Estructuras de Datos Avanzadas
Tarea Examen 1

Miguel Concha Vázquez

28 de septiembre de 2018

1. Una familia de árboles es balanceada si todo árbol de la familia tiene altura $O(log(n))$, donde n es el número de nodos en el árbol. Determina si las siguientes familias de árboles binarias son familias balanceadas. Justifica tu respuesta.

   a) Todo nodo del árbol es una hoja o tiene dos hijos.

   b) Existe una constante c tal que, para cada nodo en el árbol, las alturas de sus subárboles difieren a lo más c.

   c) La profundidad promedio de un nodo es $O(log(n))$.

   Respuestas:

   Recordemos que usamos notación de *Big O* para expresar una cota asintótica superior al crecimiento de una función que depende den tamaño del ejemplar de entrada. Para una función $g(n)$, denotamos por $O(g(n))$ al conjunto de funciones[1]:

$$O(g(n)) = \{f(n) \mid \exists c, n_0 \in \mathbb{N}^+ \ni 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0\}$$

   a) Veamos que la familia de árboles en donde todo árbol es una hoja o tiene dos hijos no es balanceada. Denotemos por $\mathcal{F}$ a esta familia. Por contraejemplo mostremos un subconjunto infinito de árboles $\mathcal{G}$ que pertenecen a $\mathcal{F}$ al cumplir la propiedad ($\mathcal{G} \subset \mathcal{F}$) y cuya altura $h \notin O(logn)$ con n el número de nodos en los árboles[2].

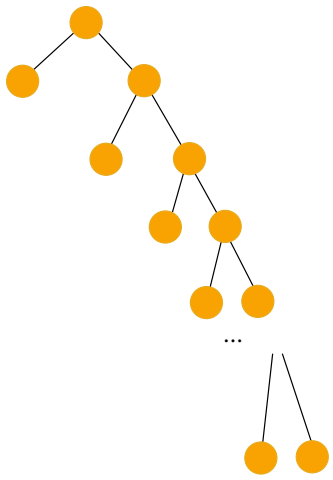
      Los árboles de este subconjunto son tales que tienen exactamente dos nodos en cada nivel —a excepción del nivel cero en donde solamente se tiene la raíz del árbol que denotamos por r—, con cada hijo derecho teniendo dos descendientes: un hijo izquierdo que es una hoja y un hijo derecho, hasta llegar al último nivel h en donde ambos nodos son hojas. Denotando por l_k al número de nodos en cada nivel k del árbol T descrito, con $T \in \mathcal{G}$, en otras palabras tenemos que:

      - $l_0 = 1$, pues en el nivel cero solamente tenemos a la raíz del árbol.
      - $l_i = 2, \forall i \in \{1, \ldots, h\}$, en donde se tienen exactamente dos nodos para todos los siguientes niveles.

[1] *CLRS, Introduction to Algorithms, Tercera ed., p. 47.*

[2] La altura de un nodo se define como el número de aristas en el camino más largo desde el nodo a un nodo hoja; la altura h de un árbol se define como la altura del nodo raíz.

Como se ilustra en la siguiente figura:



Sabemos que el total de nodos n debe ser igual a la suma de los nodos en todos los niveles, i.e.:

$$\sum_{i=0}^{h} l_i = l_0 + l_1 + \cdots + l_h = 1 + \underbrace{2 + \cdots + 2}_{h \text{ sumandos}} = n$$

En donde los h sumandos de dos unidades se tienen porque cada nivel del árbol tiene dos nodos excepto por el nivel cero en donde se ubica la raíz r y hay un total de $h+1$ niveles considerando nivel correspondiente a la raíz. Así:

$$\begin{aligned}
1 + 2h &= n \\
2h &= n - 1 && \text{Sumando } -1 \text{ de ambos lados de la igualdad.} \\
h &= \frac{n-1}{2} && \text{Multiplicando por } \frac{1}{2} \text{ de ambos lados de la igualdad.} \\
h &= \frac{1}{2}(n-1) && \text{Reacomodando.}
\end{aligned}$$

Y tenemos entonces que h la altura del árbol de $\mathcal{G}$ construido es del orden $O(n)$ puesto que $\exists c = 1, n_0 = 1 \in \mathbb{N}^+$ tales que $h = \frac{1}{2}(n-1) \leq c \cdot n = 1(n) = n, \forall n \geq 1$. Probamos rápidamente por inducción natural sobre n la aseveración previa:

Caso Base:

Para $n = 1$ se tiene que en efecto se cumple que $\frac{1}{2}(n-1) = \frac{1}{2}(1-1) = \frac{1}{2}(0) = 0 \leq n = 1$.

