Repaso I1

Paula Grune, Alexander Infanta, Bernardita Nazar

Algoritmos de Ordenación

Considera el siguiente algoritmo de ordenación, para ordenar el arreglo A de largo n.

Demuestra que este algoritmo es correcto según los dos criterios vistos en clases

$$
\begin{array}{l}
\textbf{sort}(A, n): \\
\qquad \textbf{for}\ \ g \in \{5,\ 3,\ 1\}: \\
\qquad\qquad \textbf{for}\ i \in [g, n[: \\
\qquad\qquad\qquad j \leftarrow i \\
\qquad\qquad\qquad \textbf{while}\ (j \geq g\ \wedge\ A[j] < A[j-g]): \\
\qquad\qquad\qquad\qquad A[j] \rightleftarrows A[j-g] \\
\qquad\qquad\qquad\qquad j \leftarrow j - g
\end{array}
$$

Algoritmos de Ordenación

Primer criterio: ¿El algoritmo es **finito**?

$$
\begin{aligned}
&\boldsymbol{sort}(A, n):\\
&\quad \boldsymbol{for}\ \ g \in \{5,\ 3,\ 1\}:\\
&\quad\quad \boldsymbol{for}\ i \in [g, n[:\\
&\quad\quad\quad j \leftarrow i\\
&\quad\quad\quad \boldsymbol{while}\ (j \geq g\ \wedge\ A[j] < A[j-g]):\\
&\quad\quad\quad\quad A[j] \rightleftarrows A[j-g]\\
&\quad\quad\quad\quad j \leftarrow j - g
\end{aligned}
$$

El primer “for” termina ya que recorre un conjunto finito

El segundo “for” termina ya que recorre un conjunto finito

Como J es monótonamente decreciente y G es finita, tenemos que el “while” siempre va a terminar ya que la primera condición se va a romper (J >= G).

Por lo tanto el algoritmo termina!

Algoritmos de Ordenación

Segundo criterio: ¿El algoritmo es **correcto**?

$$
\begin{array}{l}
\textbf{\textit{sort}}(A, n): \\
\quad \textbf{\textit{for}} \;\; g \in \{5,\ 3,\ 1\}: \\
\quad\quad \textbf{\textit{for}} \; i \in [g, n[: \\
\quad\quad\quad j \leftarrow i \\
\quad\quad\quad \textbf{\textit{while}} \; (j \geq g \;\wedge\; A[j] < A[j-g]): \\
\quad\quad\quad\quad A[j] \rightleftarrows A[j-g] \\
\quad\quad\quad\quad j \leftarrow j - g
\end{array}
$$

Opción 1:

Con G = 1, el algoritmo es igual a Insertion Sort. Como sabemos que Insertion Sort es correcto y siempre se va a ejecutar el algoritmo con G = 1, el algoritmo Sort() es correcto.

Algoritmos de Ordenación

$$
\begin{array}{l}
\textbf{sort}(A, n): \\
\quad \textbf{for}\ g \in \{5,\ 3,\ 1\}: \\
\quad\quad \textbf{for}\ i \in [g, n[: \\
\quad\quad\quad j \leftarrow i \\
\quad\quad\quad \textbf{while}\ (j \geq g\ \wedge\ A[j] < A[j-g]): \\
\quad\quad\quad\quad A[j] \rightleftarrows A[j-g] \\
\quad\quad\quad\quad j \leftarrow j - g
\end{array}
$$

Segundo criterio: ¿El algoritmo es correcto?

Opción 2:

Solo considerando con G = 1. Los otros valores de G no aportan a la demostración.

Invariante: Luego de la iteración i, el arreglo está ordenado hasta el índice i.
Lo demostramos por Inducción.

Caso Base. i = 1. El primer elemento del arreglo. Un arreglo de largo uno está siempre ordenado.

Hipótesis Inductiva. Tras la iteración i, A está ordenado hasta el índice i.

Algoritmos de Ordenación

$$
\begin{aligned}
&\boldsymbol{sort}(A, n):\\
&\quad \boldsymbol{for}\ \ g \in \{5,\ 3,\ 1\}:\\
&\quad\quad \boldsymbol{for}\ i \in [g, n[:\\
&\quad\quad\quad j \leftarrow i\\
&\quad\quad\quad \boldsymbol{while}\ (j \geq g\ \wedge\ A[j] < A[j-g]):\\
&\quad\quad\quad\quad A[j] \rightleftarrows A[j-g]\\
&\quad\quad\quad\quad j \leftarrow j - g
\end{aligned}
$$

En la iteración i+1 existen dos casos:

