uc.png

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

IIC2133 — Estructuras de Datos y Algoritmos 1'2025

Interrogación 1

7 de mayo de 2025

Tiempo: 2 horas

Condiciones de entrega: Se deben entregar 3 de las 4 preguntas.

Forma de Entrega: La entrega es a través de Canvas. Al final de la prueba tienen 10 minutos para subir la imagen de la prueba. En formato vertical, cada pregunta por separado.

Dispositivo: Para digitalizar la prueba se permite el uso de UN dispositivo (teléfono, tablet) el cual debe estar guardado y en silencio. En el caso de ir al baño el dispositivo debe dejarse adelante en la mesa con los ayudantes.

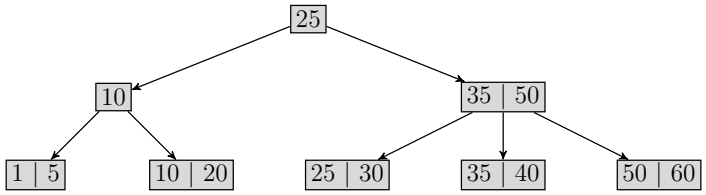
Torpedo: Se permite una hoja tamaño oficio escrita a mano.

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

Hint: Lea todas las preguntas antes de iniciar las respuestas.

1. Dividir para conquistar y Árboles

Considera la siguiente variante de un árbol 2-3 en donde todas las hojas son nodos de tipo 3-nodo, y las dos claves almacenadas en una hoja representan un intervalo. Por ejemplo, en la siguiente figura están representados los intervalos [1-5], [10-20],[25-30], [35-40], y [50-60].



El resto de los nodos (los que no son hojas) pueden ser de tipo 2-nodo o 3-nodo y contienen algunas de las claves iguales a las contenidas en las hojas. Es decir una clave que está en una hoja, puede estar también en un nodo no hoja. Por ejemplo, la clave 10 está contenida en el 2-nodo [10] y en su hijo [10-20] (que es una hoja), mientras que la clave 7, al no estar contenida en ninguna hoja, tampoco estará en un nodo no hoja.
Todos los intervalos almacenados en los nodos hoja son disjuntos, es decir $\forall X, Y \in \text{hojas}, X \cap Y = \emptyset$
La figura mostrada anteriormente ejemplifica la estructura posible para los intervalos ya indicados.

a) (2 pts) Usando la estrategia de dividir para conquistar, escriba en pseudocódigo la función $find(k) \rightarrow [k_i, k_f]$ que busque un valor k en el árbol y entregue el 3-nodo que lo contiene $i.e. k_i \leq k \leq k_f$ si existe y null si no. P. ej. $find(4) \rightarrow [1, 5]$ y $find(43) \rightarrow null$
Solución: A diferencia de un árbol 2-3 original donde la búsqueda se puede detener en un nodo no hoja cuando se encuentra la clave, en esta versión se debe continuar la búsqueda hasta la hoja

Algorithm 1: `find`(k, t)

```
1  begin
2      Condiciones de término
3      if t = hoja and k1 ≤ k ≤ k2 then
4          return [k1, k2]
5      else
6          return null
7      recorrido por el árbol
8      if t es nodo tipo 2 then
9          if k ≤ k1 then
10             find(k, t.left)
11             Siga buscando en rama izquierda
12         else
13             find(k, t.right)
14             Siga buscando en rama derecha
15     else
16         if k ≤ k1 then
17             find(k, t.left)
               // Siga buscando en rama izquierda
18         else
19             if k1 ≤ k ≤ k2 then
20                 find(k, t.center)
                   // Siga buscando en rama central
21             else
22                 find(k, t.right)
                   // Siga buscando en rama derecha
```

b) (2 pts) ¿Qué modificaciones introduciría en la estructura anterior para que dadas dos claves k, j, con $k < j$ permita entregar, en $O(n)$, la lista de intervalos que contienen desde la clave k (primer intervalo) hasta la clave j (último intervalo). P. ej. $k = 4$ y $j = 39$ entrega el resultado [1-5], [10-20],[25-30], [35-40]
Hint: Para modificar la estructura puede agregar punteros a los nodos, punteros entre nodos, o parámetros extra a los nodos.
Solución: Se unen las hojas como una lista ligada simple o doble. La búsqueda con find es $O(log(n)$ y el recorrido $O(n)$

c) (2 pts) Para la estructura modificada por usted en b) escriba en pseudocódigo la función $Range(k, j) \rightarrow [k_a, k_b], ..., [k_z, k_t]$ descrita anteriormente en b). Para ello hay 3 casos:

- k y j están contenidos en dos intervalos, p. ej. $Range(4, 39) \rightarrow$ [1-5],[10-20],[25-30], [35-40]
- k y j están en el mismo intervalo, p. ej. $Range(10, 19) \rightarrow$ [10-20]
- k o j, o ambos, no están en los intervalos, p. ej. $Range(1, 22) \rightarrow$ null

.
Solución:

Algorithm 2: Range(k, j)

```
1 if find(k) = null then
2     Return null No hay valor de inicio
3 else
4     if j ≤k_2 then
5         Return [k_1, k_2] El rango está en un solo nodo
6     else
7         r ← [k_a, k_b] primer intervalo;
8         Recorra la lista en (t hasta t=null y j es mayor que k_i agregue la hoja a r;
9         return r;
```

2. Uso de estructuras Árboles Binarios de Búsqueda, AVL, 2-3

a) (2pts) Considere un árbol AVL inicialmente vacío:

(I) Proponga una secuencia de inserciones de claves tal que requiera lo antes posible una rotación simple para mantener el balance del árbol.
Solución: Basado en la diapo. 44 de árboles AVL: 3, 2, 1; o también 1, 2, 3 –en suma, cualquier secuencia creciente o decreciente de tres claves.

(II) Proponga una secuencia de inserciones de claves tal que requiera lo antes posible una rotación doble para mantener el balance del árbol. Dibuja paso a paso el proceso de inserciones.
Solución: También basado en la diapo. 44: 4, 2, 3; o 4, 6, 5; es decir, cualquier secuencia de tres claves que produzca un quiebre en la dirección de la inserción

(III) Cuál es la menor secuencia de inserciones en un árbol 2-3 inicialmente vacío vacío que produzca un árbol 2-3 con 4 nodos tipo 3
Solución: 4 nodos tipo 3 son 8 claves: dos en la raíz y dos en cada uno de los 3 hijos.
P.ej., las primeras 3 claves pueden ser 10, 20 y 30; y luego 5 y 40 una raíz tipo 2 (20) y dos hijos tipo 3: (5/10) y (30/40). Luego insertamos 50, que va a parar inicialmente a (30/40, dejando momentáneamente (30/40/50), que hace que 40 suba a la raíz, convirtiéndola en un nodo tipo 3 (20/40), y que 30 y 50 queden en nodos tipo 2. Finalmente, insertamos 35 y 55 (o cualesquiera dos claves que vayan a parar a (30) y (50), respectivamente)

b) (2 pts) El Profesor Eterovic indica que ha descubierto una propiedad notable de los ABB. Suponga que la búsqueda de la clave k en un ABB termina en una hoja. Consideremos tres conjuntos: A, las claves a la izquierda de la ruta de búsqueda; B, las claves en la ruta de búsqueda; y C, las claves a la derecha de la ruta de búsqueda. El profesor Eterovic afirma que tres claves cualesquiera $a \in A, b \in B$ y $c \in C$ deben satisfacer $a \leq b \leq c$. ¿Será cierto? En caso afirmativo, demuestre formalmente esta propiedad; en caso negativo, muestre con un ejemplo que la propiedad no se cumple.
Solución: La propiedad no es cierta, por lo que basta mostrar un contraejemplo: considera el ABB con raíz 4 e hijos 3 y 6, en que 6 tiene a su vez hijos 5 y 7. Considera la búsqueda del 7: ruta 4, 6, 7 $\Rightarrow$ a la izquierda quedan las claves 3 y 5, en que $5 > 4$.

c) (2 pts) Escriba el algoritmo `CheckAVL`(T), tal que dado un árbol binario de búsqueda T de n nodos (representado por un puntero a su raíz), entregue como resultado el par (v, h) tal que:

- $v =$ `true` si el ABB cumple con las propiedades de un AVL, $v =$ `false` en otro caso.
- h es la altura del ABB dado.

El árbol T es un ABB, es decir, no contiene información de balance almacenada en los nodos.

Hint: Quizás le sea de utilidad aprovechar la definición recursiva de un ABB para pensar en un algoritmo recursivo.

Solución: el algoritmo es el siguiente

Algorithm 3: `CheckAVL`(T)

```
1  begin
2      if T = ∅ then
3          return (true, 0)
4      else
5          (v1, h1) ← CheckAVL(T.izq)
6          (v2, h2) ← CheckAVL(T.der)
7          h ← 1 + máx {h1, h2}
8          if v1 AND v2 AND |h1 − h2| ≤ 1 then
9              return (true, h)
10         else
11             return (false, h)
```

Tener en cuenta que debe calcular la altura del árbol e indicar si es AVL o no. En la evaluación, dar un peso de 50 % a cada cosa.