Hipótesis de Inducción:

Supongamos que para $n = k$ se cumple la propiedad, esto es, $\frac{1}{2}(k-1) \leq k$.

Paso Inductivo:

Veamos que la propiedad se satisface para $n = k + 1$, i.e., $\frac{1}{2}((k+1)-1) \leq k + 1$:

$$\begin{aligned}
\frac{1}{2}((k+1)-1) &= \frac{k}{2} && \text{Sumando y restando la unidad} \\
&= \frac{(k-1)+1}{2} && \text{Expresando } k \text{ como } k=(k-1)+1 \\
&= \frac{(k-1)}{2} + \frac{1}{2} && \text{Separando en dos fracciones} \\
&\leq k + \frac{1}{2} && \text{Aplicando la hipótesis de inducción} \\
&\leq k+1 && \frac{1}{2} \leq 1
\end{aligned}$$

Por transitividad de la desigualdad tenemos que $\frac{1}{2}((k+1)-1) \leq k+1$.

$\therefore \frac{1}{2}(n-1) \leq n, \forall n \geq 1$.
$\therefore h \in O(n)$.

Por otro lado, tenemos que $h \notin O(logn)$, pues no existen enteros positivos c y n_0 los cuales hagan que $h = \frac{1}{2}(n-1) \leq c \cdot logn, \forall n \geq n_0$.

Por tanto, se tiene que la familia de árboles binarios $\mathcal{F}$ de árboles en donde todos los nodos de los árboles son hojas o tienen dos hijos no es una familia balanceada.

b) Veamos que la familia $\mathcal{F}$ de árboles en donde existe una constante c tal que, para cada nodo en el árbol las alturas de sus subárboles difieren a los más c es balanceada, i.e., tienen altura $O(log(n))$. Partimos de las ideas detrás de la demostración de que los árboles AVL son balanceados, con los árboles $AVL \in \mathcal{F}$, pues en este caso particular $c = 1$.

Denotemos por $N(h)$ al mínimo de número de nodos que puede haber en un árbol T cuya altura es h, con $T \in \mathcal{F}$. Los árboles con el mínimo número de nodos representan el peor escenario posible pues todos los subárboles de todos los nodos van a diferir en c. Esto es, denotando por l y r a los subárboles derechos de los nodos, se tiene que:

$$\forall v \in T, |h(v.l) - h(v.r)| \leq c$$

Tenemos:

- Si $h = 0$, entonces $N(0) = 1$, pues mínimamente debe haber un nodo para que el árbol tenga altura de cero.
- Si $h = 1$, entonces $N(1) = 2$, pues por lo menos deben haber dos nodos en el árbol para que tenga altura de dos.

  Notemos que en general se tiene que para una altura h:

  $$N(h) = 1 + N(h-1) + N(h-1-c)$$

  Esto se debe a que alguno de los dos subárboles deberá tener una altura de $h-1$ para que entonces el árbol en su totalidad con el nodo de la raíz cumpla con tener altura h y se busca que dicho subárbol tenga el menor número posible de nodos y cumpla la propiedad de la familia. Por el otro lado, el otro árbol en el peor de los casos difiere en

c con respecto al primero para cumplir la propiedad. Suponiendo por contradicción que $N(h) = 1+N(h-1)+N(x)$, con $x > h-1-c$, sería entonces posible quitarle nodos a dicho árbol y se seguiría satisfaciendo la propiedad pues $|(h-1)-x| < |(h-1)-(h-1-c)| = c$ ya que $x > h-1-c$, llegando a una contradicción de que $N(x)$ era el mínimo número de nodos en un árbol de altura x que satisfaga la propiedad.

Así, se desarrolla:

$$\begin{aligned}
N(h) &= 1 + N(h-1) + N(h-1-c) \\
&= 1 + \underbrace{1 + N(h-2) + N(h-2-c)}_{N(h-1)} + \underbrace{1 + N(h-2-c) + N(h-2-2c)}_{N(h-1-c)} \\
&= 3 + N(h-2) + 2 \cdot N(h-2-c) + N(h-2-2c) && \text{Agrupando términos} \\
&> 2 \cdot N(h-2-c)
\end{aligned}$$