- $a_i \leq a_{i+1}$ -> A está ordenado hasta el índice i+1
- $a_i > a_{i+1}$ -> Tenemos lo siguiente

Llamemos $a_j = a_{i+1}$. Como ya estaba ordenado hasta a_i , se tiene
$a_1 <= a_2 <= \ldots <= a_{i-1} <= a_i > a_j$

En cada paso el elemento a_j se cambia de posición con el anterior, dejando ordenado a ambos lados:
$a_1 <= a_2 <= \ldots > a_j <= \ldots <= a_{i-1} <= a_i$ -> while continúa.
$a_1 <= a_2 <= \ldots <= a_j <= \ldots <= a_{i-1} <= a_i$ -> while termina, y los elementos están ordenados hasta el índice i+1.

Por inducción, después de la iteración n el arreglo está ordenado hasta el índice n, por lo tanto, está completamente ordenado.

Algoritmos de Ordenación

$\textbf{\textit{sort}}(A, n)$:
    $\textbf{\textit{for}}\ g \in \{5,\ 3,\ 1\}$:
        $\textbf{\textit{for}}\ i \in [g, n[$:
            $j \leftarrow i$
            $\textbf{\textit{while}}\ (j \geq g\ \wedge\ A[j] < A[j-g])$:
                $A[j] \rightleftarrows A[j-g]$
                $j \leftarrow j - g$

¿Cual es la complejidad en el peor caso?

Veamos la complejidad en las tres iteraciones de g

En g = 5: O((n−5)×n/5) = O(n×n/5) = O(n^2 / 5)

El for tiene n−5 iteraciones, y el while ejecuta en el peor de los casos n/5 veces por (caso donde el último elemento es el más pequeño de todos y se intercambia a la izquierda en posiciones de 5 en 5).

En g = 3: O((n−3)×n/3) = O(n×n/3) = O(n^2 / 3)

En g = 1: O((n−1)×n) = O(n^2)

Por lo tanto sumando la complejidad en las tres iteraciones de g

O(n^2 / 5)+O(n^2 / 3)+O(n^2) = O(n^2)

Heaps I3 2022 - 2

Como parte de la búsqueda de una Inteligencia Artificial General (AGI) un grupo de aficionados propone utilizar los miles (n) de modelos especializados de AI existentes (narrow IA) en paralelo, enviándoles la misma “pregunta” al mismo tiempo, y elegir como “respuesta” aquella con mayor confiabilidad (la respuesta de cada modelo viene acompañada de un valor entre 0 y 1 que indica la confiabilidad de la respuesta, siendo 1 el 100% de confianza). Actualmente han logrado construir el software necesario para enviar la pregunta a los modelos especializados y almacenar las respuestas en un arreglo A[0 . . . n – 1], en orden de llegada para los primeros 3 segundos.

a) **Proponga un algoritmo en pseudocódigo para la función Answer(A) que permite obtener de la manera más eficiente la respuesta de mayor confianza desde el arreglo A sin eliminarla.**

Solución

Suponemos que los datos se almacenan con atributos respuesta y confiabilidad. Además, luego de recibir los datos iniciales, se aplica BuildHeap(A) para transformar al arreglo A en un heap binario, donde se usa el atributo confiabilidad como prioridad. Con estos supuestos,

input : Arreglo A que representa un heap binario

$\texttt{Answer}(A)$:

  return $A[0].respuesta$

b) Un caso de uso del sistema es que el usuario pueda pedir la “siguiente mejor respuesta” al sistema, eliminándola del arreglo. Proponga el pseudo código para la función nextAnswer(A).

Solución

Dado el enunciado, puede interpretarse como entregar el elemento más prioritario o entregar el segundo más prioritario. Ambos se consideran correctos para efectos de la corrección. La implementación de ambos casos es

input : Arreglo A que representa un heap binario

$\texttt{nextAnswer}(A)$:

  return $\text{Extract}(A).\mathit{respuesta}$

$\texttt{nextAnswer}(A)$:

  if $A[1].\mathit{confiabilidad} \geq A[2].\mathit{confiabilidad}$:

    return $\text{Extract}(A, 1).\mathit{respuesta}$

  return $\text{Extract}(A, 2).\mathit{respuesta}$

En el segundo caso, se asume que Extract(A, i) opera de la misma forma que Extract visto en clase, pero extrayendo la posición indicada y restableciendo la propiedad de heap.

input : Arreglo A que representa un heap binario

`nextAnswer`(A):

  return `Extract`(A).*respuesta*

`nextAnswer`(A):

  if $A[1].\mathit{confiabilidad} \geq A[2].\mathit{confiabilidad}$ **:**

    return `Extract`($A, 1$).*respuesta*

  return `Extract`($A, 2$).*respuesta*

`Extract`(H):

