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

IIC2133 — Estructuras de Datos y Algoritmos 2'2023

Interrogación 1

29 de septiembre 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. Análisis de algoritmos

Strand Sort es un algoritmo de ordenación que utiliza una secuencia auxiliar para ordenar una secuencia A.

```
   StrandSort(A):
1      B ← Secuencia vacía
2      C ← Secuencia vacía
3      while A tiene elementos :
4          x ← primer elemento de A
5          Borrar x de A e insertarlo al final de B
6          foreach a elemento de A do
7              if a > último elemento de B :
8                  Borrar a de A e insertarlo al final de B
9          C ← Merge(B, C)
10         B ← Secuencia vacía
11     return C
```

(a) [1 pto.] Demuestre que `StrandSort` termina.

Solución.
Observemos que las líneas 4 y 5 seleccionan y extraen un elemento de A en cada iteración del **while**. Además, el **foreach** de la línea 6 itera sobre los elementos restantes de A. Como A es finita, el bloque **while** se itera a lo más $|A|$ veces. Como las operaciones de inserción, extracción y `Merge` son finitas, el algoritmo termina.

Puntajes.
0.5 por argumentar que los loops terminan.
0.5 por indicar que los pasos dentro de los loops son finitos.

(b) [2 ptos.] Como `Merge` es correcto, podemos demostrar que `StrandSort` ordena sin usar inducción. Demuestre que el algoritmo es correcto. *Pista:* ¿Qué cumple B?

Solución.
Dado que `Merge` es correcto, sabemos que cuando recibe secuencias ordenadas, produce una nueva secuencia ordenada. Consideremos primero, que todo elemento de A es extraido en alguna iteración del **while** y es agregado a B solo si es mayor que todos los elementos en dicha secuencia. Con esto, B

está ordenada.

Además, al sobreescribir C con el resultado del `Merge`, nos aseguramos de que cada elemento de A es insertado de forma ordenada en C al término de la iteración en al cual fue extraido. Como C contiene los mismos elementos de A al terminar el algoritmo, y dado que `Merge` ordena, sabemos que C está ordenada y concluimos que `StrandSort` ordena.

Puntajes.
1.0 por argumentar que B se construye ordenada en cada iteración.
0.5 por mencionar que C está ordenada al término de cada iteración.
0.5 por argumentar que cada elemento de A está en C al terminar el algoritmo.

(c) [1.5 ptos.] Determine si `StrandSort` tiene un mejor y/o peor caso. Justifique.

Solución.
El mejor caso del algoritmo corresponde a una secuencia A ordenada. En tal caso, solo ocurre una iteración del **while** pues el bloque **foreach** extrae todos los elementos de A consecutivamente. El peor caso corresponde a una secuencia en orden decreciente, en cuyo caso, hay n iteraciones del **while** cuando A tiene n elementos.

Puntajes.
0.5 por identificar mejor caso.
0.5 por identificar peor caso.
0.5 por justificar.
Observación: se pueden describir distintos peores casos siempre que la justificación sea consistente.

(d) [1.5 ptos.] Determine la complejidad de tiempo y memoria de los casos identificados en (c).

Solución.
Si consideramos la implementación de `Merge` que usa memoria lineal, el algoritmo toma tiempo $\mathcal{O}(n)$ y memoria $O(n)$ en el mejor caso, pues se ejecuta una iteración del **while** y los n pasos del bloque **foreach**. Por su parte, en el peor caso el tiempo es $\mathcal{O}(n^2)$ y memoria $\mathcal{O}(n)$, pues ocurren n iteraciones del **while** y los **foreach** reducen su largo en 1 (similar a algoritmos estudiados en clase).

Puntajes.
0.5 por indicar complejidades de mejor caso.
0.5 por indicar complejidades de peor caso.
0.5 por justificar.

2. Inserción en árboles

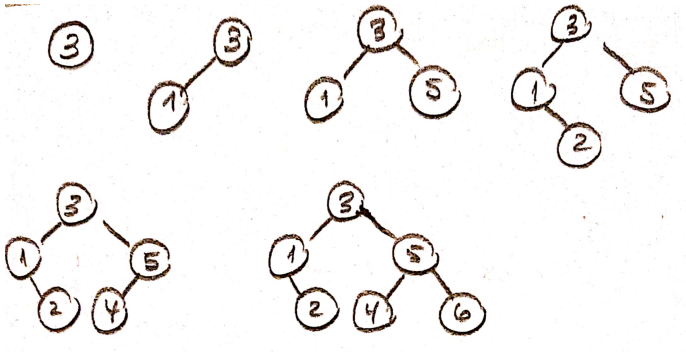
(a) [2 ptos.] Considere un árbol AVL $\mathcal{T}_1$ vacío. Para cualquier natural $k \geq 3$, proponga un orden para los naturales $1, 2, \ldots, k$ de forma que al insertarlos en $\mathcal{T}_1$ según su orden propuesto no sea necesaria ninguna rotación para reestablecer el balance AVL. Para ejemplificar, aplique su estrategia a los números $1, 2, 3, 4, 5, 6$ mostrando la secuencia de inserciones.