Lo anterior puede expandirse iterativamente para resolver la recurrencia tomando en cuenta que ya se vio que $N(0) = 1$. Entonces:

$$N(h) > 2 \cdot N(h-2-c) > 2 \cdot (\underbrace{2 \cdot N(h-4-2c)}_{\text{Desarrollando de } N(h-2-c)}) > 2 \cdot (2 \cdot (\underbrace{2 \cdot N(h-6-3c)}_{\text{Dessarrollando de } N(h-4-2c)})) > \cdots >$$

$$\underbrace{2 \cdot 2 \cdot 2 \cdot \cdots \cdot 1}_{2^{\frac{h}{2+c}}}$$

Se obtiene $2^{\frac{h}{2+c}}$ fruto de que se expanden $\frac{h}{2+c} - 1$ factores de dos al ir decrementando en cada iteración en $2 + c$ y el último factor es de una unidad correspondiente al caso base. De lo anterior se tiene entonces que:

$$\begin{aligned}
N(h) &> 2^{\frac{h}{2+c}} && \text{Transitividad de la desigualdad} \\
log(N(h)) &> log(2^{\frac{h}{2+c}}) && \text{Tomando logaritmo (base 2) de ambos lados (creciente)} \\
log(N(h)) &> \frac{h}{2+c} && \text{Definición de logaritmo} \\
(2+c) \cdot log(N(h)) &> h && \text{Multiplicando ambos lados de la desigualdad por } 2+c. \\
(2+c) \cdot log(n) &> h && n \geq N(n) \text{ por como se definió.}
\end{aligned}$$

Entonces, se tiene que $h < (2+c)log(n) = 2logn + clogn$ y por definición de O-grande, se tiene que $(2+c)log(n) \in O(log(n))$.

$\therefore h \in O(log(n))$ por transitividad.

$\therefore$ La familia de árboles $\mathcal{F}$ en donde existe una constante c tal que para cada nodo en el árbol las alturas de sus subárboles difieren a lo más c, es balanceada. $\square$

c) Veamos que la familia de árboles en donde todo árbol cumple con que la profundidad[3] promedio de un nodo es $O(log(n))$ no es balanceada. Denotemos por $\mathcal{F}$ a esta familia. Por contraejemplo mostremos un subconjunto infinito de árboles $\mathcal{G}$ que pertenecen a $\mathcal{F}$ al cumplir la propiedad ($\mathcal{G} \subset \mathcal{F}$) y cuya altura $h \notin O(logn)$ con n el número de nodos en los árboles.

Estos árboles T son tales que tienen una rama con un total de $\sqrt{n}$ nodos y el resto de los nodos $(n - \sqrt{n})$ están totalmente balanceados al disponerlos como un árbol completo, por lo que su profundidad es del orden $O(log(n - \sqrt{n})) = O(logn)$. Entonces, considerando que la rama de longitud $\sqrt{n}$ que se extiende hacia alguno de los lados puede estar colgada de cualquiera de los otros nodos cuya máxima profundidad es logarítmica, entonces la altura del árbol será de al menos $\sqrt{n} \notin O(log(n))$. Para ver que dichos árboles pertenecen a $\mathcal{F}$, solamente resta ver que la profundidad promedio de los nodos es del orden $O(log(n))$.

Estimemos la profundidad máxima posible para todos los nodos y al final dividamos entre n, el total de nodos, para obtener el promedio ponderado de las profundidades. Tenemos así que si denotamos por p_T a la suma máxima posible de las profundidades de todos los nodos en el árbol, entonces:

$$\begin{aligned} p_T &= (n - \sqrt{n}) \cdot log(n) + \sqrt{n} \cdot (\sqrt{n} + log(n)) && (1) \\ &= nlog(n) - \sqrt{n}log(n) + \sqrt{n}\sqrt{n} + \sqrt{n}log(n) \quad \text{Distribuyendo} && (2) \\ &= nlog(n) + n \quad \text{Cancelando términos y simplificando} && (3) \end{aligned}$$

Donde la igualdad en (1) se tiene porque hay $n - \sqrt{n}$ nodos con profundidad logarítmica y $\sqrt{n}$ nodos que en el peor de los casos tienen profundidad $\sqrt{n} + log(n)$.

Dividiendo p_T entre los n nodos para calcular la profundidad promedio p_P:

$$p_P = \frac{p_T}{n} = \frac{nlog(n) + n}{n} = \frac{nlog(n)}{n} + \frac{n}{n} = \frac{log(n)}{n} + 1 \in O(log(n))$$

Por tanto, se tiene que la familia de árboles binarios $\mathcal{F}$ de árboles en donde la profundidad promedio de todos los nodos es del orden $O(log(n))$ no es balanceada, pues vimos que la altura puede ser $h \notin O(log(n))$.