3. Colas con prioridades

En una farmacia de barrio donde llegan a lo más 99 clientes al día, existe un tótem que entrega 3 categorías de turnos, estos son A de atención normal, D de delivery y T de tercera edad. Cada cliente, cuando llega, saca ticket (turno) que correspondiente a su categoría con un número que va aumentando en cada categoría (Por ejemplo, A1, A2, T1, D1, A3, etc.). Existen 2 personas atendiendo (caja) y cada vez que terminan de atender, llaman al siguiente turno (Next) que se selecciona según su prioridad, donde T es más prioritario que D y el menor es A. Indique:

a) (2 pts) Qué estructura de datos eficiente usaría para esta situación, indicando la forma en que se ordenan los tickets y las restricciones que aplican.
Solución: Se implementa UN max heap. Para determinar la prioridad se define las constantes T=1000, D=100 la prioridad del ticket queda como pT=T+t, pD=D+d y pA=a, con t,d,a los números de atención siguientes en cada categoría. En caso de otra estructura se debe justificar la eficiencia, para otra forma de manejo de prioridades se debe evaluar eficiencia.

b) (2 pts) Escriba en Pseudocódigo la función Ticket → turno. Fíjese en que se debe seleccionar la categoría.
Solución: Se asume la estructura de UN max heap y la prioridad
a=0; d=0; t=0;

Algorithm 4: `Ticket(c)`

```
1  begin
2      if c=T then
3          t=t+1 Se entrega el próximo turno de la categoría
4          pt=1000+t Se calcula la prioridad
5          InsertHeap(pt) Se inserta en el heap
6          Return ct Se entrega el turno (p. ej. T1)
7      else
8          if c=D then
9              d=d+1
10             pd=100+d
11             Insertheap(pd)
12             Return cd
13         else
14             a=a+1
15             Insertheap(a)
16             Return ca
```

c) (2 pts) Escriba en pseudocódigo la función Next(caja) → turno donde caja es 1 o 2 que indica el cajero que lo atiende y entrega el número de turno para atención. El caso de borde es que alguna de las categorías no tenga turnos en espera.
Solución: Si se ocupa un solo max heap el algoritmo es básicamente el visto en clases.

Algorithm 5: `Next` (sin parámetros)

```
1  begin
2      t ← ExtractHeap(H) El heap global se llama H if t = null then
3          return null Si el heap está vacío retorna nulo (opcional)
4      else
5          if t > 1000 then
6              return T || (t/1000)
7          else
8              if 100 < t < 1000 then
9                  return D || (t/100)
10             else
11                 return A || (t)
```

4. Ordenamiento

Hasta que Murphy interfirió, estábamos disfrutando de un buen día custodiando un arreglo ordenado $A[1..n]$ de números enteros dentro del rango $[1..n^2]$. Sin embargo, en un momento de descuido, tomó nuestro arreglo A y lo cortó en bloques de b elementos usando una tijera, como si fuera una cinta (donde n es divisible por b). Después, mezcló estos bloques en una bolsa, los sacó uno por uno sin mirar y los unió en ese orden, creando un nuevo arreglo de n elementos que ya no está necesariamente ordenado.
Por ejemplo, para el arreglo

$$A[1..12] = \langle 3, 5, 7, 8, 9, 10, 12, 15, 17, 20, 21, 23 \rangle$$

y $b = 3$, un posible resultado de la acción de Murphy podría ser:

$$\langle 12, 15, 17, 3, 5, 7, 20, 21, 23, 8, 9, 10 \rangle.$$

Ya que nuestra tarea era mantener el arreglo ordenado, debemos restablecer el orden rápidamente para que nuestra jefa no lo note, además usando la menor cantidad de espacio extra posible para no ser detectados. Considere los siguientes casos. Al medir el tiempo de ejecución del algoritmo, considere todas las operaciones realizadas.

a) (2 pts) Si $b \in O(1)$, ¿qué algoritmo de ordenación usaría para recuperar el orden de A de manera que tenga el mínimo tiempo de ejecución posible, mientras usa la mínima cantidad de memoria adicional posible? Indique el tiempo de ejecución y memoria adicional usada por su algoritmo.

