A[c] > q and a < c) do
                c--
            end while
            swap(A[a], A[c])
            c--
            if (A[k] < p) then
                b++
                swap(A[a], A[b])
            end if
        end if

    end while

    // swap pivotes
    swap(A[b - 1], A[i])
    swap(A[c + 1], A[f])

    // retorna indices pivotes
    return (b - 1, g + 1)
end function
```

En primera instancia, las complejidades seguirían siendo las mismas, pero dado que la array se divide en 5 partes en lugar de 3, ahora el algoritmo de *quicker-sort* realiza 3 llamadas recursivas, en lugar de sólo 2. Puesto que $\log_3(n) < \log_2(n)$, el mejor caso y el caso promedio serán un poco más rápidos, pero la complejidad temporal, y la proporción a la cual escala el tiempo con respecto a la cantidad de elementos, seguirá siendo la misma: $n \log(n)$

Con respecto al peor caso, si bien para ámbos algoritmos es $O(n^2)$, en el caso de *quicker-sort*, es menos probable encontrar este caso, dado que elegimos dos pivotes en lugar de uno. De esta manera, para elegir el peor pivote en el *quicksort* normal, sólo basta con elegir uno de los extremos, mientras que para elegir el peor pivote en *quicker-sort*, se deben elegir los dos pivotes en el mismo extremo, o un pivote en cada extremo.

Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta"
– Matías Duhalde 12/11/19

FECHA: 5 de Octubre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

2. Con respecto a los Heaps binarios de n elementos

a) Escribe un algoritmo $O(n)$ para convertir un ABB en un min-Heap. Demuestra que es correcto.

Asumiendo que todas las claves son distintas entre sí, el siguiente algoritmo que implementa la función `ABBToMinHeap` es capaz de construir un heap en una array a partir de un ABB `T`.

```
function ABBToMinHeap(k, H, i)
    if (k.left != NULL) then
        ABBToMinHeap(k.left, H)
    end if

    H[*i] = k
    *i += 1

    if (k.right != NULL) then
        ABBToMinHeap(k.right, H)
    end if
end function