[3]La profundidad de un nodo se define como el número de aristas de la raíz al nodo.

2. Muestra que dado un conjunto T de n nodos $x_1, x_2, \dots x_n$ con valores y prioridades distintas, el árbol treap asociado a T es único.

   Solución:

   Sea $T = \{x_1, x_2, \dots, x_n\}$ un conjunto de n nodos con valores y prioridades que denotaremos por v y p, respectivamente, tal que se cumple que $x_i.v \neq x_j.v$ y $x_i.p \neq x_j.p$ para todo $i \neq j$, $i, j \in \{1, \dots, n\}$. Veamos que el árbol treap asociado al conjunto T es único.

   Se procede a probarlo por inducción fuerte sobre n, el número de nodos en T:

   Caso Base: Consideremos el caso cuando $n = 0$. Dado que entonces el árbol treap asociado a $T = \varnothing$ es vacío, trivialmente se satisfacen las propiedades del treap y es único.

   Hipótesis de Inducción: Supongamos que para $n \leq k$ se cumple que el árbol treap asociado a $T = \{x_1, x_2, \dots, x_k\}$ de k nodos con valores y probabilidades distintas es único.

   Paso inductivo: Veamos que el enunciado es cierto para $n = k + 1$, i.e., el árbol treap asociado a $T = \{x_1, x_2, \dots, x_k, x_{k+1}\}$ de nodos con valores y probabilidades distintas es único. La raíz de dicho árbol solamente puede corresponder al nodo de T tal que tenga la menor (mayor) prioridad de entre todo el conjunto de nodos para que se satisfaga la propiedad de que si un nodo v es descendiente de un nodo u, entonces $v.p > u.p$ ($u.p > v.p$)[4][5]; denotemos por r a la raíz del árbol treap asociado. Ahora, se deberá cumplir que los nodos ubicados en el subárbol izquierdo enraizado en r son aquellos x_l tales que $x_l.v < r.v$ con $l \in \{1, \dots, k+1\}$ para que se cumple la propiedad del orden de los valores del árbol binario de búsqueda en el treap. De forma recíproca, los nodos en el subárbol derecho enraizado en r deberán ser aquellos x_l tales que $x_l.v > r.v$. Llamemos V al conjunto de nodos x_l tal que $x_l.v < r.v$ y W al conjunto de nodos x_l tal que $x_l.v > r.v$, con $V, W \subseteq T$.

   Notemos que los subárboles izquierdo y derecho enraizados en r son árbol treaps porque de lo contrario contradiría la hipótesis de que el árbol treap asociado que estamos considerando en la construcción en efecto es un treap al no cumplir alguna de las propiedades de orden. Más aún, son árboles asociados a conjuntos de nodos T_1 y T_2 con valores y probabilidades distintas, pues $T_1 = \{x_1, x_2, \dots, x_k, x_{k+1}\} \setminus (\{r\} \cup W)$, $T_2 = \{x_1, x_2, \dots, x_k, x_{k+1}\} \setminus (\{r\} \cup V)$. Mínimamente la cardinalidad de T_1 y T_2 decrece en una unidad con respecto a $|T|$ al no considerar al nodo que constituye la raíz del árbol treap asociado a T, r. En consecuencia directa, aplicando la hipótesis de inducción con los conjuntos de nodos T_1 y T_2 pues $|T_1| \leq |T| - 1$ y $|T_2| \leq |T| - 1$, se tiene que cada subárbol enraizado en r asociado a los conjuntos de nodos T_1 y T_2 son únicos. Por ende, el árbol treap asociado a T es único.

   $\therefore$ Dado un conjunto T de n nodos $x_1, x_2, \dots x_n$ con valores y prioridades distintas, el árbol treap asociado a T es único. $\square$

[4] *CLRS, Introduction to Algorithms, Tercera ed., pp. 334.*

[5] Se presenta el caso también en que la propiedad de las prioridades obedezca a la propiedad de orden en un *max-heap* entre paréntesis, pero la demostración es análoga.

3. Explica cómo generalizar un *min-max heap* para obtener en $O(1)$ tiempo el k-ésimo elemento.

Solución:

Recordemos que un *min-max heap* de un conjunto de valores S es un árbol binario de búsqueda T que cumple con las siguientes propiedades[6]

1) T cumple con tener la forma de un *heap*.
2) ◦ Los valores almacenados en los nodos de niveles pares son menores o iguales que los valores almacenados en sus nodos descendientes en caso de tener.
   ◦ Los valores almacenados en los nodos de niveles impares son mayores o iguales que los valores almacenados en sus nodos descendientes en caso de tener.