Solución.

Proponemos como posible solución la siguiente estrategia, indicando en paréntesis qué elementos se corresponden para la lista 1,2,3,4,5,6:

1. tomar la "mediana" de la lista de números (3)
2. tomar las "medianas" de los tramos de números restantes (1,5), en cualquier orden entre ellos
3. tomar las "medianas" de los tramos restantes (2,4,6), en cualquier orden entre ellos
4. seguir hasta llegar a tramos vacíos

Podemos notar que esta estrategia busca poblar el AVL por "nivel": primero el nodo del nivel 0 (la raíz), luego los nodos del nivel 1 (1,5), luego los del nivel 2 (2,4,6). Cuando ya no quedan números, el último nivel m puede quedar con menos de 2^m elementos. A continuación mostramos las inserciones en un AVL con el orden 3,1,5,2,4,6.

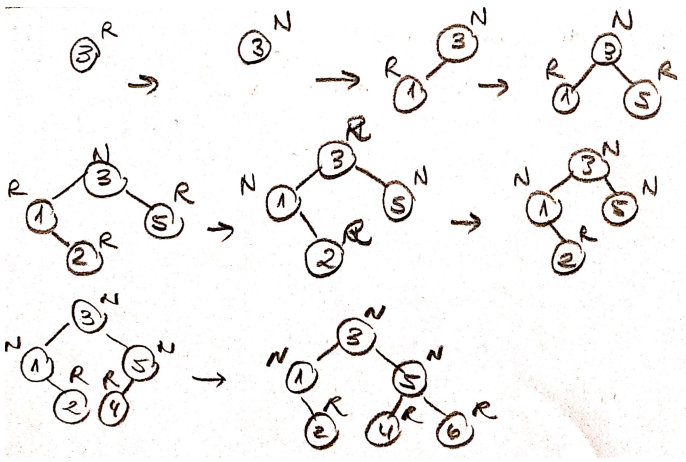


Puntajes.
1.0 por descripción de la estrategia.
1.0 por diagrama de inserciones correcto dado el orden propuesto.
Observación: pueden proponerse otros órdenes. Lo importante es que al insertar no hayan rotaciones.

(b) [2 ptos.] Considere un árbol rojo-negro $\mathcal{T}_2$ vacío. ¿El orden propuesto en (a) permite insertar en un $\mathcal{T}_2$ sin rotaciones ni cambios de color? ¿Permite insertar con cambios de color y sin rotaciones? Ejemplifique con las llaves $1, 2, 3, 4, 5, 6$.

Solución.

El orden propuesto exige realizar cambios de colores pero no rotaciones. Ejemplificamos con la misma secuencia 3,1,5,2,4,6.



Puntajes.
0.5 por concluir.
1.5 por diagrama correcto que verifica necesidad de cambios de color sin rotaciones para la secuencia dada.

(c) [2 ptos.] Generalice su estrategia de (a) al caso en que el conjunto de llaves no son números consecutivos. Para esto, proponga el pseudocódigo de un algoritmo que dado un arreglo $A[0 \dots n-1]$ de n llaves distintas ordenadas, retorne una lista B con las llaves en orden de inserción para garantizar que no ocurren rotaciones en un árbol AVL inicialmente vacío.

Solución.