  $i \leftarrow$ última celda no vacía de H

  $\mathit{best} \leftarrow H[0]$

  $H[0] \leftarrow H[i]$

  $H[i] \leftarrow \varnothing$

  `SiftDown`($H, 0$)

  return *best*

`SiftDown`(H, i):

  if *i tiene hijos* **:**

    $j \leftarrow$ hijo de i con mayor prioridad

    if $H[j] > H[i]$ **:**

      $H[j] \leftrightharpoons H[i]$

      `SiftDown`(H, j)

c) Dado que algunos modelos especializados son más lentos en responder se decide permitir que el arreglo A pueda recibir respuestas “tardías” (después de los 3 segundos iniciales) para ser consideradas en los algoritmos anteriores (una vez que dichas respuestas están disponibles). Proponga el pseudo código para la función lateAnswer(A, respuesta, confiabilidad) que permite incorporar de manera eficiente una respuesta nueva al arreglo A una vez que este ya fue “consultado” por los algoritmos anteriores.

Solución.

input : Arreglo A que representa un heap binario, respuesta y confiabilidad

$\texttt{lateAnswer}(A, respuesta, confiabilidad)$:

$i \leftarrow$ primera celda vacía de A

$A[i].\texttt{respuesta} \leftarrow respuesta$

$A[i].\texttt{confiabilidad} \leftarrow confiabilidad$

$\texttt{SiftUp}(A, i)$

$\texttt{SiftUp}(H, i)$:

if *i tiene padre* :

$j \leftarrow \lfloor i/2 \rfloor$

if $H[j] < H[i]$:

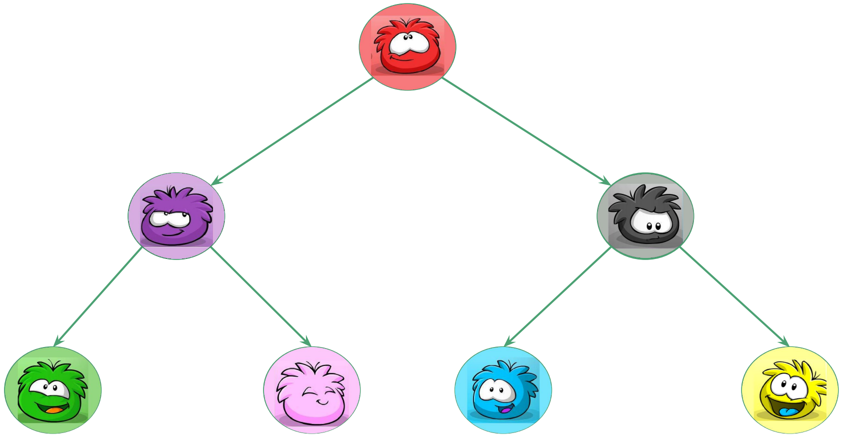
$H[j] \leftrightarrows H[i]$

$\texttt{SiftUp}(H, j)$

ABB, ABB-balanceado I1 2020-2

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

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 $\mathcal{O}(n+m)$ pasos para dejarlos como un solo ABB balanceado T con los nodos de ambos árboles.



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

Supongamos tenemos un **nodo arbitrario "y"** que debemos **posicionar como raíz**.

Es necesario hacer las **rotaciones** de tal forma que "**y"** quede en la raíz del árbol.

Recordar de clases:

- **Búsqueda** de un nodo en un ABB: complejidad **O(h)**
- **Rotaciones** para balancear: complejidad **O(1) y cada una puede subir al nodo "y" en un nivel**
- **Peor caso:** nodo **y** es una hoja