   En consecuencia el valor más pequeño es almacenado en en la raíz de T y el mayor está ubicado como valor de uno de los nodos hoja de T.

Además, como revisamos en el curso y se menciona en el artículo, es posible construirlos en tiempo lineal $O(n)$ con una adaptación del algoritmo de Floyd para construcción de *heaps*.

Profundicemos ahora en la última sección del artículo en donde se presenta el uso de varios *min-max heaps* como parte de la implementación de árboles de estadísticos de orden (*order statistics trees*).

En el caso general, tenemos un conjunto $K = \{k_1, \ldots, k_m\}$ de valores fijos predeterminados (valores de *ranquins*[7]) en donde $k_i < k_{i+1}, i \in \{1, \ldots, m\}$; a partir de estos valores es que podremos luego de la generalización preguntar por el k_i-ésimo elemento en tiempo constante, con $k_i \in K$. Para éste propósito se cuenta con un arreglo unidimensional $V = [0, \ldots, m+1]$ y un total de $m+1$ *min-max heaps*[8] $T_0, \ldots, T_m$. Se tiene que $V[0] = -\infty$ y $V[m+1] = \infty$.

En la i-ésima entrada del arreglo V se tendría en caso de haberlo el k_i-ésimo elemento más pequeño del conjunto de elementos S y en el correspondiente *min-max heap* a $V[i]$, T_i, se almacenarían los elementos cuyo *ranquin* (*rank*) es mayor o igual que $V[i]$ y menor estricto que $V[i+1]$.

Para obtener el k_i-ésimo elemento ($k_i \in K$), bastaría con hacer un recorrido de V con tal de hallar la k_i de interés. Se tiene que la cantidad de elementos de V es constante pues tiene $m+2$ entradas y m es constante; ergo, este recorrido toma tiempo $O(1)$. Luego, recurrimos al *min-max heap* asociado a la posición de V para la k_i. Dicho *min-max heap* tendrá a los elementos a partir de k_i hasta el mayor elemento menor estricto que el k_{i+1}-ésimo de S. Por la propiedad del *min-max heap*, el menor elemento se ubica en la raíz y entonces podemos devolver a dicho elemento correspondiente al k_i-ésimo en tiempo constante $O(1)$.

El hacer el recorrido de V nos tomó tiempo $O(1)$, así como también devolver la raíz del *min-max heap* asociado al comienzo de dicho intervalo de *ranquins* toma tiempo $O(1)$. Se sigue que la complejidad en tiempo para obtener el k_i-ésimo elemento es $O(1) + O(1) = O(1)$.

[6] *Atkinson, M. D; Sack, J.-R; Santoro, N.; Strothotte, T. (1986). Munro, Ian, ed. Min-Max Heaps and Generalized Priority Queues.*, donde se introdujeron los *min-max heaps* y su contraparte, los *max-min heaps*.

[7] Traducción propuesta de *ranking*.

[8] Ésto también abre la posibilidad de realizar consultas de rangos del tipo *encontrar los elementos cuyo ranquin esté entre* k_i *y* k_j *para* $k_i < k_j$.

Ahora, si se quiere considerar a todo el conjunto de *min-max heaps* asociados como uno solo que preserve las operaciones convencionales, se tienen que hacer modificaciones ligeras en las operaciones de los *min-max heaps*. Primero, al querer insertar un nuevo elemento, debemos hacer un recorrido de V viendo en cada caso si el número de elementos que mantiene el *min-max heap* a dicho índice tiene un número de elementos menor estricto que $V[i]$, en cuyo caso se procede a insertar el elemento a dicho *min-max heap* de forma tradicional. Nótese que la cantidad de elementos en cada *min-max heap* se puede ir manteniendo al incrementar una variable del *min-max heap* de ese índice en tiempo constante luego de cada inserción o decrementándola luego de cada eliminación. En contraparte, si el *min-max heap* en donde debe ir colocado el nuevo elemento ya tiene tantos elementos como indica $V[i]$, se procede a comparar el elemento con el mayor elemento de dicho *min-max heap*[9]; hay dos subcasos:

- En caso de que el elemento a insertar sea mayor que el mayor elemento actual en el *min-max heap*, entonces lo insertamos en el siguiente *min-meax heap* que sería T_{i+1}.
- En el caso contrario, al mayor elemento actual debemos pasarlo al siguiente *min-max heap* e introducir este nuevo elemento en T_i.