Definimos el método `GetRanges` para obtener los rangos a analizar cuando se extrae la mediana del rango i, f. Su retorno es una lista con los rangos descritos como pares de índices.

```
GetRanges(A, B, i, f):
    if i ≥ f :
        Insertar A[i] al final de B
        return Lista vacía
    else:
        m ← ⌊(f − i)/2⌋
        Insertar A[m] al final de B
        L ← lista vacía
        Insertar tupla (i, m + i − 1) en L
        Insertar tupla (m + i + 1, f) en L
        return L
```

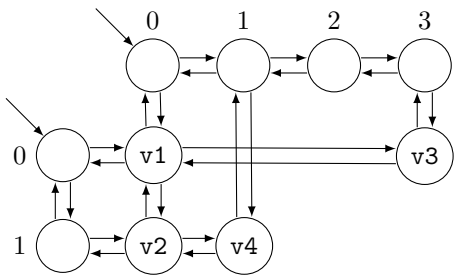
Con este método, se define `GetOrder` para obtener el orden de los elementos en la lista B. La lista auxiliar L almacena los intervalos que se deben estudiar en cada etapa.

```
GetOrder(A, n):
    B ← lista vacía
    L ← lista vacía
    Insertar tupla (0, n − 1) en L
    while L ≠ ∅ :
        (i, f) ← extraer primera tupla de L
        L′ ← GetRanges(A, B, i, f)
        Insertar elementos de L′ al final de L
    return B
```

Puntajes.
1.0 por método para obtener orden en el cual obtener medianas.
1.0 por agregar las medianas en el orden adecuado a la lista final.
Observación: debido a un typo, se omitió el análisis de complejidad y se consideró correcto cualquier algoritmo que entregue un orden adecuado de los nodos (hay varios órdenes posibles).

3. Diseño de algoritmos

Se necesita la creación de una estructura de datos para una planilla de cálculo con un número indefinido de filas y columnas. Para esto, se implementa la estructura Matyriz que consiste en una matriz poco densa que almacena celdas como nodos conectados por filas y columnas. Cada nodo `n` posee sus coordenadas `n.fila` y `n.columna`, así como `n.valor` y el tipo de valor en `n.tipo`. Cada nodo tiene cuatro punteros a las celdas siguientes en cada dirección (`null` en caso que no exista tal celda). En el ejemplo a continuación, existen 4 celdas con valores `v1,v2,v3,v4`. Además, la matriz posee celdas vacías para indicar los bordes izquierdo y superior de la planilla. Los únicos punteros de los que se dispone a nivel de código son `M.filas` y `M.columnas`, representados por las flechas diagonales que apuntan a los primeros elementos de los bordes.



(a) [1 pto.] Proponga el pseudocódigo para `Search(M,c,f)` que retorna el valor almacenado en la celda de fila `f` y columna `c`, o `null` en caso que no exista.

Solución.

Se asume que cada nodo tiene punteros `up`, `down`, `left` y `right` para acceder a sus posibles vecinos. También se asume que los nodos bordes de filas tienen índice -1 como columna. La idea análoga se aplica a los bordes de columnas.

input : Matriz M, índice de columna y fila c y f
output: Nodo de la matriz o `null`

`Search`(M, c, f)**:**

```
1   curr ← M.filas
2   while curr.fila < f :
3       if curr.down ≠ null y curr.down.fila ≤ f :
4           curr ← curr.down
5       else:
6           return null
7   while curr.columna < c :
8       if curr.right ≠ null y curr.right.columna ≤ c :
9           curr ← curr.right
10      else:
11          return null
12  return curr
```

Puntajes.
0.5 por recorrer y acceder correctamente a la fila.
0.5 por recorrer avanzando por columna.
Observación: otras formas de recorrido también son válidas.

(b) [1 pto.] Proponga el pseudocódigo para `Insert(M,c,f,v)` que inserta en la matriz `M` el valor `v` en fila `f` y columna `c`.

Solución.
Definimos primero un método auxiliar para identificar el nodo borde para la fila de interés. En caso que no haya suficientes nodos en el borde, los agrega. Llamamos `BorderRow` a este método.

input : Matriz M, índice de fila f
output: Nodo borde de la fila

`BorderRow`(M, f)**:**

```
1   b ← Search(M, −1, f)
2   if b == null :
3       b ← M.filas
4       while b.fila < f :
5           if b.down == null :
6               b.down ← Nodo con columna −1, fila b.fila + 1 y sin valor
7               b.down.up ← b
8           b ← b.down
9   return b
```

Análogamente se define `BorderColumn`. Con estos métodos, el algoritmo para insertar puede ser el siguiente. Observemos que dado que las filas y columnas son listas ligadas, esto se aprovecha para simplificar el pseudocódigo al iterar sobre sus elementos. Se asume que para las filas se itera de izquierda a derecha (puntero `right`) y para las columnas, de arriba hacia abajo (puntero `down`).