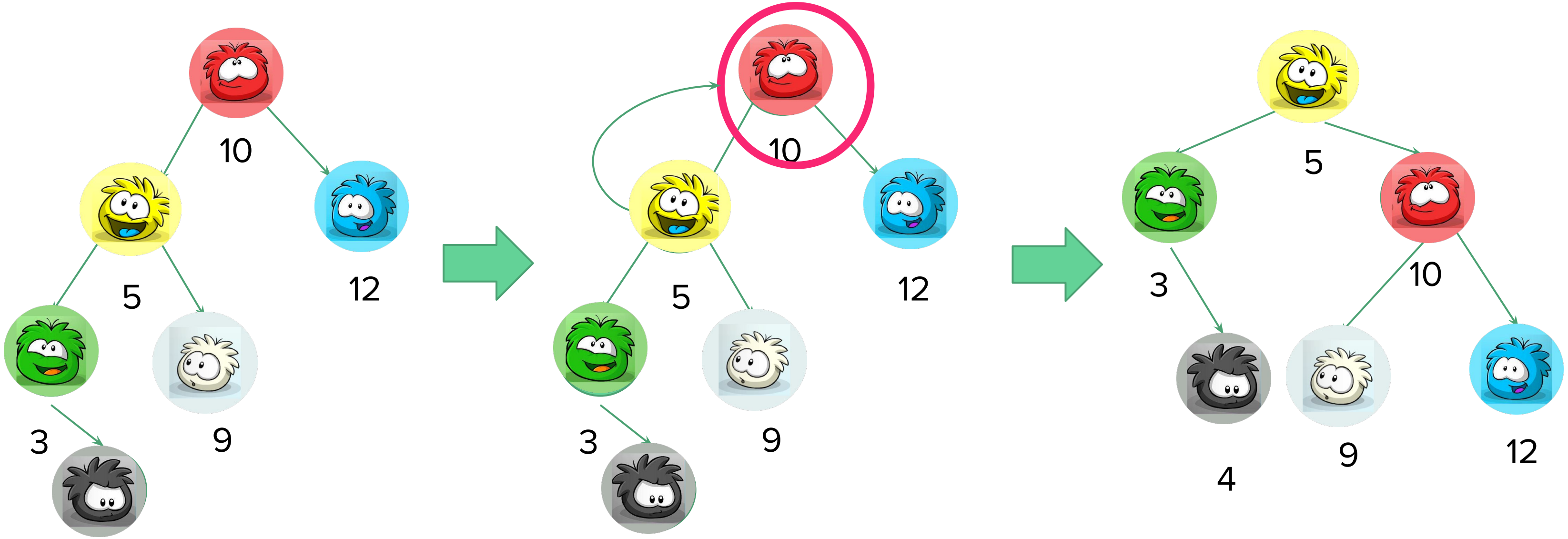
Algoritmo

```
procedure VolverRaiz(nodo y)
    while y tiene padre do
        x ← y.padre
        if y es hijo izquierdo de su padre then
            rotar hacia la derecha en torno a x-y        > Modifica al padre de "y"
        else
            rotar hacia la izquierda en torno a x-y      > Modifica al padre de "y"
        end if
    end while
end procedure
```

Viéndolo de cerca

if y es hijo izquierdo de su padre **then**
    rotar hacia la derecha en torno a x-y

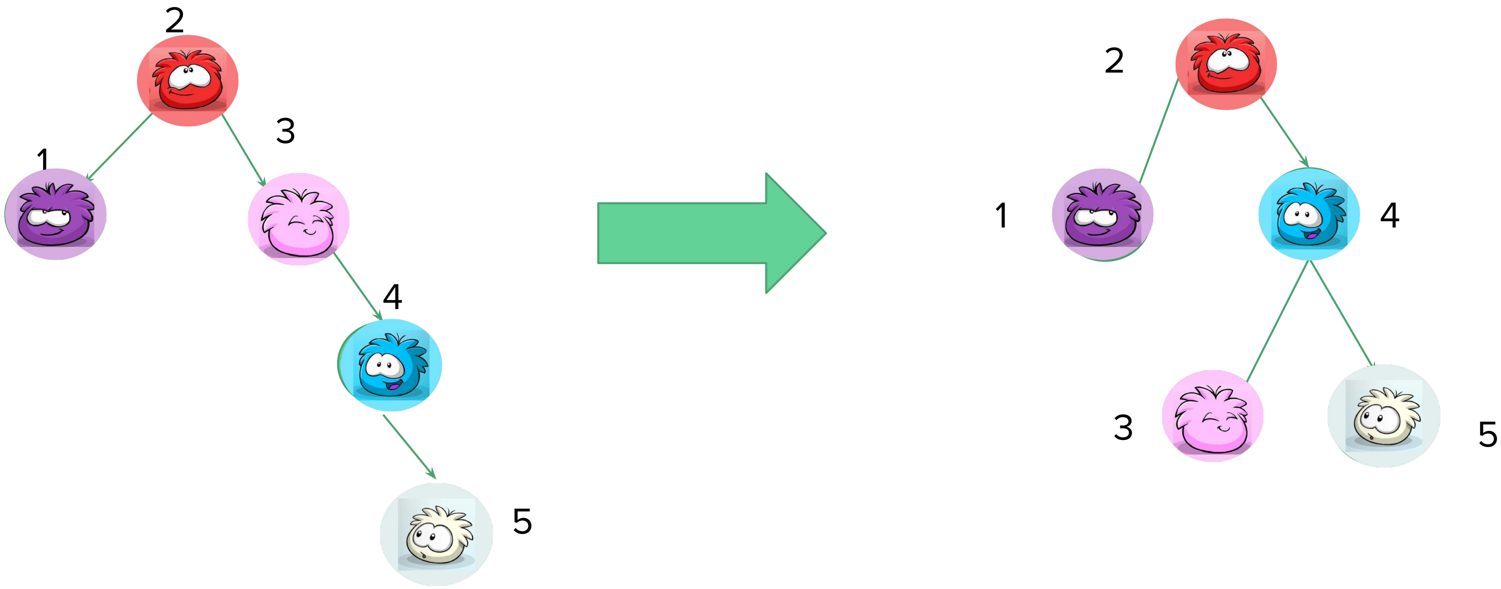
Ejemplo: nodo **y** sería el puffle amarillo