Luego, al eliminar el menor elemento del *min-max heap global*[10], se comienza checando cada T_i, con $i \in \{0, \dots m\}$ pues están en orden y hay un número constante de ellos dado que $|K| = O(1)$. Luego, si el *min-max heap* que vemos en un punto no es vacío, debemos extraer el elemento de la raíz que será por definición el menor y la raíz del siguiente *min-max heap* la pasamos a aquel del que acabamos de extraer al menor elemento.

Finalmente, al querer obtener el mayor elemento de todo le *min-max heap*, tenemos que comenzar por T_m por como fueron construios y extraer a su mayor elemento en tiempo constante al encontrarse en el siguiente nivel de la raíz. En cambio, si dicho *min-max heap* es vacío, debemos consultar el siguiente *min-max heap* T_{m-1} y así sucesivamente hasta encontrar el primero no vacío del cual extraer el mayor elemento.

Debido a que las operaciones añadidas en cada caso son un número constante y cada una tiempo complejidad del orden $O(1)$, entonces no se afectan las complejidades de las operaciones convencionales de los *min-max heaps*.

Para ilustrarlo más a fondo, en nuestro caso en el enunciado se pide obtener solamente el k-ésimo elemento, que corresponde al caso particular cuando $|K| = 1$ pues $K = \{k\}$ con $m = 1$. En dicho escenario, tendríamos que $V = [0, \dots, m+1] = [0, \dots, 1+1] = [0, 1, 2]$ con $V[0] = -\infty$, $V[1] = k$ y $V[2] = \infty$. En este caso se tiene $m + 1 = 1 + 1 = 2$ *min-max heaps*:

- T_0 que contiene los elementos cuyos *ranquins* son mayores o iguales que $V[0] = -\infty$ y menores estrictos que el *ranquin* de interés k.
- T_1 que contiene los elementos cuyos *ranquins* son mayores o iguales que $V[1] = k$ y el resto. En este *min-max heap* es entonces que se ubica el k-ésimo elemento en la raíz.

[9] Como comparamos con el máximo, está en el segundo nivel y así lo podemos obtener en tiempo constante $O(1)$.
[10] Evidentenemente no hay como tal un solo *min-max heap*, pero esto puede ser transparente para quien use la estructura.

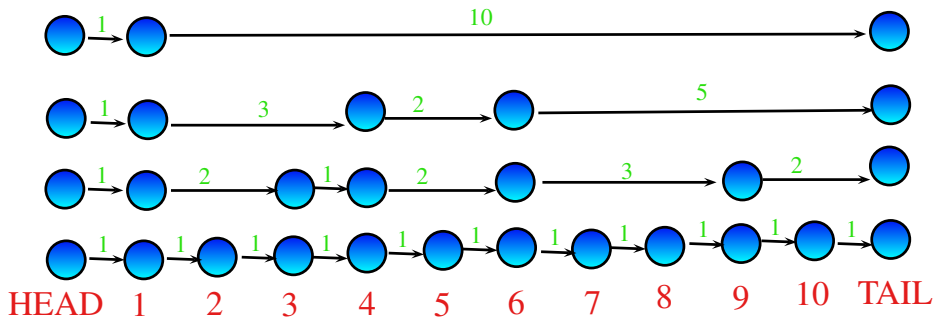
4. Describe cómo modificar una *skip-list* L para poder realizar las siguientes dos operaciones en tiempo esperado $O(logn)$:

   - Dado un índice i, obtener el elemento de L en la posición i.
   - Dado un valor x, obtener la cantidad de elementos en L menores a x.

   Soluciones:

   Supongamos por el momento que las búsquedas en las *skip-lists* toman tiempo esperado $O(logn)$; esto se probará a detalle más adelante. Es posible obtener listas de este tipo y que sean indexables introduciendo una ligera extensión.

   Lo que se añade es un valor numérico asociado a cada apuntador de L referido en la literatura como el *ancho* (*width*) del apuntador y definido como la cantidad de apuntadores del primer nivel de la lista que permite recorrer dicho apuntador, como en el siguiente ejemplo de color verde:



   Nótese que el *ancho* de un apuntador es exactamente igual a la suma de los anchos de los apuntadores entre los nodos que permite recorrer y que se encuentran en el nivel anterior. En consecuencia se tiene que la suma de los *anchos* de todos los niveles siempre es el mismo. También es de notar que se agregan *anchos* de una unidad entre los nodos de la cabeza de la lista y los primeros nodos de cada nivel; en el algoritmo se verá la razón de ésto.