// Llamada inicial
let heap = n[] // array vacía de largo n
let counter = 0
ABBToMinHeap(T, heap, &counter)
```

`i` corresponde a la dirección de memoria de un contador, el cual indica el último índice ocupado de la array `H`. Es compartido por todos los llamados recursivos, y de esta manera se pueden realizar inserciones en la array en tiempo constante sin sobre-escrituras.

Complejidad:

El algoritmo comienza de la raíz del árbol `T`. Este realiza a lo más dos llamadas recursivas, una por cada nodo hijo que tenga. Debido a que cada nodo realizará recursivamente dos llamadas a `ABBToMinHeap` (una por hijo), en total, se realizarán n llamadas a dicha función, y la complejidad temporal del algoritmo será $O(n)$ en todos los casos.

Correctitud:

Cada nodo k ejecuta a lo más dos llamados recursivos a sus nodos hijos (izquierdo y derecho). Dado que el ABB también es finito, eventualmente se alcanzará una hoja y ese nodo no efectuará más llamados recursivos, terminando el algoritmo en una cantidad finita de pasos ($O(n)$, como se demostró anteriormente).

Un min-heap cumple la propiedad de que para todo nodo k en un árbol T, si k es padre de algún nodo hijo h, la clave de k es menor que la calve de h.

Caso base: El primer árbol insertado en la array representando al min-heap corresponde al menor valor del ABB T, respetando las propiedades establecidas anteriormente.

Hipótesis inductiva: Los primeros i números ingresados en el min-heap respetan las propiedades establecidas anteriormente.

Por demostrar... Si el nodo k_{i+1} posee un hijo izquierdo, significa que la clave de este hijo es **menor** a la clave de k_{i+1}, por lo tanto, ya se encuentra insertado en el heap (el cual es correcto según **HI**). Si k_{i+1} posee un hijo derecho, significa que la clave de este hijo es **mayor** a la clave de k_{i+1}, por lo tanto k_{i+1} debe ser insertado antes en el heap. Dado que la clave de k_{i+1} es mayor que todas las claves de los nodos k_j tal que $j < i + 1$, al ser insertado k_{i+1} en el heap, se cumplirán las propiedades del min-heap. ■

b) ¿Por qué encontrar el elemento de menor prioridad en un Heap requiere revisar solo $\frac{n}{2}$ elementos?

Para un heap, SIEMPRE hay garantía de que el elemento de menor prioridad se encuentra en las hojas de este. De esta manera, sólo bastaría revisar las hojas para poder encontrar el elemento de menor prioridad.

Sea H un heap de altura h (comenzando desde 1), si este estuviera completo, la cantidad de elementos que H posee, es $2^h - 1$.

Para todo heap H con altura $h > 1$, tenemos que su nivel $h - 1$ está completo, y posee una cantidad $2^{h-1} - 1$ de elementos hacia arriba. De esta manera, la cantidad de elementos que posee sólo el nivel h, corresponde a $n - (2^{h-1} - 1)$. Todos los elementos del nivel h corresponden a hojas, mientras que de los elementos del nivel $h - 1$ sólo algunos (o ninguno, en el caso que el nivel h esté completo, o sólo le falte 1 elemento) elementos corresponden a una hoja.

Dado que cada elemento del nivel h posee un elemento padre, y dos elementos comparten a un solo padre, la cantidad de elementos del nivel $h - 1$ que son padres corresponde a:

$$\lceil \frac{n - 2^{h-1} + 1}{2} \rceil$$

Para cualquier nivel completo h en el heap, la cantidad de elementos que posee corresponde a 2^{h-1}. Por lo tanto, para el nivel $h - 1$ posee un total de 2^{h-2} elementos, y por lo tanto, la cantidad de hojas del penúltimo nivel corresponde a:

$$2^{h-2} - \lceil \frac{n - 2^{h-1} + 1}{2} \rceil$$

Esto, sumado a la cantidad de hojas del último nivel, resulta en el siguiente total:

$$n - 2^{h-1} + 1 + 2^{h-2} - \lceil \frac{n - 2^{h-1} + 1}{2} \rceil$$

Si n es par ($n = 2k$), entonces

$$\begin{aligned}
&= 2k - 2^{h-1} + 1 + 2^{h-2} - \lceil k - 2^{h-2} + \frac{1}{2} \rceil \\
&= 2k - 2^{h-1} + 1 + 2^{h-2} - k + 2^{h-2} + \lceil \frac{1}{2} \rceil \\
&= k + 1 - 1 \\
&= \frac{n}{2}
\end{aligned}$$

Si n $(n = 2k + 1)$ es impar

$$
\begin{aligned}
&= 2k + 1 - 2^{h-1} + 1 + 2^{h-2} - \lceil \frac{2k+1}{2} - 2^{h-2} + \frac{1}{2} \rceil \\
&= 2k + 2 - 2^{h-1} + 2^{h-2} + 2^{h-2} - \lceil k + 1 \rceil \\
&= 2k + 2 - \lceil k + 1 \rceil \\
&= 2k + 1 - \lceil k \rceil \\
&= \lceil 2k + 1 - k \rceil \\
&= \lceil k + 1 \rceil \\
&= \lceil \frac{n-1}{2} + 1 \rceil \\
&= \lceil \frac{n}{2} + \frac{1}{2} \rceil \\
&= \lceil \frac{n}{2} \rceil
\end{aligned}
$$

Para un heap de altura $h = 1$, se tiene que este posee un único elemento $n = 1$, el cual se debe revisar trivialmente para poder obtener el elemento de menor prioridad, por lo tanto, también se cumple que la cantidad de elementos que se requiere revisar corresponde a $\lceil \frac{n}{2} \rceil = \lceil \frac{1}{2} \rceil = 1$.

Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta"
– Matías Duhalde 12/11/19

FECHA: 5 de Octubre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

3. Con respecto a los Árboles Binarios de Búsqueda

a) Sea T un ABB de altura h. Escribe un algoritmo que posicione un elemento arbitrario de T como raíz de T en $O(h)$ pasos.

Asumiendo que todas las claves son distintas entre sí, el siguiente algoritmo que implementa la función `setAsRoot` es capaz de posicionar el elemento `k` (el cual pertenece al ABB `T`) como raíz del ABB `T`.

```
function setAsRoot(k)
    while (k.parent != NULL) do
        let parentNode = k.parent