6: **else** > (nodo **"y"** es hijo derecho del padre)

7: rotar hacia la izquierda en torno a $\boldsymbol{x}$-$\boldsymbol{y}$

Ejemplo: nodo **y** sería el puffle celeste



Algoritmo

procedure VolverRaiz(nodo $\boldsymbol{y}$)
  while $\boldsymbol{y}$ tiene padre **do**
    $x \leftarrow y.padre$
    if $\boldsymbol{y}$ es hijo izquierdo de su padre **then**
      rotar hacia la derecha en torno a $\boldsymbol{x}$-$\boldsymbol{y}$
    else
      rotar hacia la izquierda en torno a $\boldsymbol{x}$-$\boldsymbol{y}$
    end if
  end while
end procedure

- **Búsqueda**: **O(h)**
- **Rotaciones en O(1) y cada una puede subir al nodo "y" en un nivel.**
- **Peor caso:** nodo **y** es una hoja, por lo que debe **subir h niveles**, lo que implica **O(h)** rotaciones.

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 $\mathcal{O}(n+m)$ pasos para dejarlos como un solo ABB balanceado T con los nodos de ambos árboles.

Basta con describir el proceso

Podríamos plantear una solución basada en lo siguiente:

1. Obtener un arreglo ordenado para cada árbol con sus nodos
2. Combinar estos 2 arreglos para tener un solo arreglo ordenado
3. Crear un ABB balanceado a partir del arreglo ordenado

Recordar justificar la complejidad del proceso!

1. Se **itera por sobre los nodos** de ambos árboles **de manera ordenada**, **copiando los nodos a un array**. Este paso consiste en un proceso recursivo que **visita siempre el nodo izquierdo antes que el derecho** (recorrido in-order o u otro algoritmo que lo logre). Haciendo esto tenemos **dos arreglos ordenados**, con los elementos de T1 y T2 respectivamente.
2. **Combinamos** estos **dos arreglos** usando la **subrutina merge** de MergeSort en **O(n + m)**. Con esto obtenemos un arreglo ordenado A con los n +m elementos de ambos árboles.
3. Finalmente, se **convierte el array ordenado A en un ABB balanceado** poniendo la **mediana como raíz** e insertando los valores menores y mayores a esta en las ramas izquierdas y derechas respectivamente. Notar que encontrar la mediana en un array ordenado es O(1) por lo que proceso se realiza en O(m+n) pasos.

La solución consta de tres pasos O(m+ n), por lo que la rutina completa es

O(m+ n).

Aclaración de cómo obtener el arreglo ordenado

Recordemos que un ABB tiene un orden total de los datos, por lo que podemos obtener las claves ordenadas haciendo un recorrido *in-order*, el cual es $\mathcal{O}(n)$. En clases se discutió como hacer un recorrido *in-order* del árbol para imprimir las claves en orden, de la siguiente manera:

```
procedure InOrder(árbol T)
    if T es un árbol then
        InOrder(T.left)
        imprimir T.key
        InOrder(T.right)
    end if
end procedure
```

Necesitamos modificar esta función para que en lugar de imprimir las claves en orden, nos entregue una lista ligada o arreglo con las claves en orden. Lo más simple es hacerlo sobre una lista:

```
procedure InOrder(árbol T, lista ligada L)
    if T es un árbol then
        InOrder(T.left)
        agregar (T.key, T.value) al final de L
        InOrder(T.right)
    end if
end procedure
```

Tras llamar a InOrder(T, L), tenemos en la lista L todos los pares $(key, value)$ de T, ordenados de menor a mayor clave. Esta lista podemos trivialmente convertirla a un arreglo A en $\mathcal{O}(n)$.

¡Mucha suerte en el estudio!

Equipo docente EDD