   Para poder tener estos *anchos*, luego de la inserción convencional de un elemento en la *skiplist*, lo que se hace es ir a la lista en el nivel inferior en caso de haberla para poder determinar la cantidad de referencias que abarca la referencia previa, para luego proceder a asociarle su *ancho*. En el primer nivel, dado que no es posible bajar más, simplemente se indica que el apuntador siguiente permite dar un salto de una unidad, así que el *ancho* es de 1. Nótese que en el peor caso se tendría que recorrer todo el nivel anterior y entonces la inserción tomaría ahora el doble de tiempo que antes, pero sigue siendo del orden esperado $O(logn)$. Ahora, al eliminar un elemento primero se hace una búsqueda del mismo con una complejidad en tiempo esperada del orden $O(logn)$ y después de cerciorarnos de que está presente en la *skiplist*, comenzamos a bajar por los niveles buscando al elemento para eliminarlo; en el caso de los apuntadores cuyo elemento siguiente es mayor que el elemento que se busca eliminar, se tiene que sustraerles una unidad a sus *anchos* para que se refleje la eliminación del elemento.

   Tengamos en cuenta el algoritmo de búsqueda en la lista L expresado en el siguiente pseudocódigo:

```
1 BUSCA(L, elemento)
2   p = L.topleft
3   while p.below != NIL
```

```
4       p = p.below
5       while p.next.value <= elemento
6           p = p.next
7       return p
```

En donde a grandes rasgos nos movemos a lo largo de los niveles comenzando por los superiores en donde los apuntadores permiten dar *saltos* mayores. Cada vez que se compara con un nodo cuyo elemento almacenado es mayor que el que se busca, entonces debemos bajar al siguiente nivel en donde los *saltos* dados son menores para tratar de hallar el elemento en intervalo que antes se había saltado.

Ahora, para realizar las operaciones en tiempo esperado $O(logn)$, lo que haremos será:

- Para obtener el elemento de la lista L en la i-ésima posición, se llevará una cuenta del *ancho* acumulado mismo que se inicia en cero al ir sumando los *anchos* de los apuntadores que se van siguiendo:
  - Si el total acumulado más el *ancho* del apuntador siguiente coincide con la i dada como parámetro, entonces devolvemos el elemento al que apunta dicho apuntador siguiente.
  - Si se incrementa el *ancho* total acumulado con el *ancho* del siguiente apuntador y no se excede el valor de i pues es menor estricto, entonces nos mantenemos en el mismo nivel, pues significará que el *salto* nos llevará a un j-ésimo elemento, con $j > i$.
  - De lo contrario, así como el algoritmo de búsqueda convencional, debemos bajar un nivel y seguir con el proceso.

  Al final, puede ser que el índice provisto sea inválido, así que si no se regresó antes un elemento, se devuelve NIL.

  El pseudocódigo sería entonces:

```
1  BUSCA_CON_INDICE(L, i) //i es el indice
2    p = L.topleft
3    acu = 0
4    while p.below != NIL
5      p = p.below
6      while acu + p.next.width <= i
7        if acu + p.next.width == i
8          return p.next
9        p = p.next
10       acu += p.next.width
11   return NIL
```

  Dado que el algoritmo de búsqueda toma tiempo esperado $O(logn)$ y en esta modificación solamente se llevan a cabo un número constante de operaciones adicionales de orden constante en tiempo $O(1)$ que son las siguientes:

  - Inicializar el la suma total del ancho acumulado.
  - Una suma adicional en la línea 6 en contraste con la búsqueda tradicional.
  - Haciendo una comparación luego de una suma en la línea 7 y en dado caso devolviendo un valor en orden de tiempo $O(1)$.
  - Incrementando la variable del acumulado en la línea 10.

  Entonces el tiempo esperado del algoritmo sería del orden en tiempo $O(logn)+O(1) = O(logn)$ donde es $O(logn)$ esperado.

- Para poder determinar el número de elementos en la lista L que son menores (estrictos) que un elemento proporcionado x, la información la podemos obtener de igual forma de los *anchos*

de los apuntadores. De nueva cuenta se lleva un contador para la respuesta que corresponde al número de elementos menores que x y que se irá incrementando en cada iteración con el *ancho* de los apuntadores, pues este *ancho* es justamente el numero de elementos (nodos que tienen elementos almacenados) entre el elemento actual y al que hace referencia el apuntador. En cualquier paso:

- Si el elemento con el que se compara resulta ser mayor o igual que el elemento x, entonces no se incrementa con el ancho del apuntador, sino que se baja un nivel para verificar si existen en este intervalo otros elementos que cumplan con ser menores a x.
- En otro caso, entonces sabemos que hay tantos elementos adicionales a los ya considerados hasta el momento como demarca el *ancho* del apuntador al siguiente nodo, por lo que nos mantenemos en el mismo nivel, avanzamos al siguiente nodo e incrementamos el contador en el *ancho* del apuntador por el que nos movemos.