input : Matriz M, índices de columna y fila c, f y valor v

`Insert`(M, c, f, v)**:**

```
1   ℓ ← BorderRow(M, f)
2   u ← BorderColumn(M, c)
3   r ← null
4   d ← null
5   for e elemento de la lista en la fila ℓ :
6       if e.columna == c :
7           e.valor ← v
8           return
9       if e.columna < c AND (e.right == null OR e.right.columna > c) :
10          ℓ ← e
11          r ← e.right
12          break
13  for e elemento de la lista en la columna u :
14      if e.fila < f AND (e.down == null OR e.down.fila > f) :
15          u ← e
16          d ← e.down
17          break
18  n ← Nodo con valor v, fila f y columna c
19  Actualizar punteros de n y de los nodos ℓ, u, r, d
```

Puntajes.
0.4 por encontrar nodos vecinos en la misma fila.
0.4 por encontrar nodos vecinos en la misma columna.
0.2 por actualizar punteros.
Observación: si se asume que hay filas y columnas suficientes, se reemplazan las líneas 1 y 2 de `Insert` por llamados a `Search`. Lo importante en este inciso es la identificación de los vecinos que deben modificarse.

(c) [3 ptos.] Proponga el pseudocódigo de `SortRows(M,c)` que ordena las filas de la planilla según el valor de la columna `c`. Las filas sin valor en dicha columna deben quedar al final.

Solución.

Definimos un método para intercambiar filas consecutivas, de manera que podamos llamarlo dentro de `InsertionSort`. Luego de la ejecución de `ExchangeRows`, las filas indicadas invierten su posición. Observemos que solo es necesario reasignar punteros cuando ambas filas comparten elementos en una misma columna (bloque **if** de la línea 6).

input : Matriz M, índices de filas consecutivas r_1, r_2

```
ExchangeRows(M, r1, r2):
    n1 ← Search(M, −1, r1)
    n2 ← Search(M, −1, r2)
    while n1 ≠ null AND n2 ≠ null :
        if n1.columna == n2.columna :
            if n1.up ≠ null :
                n2.up ← n1.up
                n1.up.down ← n2
            if n2.down ≠ null :
                n1.down ← n2.down
                n2.down.up ← n1
            n2.down ← n1
            n1.up ← n2
            n1 ← n1.right
            n2 ← n2.right
            n1.fila ← r2
            n2.fila ← r1
        elif n1.columna > n2.columna :
            n2.fila ← r1
            n2 ← n2.right
        elif n1.columna < n2.columna :
            n1.fila ← r2
            n1 ← n1.right
```

Con ayuda de este método, un algoritmo que ordena por columna puede ser una versión de `InsertionSort` en la cual:

(i.) En lugar de comparar valores en un arreglo, al comparar las filas f_1 y f_2, se comparan los valores `Search`(M, c, f_1) y `Search`(M, c, f_2) al ordenar por fila c. Se asume que `null` es interpretado como un valor máximo.

(ii.) Si se debe intercambiar la fila j con la $j-1$, se llama a `ExchangeRows`$(M, j, j-1)$.

Puntajes.
1.5 por método de intercambio de filas o equivalente.
1.5 por método para ordenar la matriz completa.
`Observación:` otros métodos son correctos mientras cumplan el objetivo de ordenar las filas según el valor de una columna específica. Se pueden considerar supuestos.

(d) [1 pto.] ¿Cuál es el peor caso para el algoritmo de (c)? Indique su complejidad cuando existen n celdas en la matriz.

Solución.
El algoritmo dado hereda el comportamiento de `InsertionSort`: dado que la complejidad de peor caso corresponde al caso en que las filas están ordenadas de forma decreciente, el peor caso puede visualizarse en el caso en que hay una celda por fila (n filas). Luego, como cada operación de búsqueda toma tiempo $\mathcal{O}(n)$, el intercambio de dos filas es $\mathcal{O}(n)$. Con esto, el algoritmo resultante es $cO(n^3)$.

Puntajes.
0.5 por mencionar peor caso.
0.5 por deducir complejidad.

4. Aplicaciones de árboles

Una empresa de retail cuenta con un sistema de gestión de clientes (CRM) que registra la información de los clientes utilizando un árbol rojo-negro `C`, en que la llave es el RUN del cliente (tipo `int`) y en cada nodo `x` se

almacena la información del cliente de la forma `x.run`, `x.nombre`, `x.edad`, `x.renta`, `x.color`, `x.p` (puntero al padre del nodo x), etc. Solo se consideran las operaciones de búsqueda e inserción de clientes.