        k.parent = parentNode.parent
        if (parentNode.value < k.value) then
            parentNode.rightChild = k.leftChild
            k.leftChild = parentNode
        else
            parentNode.leftChild = k.rightChild
            k.rightChild = parentNode
        end if
    end while
end function
```

Correctitud:

En cada iteración se cambia el padre del nodo actual al "padre del padre" (notar la línea `k.parent = parentNode.parent`). Eventualmente, se alcanzará la raíz del nodo, la cual posee su atributo `parent` como `NULL`, y se colocará `k.parent = NULL`. Por lo tanto, el algoritmo es finito.

El algoritmo también entrega un resultado correcto, debido a que en cada rotación, se pueden producir dos casos:

- si la clave del padre es mayor a la de `k`, es decir, `k` es hijo izquierdo de su padre, el nodo padre se coloca como hijo derecho de `k`, y se reemplaza el hijo izquierdo del padre con el hijo derecho de `k`
- si la clave del padre es menor a la de `k`, es decir, `k` es hijo derecho de su padre, el nodo padre se coloca como hijo izquierdo de `k`, y se reemplaza el hijo derecho del padre con el hijo izquierdo de `k`

De esta manera, se mantiene la propiedad de orden de los ABB.

Complejidad:

La demostración es bastante directa; dado que todos los *statements* contenidos dentro del `while` tienen tiempo constante, sólo nos interesa determinar la cantidad de ciclos que este ciclo ejecuta en cada llamada.

Sea `h` la altura máxima del ABB `T`. Podemos notar que en cada iteración se ejecuta la instrucción `k.parent = parentNode.parent`, es decir, se cambia el padre del nodo actual por el "padre del padre", y se reduce en uno la altura del nodo actual. De esta manera, el algoritmo itera tantas veces como la altura inicial del nodo a colocar como raíz. Dado que el peor caso es aquel en que el nodo inicial `k` es hoja de la rama más alta (con altura `h`), la complejidad temporal del algoritmo en el peor caso es O(`h`).

b) Sean T_1 y T_2 dos ABB de n y m nodos respectivamente. Explica, de manera clara y precisa, cómo realizar un merge entre ambos árboles en $O(n+m)$ pasos para dejarlos como un solo ABB balanceado T con los nodos de ambos árboles.

Usando el mismo algoritmo para construir un min-heap a partir de un ABB usado en la pregunta 2, se puede construir una array que tenga los elementos del arbol ordenados. El algoritmo es el siguiente:

```
function ABBToSortedArray(k, A, i)
    if (k.left != NULL) then
        ABBToMinHeap(k.left, A)
    end if

    A[*i] = k
    *i += 1

    if (k.right != NULL) then
        ABBToMinHeap(k.right, A)
    end if
end function

// Llamada inicial
let array = n[] // array vacía de largo n
let counter = 0
ABBToSortedArray(T, array, &counter)
```

Se demostró que este algoritmo tenía complejidad $O(n)$, donde n es la cantidad de elementos del ABB.

Se puede usar este algoritmo para transformar T_1 y T_2 en arrays A_1 y A_2, y en tiempos $O(n)$ y $O(m)$ respectivamente:

$$A_1 = \{n_1, n_2, ..., n_n\}$$

$$A_2 = \{m_1, m_2, ..., m_m\}$$

Donde n_i, $0 <= i <= n$ son los elementos del árbol T_1, y n_j, $0 <= j <= m$ son los elementos del árbol T_2. Luego, se pueden mezclar estas arrays usando el siguiente algoritmo:

```
function MergeArrays(A, B)
    let k = |A| + |B|
    let R = (k)[] // Array de largo n + m
    let a_index = 0
    let b_index = 0
    for (i = 0; i < k; i++) do
        if A[a_index] < B[b_index] then
            R[i] = A[a_index]
            a_index++
        else
            R[i] = B[b_index]
            b_index++
        end if
    end for
    return R
end function
```

Este algoritmo crea una nueva array R de largo $n + m$. En cada iteración, se inserta en R el elemento menor entre A_1 y A_2. Todas las instrucciones tienen tiempo constante, y dado que se itera $n + m$ veces, el algoritmo tiene tiempo $O(n + m)$.

A continuación, para transformar esta array R a un árbol balanceado, se puede usar el siguiente algoritmo:

```
function CreateTree(A, k, i, f)
    if f - i < 1 then
        return NULL
    end if

    currentIndex = i + (f - i)/2
    k.value = A[currentIndex]
    k.leftChild = CreateTree(A, k.leftChild, i, currentIndex)
    k.rightChild = CreateTree(A, k.leftChild, currentIndex + 1, f)

    return k
end function

let raiz = node{} // k comienza como un nodo vacío
CreateTree(R, k, 0, |R|)
```

Este algoritmo iría tomando la mediana de la array en cada recursión, y haciendo llamados recursivos con los elementos a la izquierda de la mediana y a la derecha de la mediana. En total, se realizan realizan n + m llamados

Dado que la array está ordenada, se cumple la propiedad de los ABB, y dado que siempre se divide por la mediana, en cada nodo hay la misma cantidad de nodos en el arbol izquierdo y en el derecho (el derecho puede contener uno menos en casos donde $f - i$ es impar), por lo que se logra la condición de balance.

En resumen, tendríamos:

$$O(n) + O(m) + O(n + m) + O(n + m) = O(n + m)$$

Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta"
– Matías Duhalde 12/11/19

FECHA: 5 de Octubre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

Pregunta 4

Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta"
– Matías Duhalde 12/11/19