Solución: Dado que el arreglo está parcialmente desordenado (porque cada bloque de b elementos se mantiene ordenado), se podría aprovechar para reducir el tiempo necesario para ordenar, comparando sólo los primeros (o últimos) elementos de cada bloque. Sin embargo, para $b = 1$ lo mejor que pueden hacer es algo que ordene en tiempo $O(n \lg n)$ **[50 % del puntaje]** y usando espacio $O(1)$ (es decir, in-place) **[50 % del puntaje]**. Pueden plantear usar `HeapSort` o definir uno propio (mientras que respeten las complejidades de tiempo y espacio mencionadas). Quizás también podrían plantear usar `RadixSort`, pero deberían argumentar que la cantidad de dígitos de cada número del arreglo es $O(\log n)$. Estaría mal usar `CountingSort` en este caso, ya que el rango de los números hace que ese algoritmo tenga tiempo de ejecución (y espacio adicional) $O(n^2)$.

b) (2 pts) Si $b = \sqrt{n}$, diseñe un algoritmo de ordenación para A cuyo tiempo de ejecución sea estrictamente menor que el de los algoritmos estudiados en clases. Si lo necesita, puede usar algoritmos estudiados en clases como subrutina, indicando claramente cuáles son (y demás detalles que considere necesarios para entender su algoritmo). Su algoritmo puede usar espacio adicional $O(n)$.

Solución: Aquí se puede lograr tiempo $O(n)$ definiendo un algoritmo que coloque los primeros (o últimos) elementos de cada bloque en un arreglo separado, ordene esos elementos, y luego copie los bloques de acuerdo a ese orden en una secuencia adicional (que necesita espacio $O(n)$). Es importante que los primeros elementos de cada bloque se ordenen en un arreglo separado, para evitar el tiempo original $O(n \log n)$. Un ejemplo de algoritmo es el siguiente:

Algorithm 6: `Ordenar`$(A[1..n])$

```
1 begin
2     Copiar el primer elemento de cada bloque en un arreglo B[1..√n], en donde cada elemento
        almacena un elemento de A y el bloque al que corresponde
3     HeapSort(B)
4     for i = 1, ..., n do
5         Copiar el bloque indicado por el elemento B[i] al final de la secuencia C[1..n]
6     Copiar C en A
```

Dado que `HeapSort` se realiza sobre un arreglo de $\sqrt{n}$ elementos, el tiempo de ejecución es $O(\sqrt{n}\lg(\sqrt{n}))$, lo cual es $O(n)$. el resto de los pasos también toman tiempo $O(n)$, por lo que ese es el tiempo total. En lugar de `HeapSort`, podrían haber usado otro algoritmo y estar bien (por ejemplo, `QuickSort`, `MergeSort`, `SelectionSort`, `InsertionSort`, o `RadixSort`). Nuevamente, no es correcto usar `CountingSort`, por la misma razón de la parte (a).
Nota Importante: Aquí pueden argumentar también que la solución a la parte (c) sirve también para esta parte, lo cual está bien (incluso, por ejemplo, si la parte (c) usa espacio mayor a $O(\sqrt{n})$, lo cual le daría puntaje parcial en aquella pregunta, mientras sea $O(n)$ va a estar bien para esta parte).

c) (2 pts) Si $b = \sqrt{n}$, diseñe un algoritmo de ordenación para A cuyo tiempo de ejecución sea estrictamente menor que el de los algoritmos estudiados en clases, y cuyo espacio adicional sea $O(\sqrt{n})$. Si lo necesita, puede usar algoritmos estudiados en clases como subrutina, indicando claramente cuáles son y demás detalles que considere necesarios para entender su algoritmo.

Solución: A diferencia de la parte anterior, sólo podemos usar espacio $O(\sqrt{n})$. La siguiente solución usa incluso menos espacio adicional $O(1)$. La idea es usar algo similar a `SelectionSort` con el primer elemento de cada bloque.

Algorithm 7: `SelectionSortBloques`$(A[1..n])$

```
1 begin
2     for i = 1..√n do
3         min ← i  // Posición del mínimo
4         for j = i + 1..√n do
5             if A[(min − 1)√n + 1] > A[(j − 1)√n + 1] then
6                 min ← j
7         Intercambiar los elementos de los bloques i y min, elemento a elemento (O(1) espacio
           adicional)
```

Dado que es `SelectionSort`, el tiempo de ejecución es $O((\sqrt{n})^2) = O(n)$. El espacio adicional es $O(1)$. Cualquier alternativa que tome tiempo $O(n)$ y use espacio $O(\sqrt{n})$ y sea correcta es válida.
Nota importante: también es posible haber escrito esta solución en la parte (b) e indicar en la parte (c) que es la misma. Si la respuesta es correcta y lo indica, se tiene puntaje completo en ambas partes.