La empresa requiere generar de manera eficiente listas de clientes para realizar campañas de marketing. Por ejemplo, obtener desde `C`: "*la lista de run de todos los clientes mayores de 25 años y menores de 35, ordenados por edad*" (`select run from C where (25 < edad < 35) order by edad;`).

El arquitecto de sistemas define que se construyan índices para `C` utilizando tres nuevos árboles rojo-negro con llaves adecuadas para las búsquedas indicadas: `E` con llave `edad.run`, `R` con llave `renta.run`, `N` con llave `nombre.run`. Los nodos `i` de estos árboles índice tienen un puntero `data` al nodo de `C` correspondiente al run (`i.data->x`(con llave run)). Proponga para los casos siguientes un algoritmo que resuelva lo solicitado, asuma que los árboles `C, E, R, N` existen y están poblados correctamente.

(a) [1 pto.] Escriba en pseudo código el algoritmo `buscarCliente(C,E,R,N,x,llave)` que busca al cliente `x` por la llave indicada (run, edad, renta, nombre) en el árbol/índice adecuado retornando el nodo encontrado o null si no está en el árbol `C`.

Solución.

```
Buscar(n, x, llave):
    if n == null :
        return n
    if llave == RUN :
        k ← x.run
    else:
        k ← x.llave + '.' + x.run
    if n.llave == k :
        return n
    if n.llave > k :
        return Buscar(n.left, x, llave)
    else:
        return Buscar(n.right, x, llave)

BuscarCliente(C, E, R, N, x, llave):
    if llave == RUN :
        n ← C
    elif llave == edad :
        n ← E
    elif llave == renta :
        n ← R
    elif llave == nombre :
        n ← N
    m ← Buscar(n, x, llave)
    return m
```

Puntajes.
0.5 por buscar en el árbol adecuado.
0.5 por buscar recursivamente en árbol binario de búsqueda.

(b) [2 ptos.] Escriba en pseudo código el algoritmo `insertCliente(C,E,R,N,x)` que inserta los datos del cliente `x` (run, edad, renta, nombre) en `C` y actualiza los índices `E,R,N`. Asuma que la función `FixBalance(A,z)` está disponible y la puede utilizar en el árbol `A` adecuado al insertar un nodo `z`.

Solución.

```
InsertCliente(C, E, R, N, x):
    for llave ∈ {run, edad, renta, nombre} :
        ref ← null
        n ← BuscarCliente(C, E, R, N, x, llave)
        if n == null :
            if llave == run :
                ref ← Nuevo nodo con los datos de x
                Insert(C, ref)
                FixBalance(C, ref)
            if llave == run :
                if llave == edad :
                    b ← E
                elif llave == renta :
                    b ← R
                elif llave == nombre :
                    b ← N
                nodo ← Nuevo nodo
                nodo.llave ← ref.llave  + '.' + ref.run
                nodo.data ← ref
                Insert(b, nodo)
                FixBalance(b, nodo)
```

Puntajes.
0.5 por insertar en el árbol base C.
1.5 por insertar en los tres árboles paralelos con la llave modificada.
Observación: pueden insertar en los rojo-negro con un método que directamente arregle el balance, pero se debe mencionar que dicho método rebalancea.

(c) [3 ptos.] Escriba en pseudo código el algoritmo `rangoClientes(C,E,R,N,llave,min,max)` que retorna una lista de punteros a nodos de `C` que cumplen que el valor de la llave se encuentra entre los parámetros `min y max`, y la lista está ordenada de manera ascendente en el valor de la llave.

Solución.

```
RangoClientes(C, E, R, N, llave, min, max):
    if llave == RUN :
        n ← C
    elif llave == edad :
        n ← E
    elif llave == renta :
        n ← R
    elif llave == nombre :
        n ← N
    L ← lista vacía
    ListarNodosOrdenados(n, llave, min, max, L)
    return L
```

```
ListarNodosOrdenados(n, llave, min, max, L):
1    if n ≠ null :
2        if n.llave ≥ min AND n.llave ≤ max :
3            ListarNodosOrdenados(n.left, llave, min, max, L)
4            Agregar n al final de la lista L
5            ListarNodosOrdenados(n.right, llave, min, max, L)
6        elif n.llave < min :
7            ListarNodosOrdenados(n.right, llave, min, max, L)
8        elif n.llave > max :
9            ListarNodosOrdenados(n.left, llave, min, max, L)
```

Puntajes.
0.5 por escoger árbol a recorrer.
1.0 por realizar un recorrido in-order de los elementos del árbol (menores,mayores).
1.5 por restringir el rango e incluir todos los nodos adecuados en el reesultado