El pseudocódigo sería entonces:

```
1 CUENTA_NUMERO_MENORES(L, x) // x es el elemento de la pregunta
2   p = L.topleft
3   menores = 0
4   while p.below != NIL
5     p = p.below
6     while p.next.value <= x
7       menores += p.next.width
8       p = p.next
9   return menores
```

De nueva cuenta vemos que en esencia es el mismo algoritmo de búsqueda que tiene una complejidad en tiempo esperada $O(logn)$. Las operaciones adicionales:

- Inicializar la variable para la respuesta en la línea 3.
- Incrementar la variable en la línea 7.

son un número constante de operaciones adicionales, cada una de orden constante $O(1)$ en tiempo. Por tanto, la complejidad del algoritmo es del orden en tiempo esperado $O(logn) + O(1) = O(1)$.

Resta ver nada más que en efecto la complejidad esperada para el algoritmo de búsqueda en una *skiplist* es $O(logn)$ en tiempo. El primer ciclo del algoritmo es el encargado de ir bajando en los niveles de la lista L, mientras que el ciclo anidado permite ir avanzando en en mismo nivel en tanto el elemento almacenado en el nodo cumpla con ser menor o igual al que se busca. Veremos que la altura esperada de una *skiplist* es del orden $O(logn)$ en breve y en consecuencia directa el primer ciclo se ejecuta un número esperado del orden $O(logn)$.

En el caso del ciclo anidado para avanzar hacia delante en un nivel l, veremos que hay un tiempo esperado de búsqueda constante por dicho nivel. El argumento es que al recorrer dicho nivel, los nodos cuyos valores comparamos corresponden a columnas cuyo máximo nivel es l (excepto por los de la columna de la cola). Si procedemos por contradicción y suponemos que hay un nodo con un elemento v con el cual comparamos al buscar al elemento x y dicho nodo está en una columna cuyo nivel máximo es mayor que l, si se tiene que $x > v$, entonces seguiríamos el apuntador siguiente del nodo de la columna, pero como nos movemos de arriba hacia abajo, esto implicaría que también nos hubiéramos movido hacia adelante en el nivel máximo que alcanza dicha columna debido a que nunca regresamos a la izquierda en el recorrido ※ . La contradicción se tiene de suponer que se compara con un nodo que corresponde a una columna cuyo máximo nivel alcanza un $l' > l$, por lo que entonces solamente comparamos con nodos de columnas cuyo máximo nivel es l al movernos

hacia adelante en dicho nivel.

El número de nodos n_l en columnas cuyo nivel máximo es l con los cuales se compara al movernos hacia delante en un nivel l puede analizarse tomando en cuenta columna es generada independientemente de forma aleatoria con una distribución geométrica que tiene parámetro $p = \frac{1}{2}$ pues cada elemento de una columna C en un nivel i es elegido para aparecer en el siguiente nivel con una probabilidad de 0.5. De esta manera, el valor esperado para n_i sería el valor esperado de la variable que se distribuye geométricamente ($n_l \sim geo(\frac{1}{2})$[11]. Sabemos por el curso de probabilidad que la función de probabilidad para esta distribución de probabilidad discreta es:

$$f(x) = \begin{cases} p(1-p)^{x-1} & si\, x = 0, 1, \ldots \\ 0 & e.o.c. \end{cases}$$

Y la esperanza:

$$E(X) = \frac{1}{p}$$

De esta forma tenemos que:

$$E(n_l) = \frac{\frac{1}{1}}{\frac{1}{2}} = \frac{2}{1} = 2$$

Entonces, si demostramos que el número esperado de niveles sí es $O(logn)$, las búsquedas por cada nivel toman tiempo esperado constante $O(1)$ y entonces el algoritmo de búsqueda tomará tiempo esperado $O(logn)$.

Para ésto notemos primero que la probabilidad que un nodo aparezca en un nivel i de la *skiplist*, es la de $p = \underbrace{\frac{1}{2} \times \frac{1}{2} \times \cdots \times \frac{1}{2}}_{i \text{ veces}} = \frac{1}{2^i}$ porque para que aparezca en el siguiente nivel se tiene probabilidad de $\frac{1}{2}$ como ya se había dicho antes. Si consideramos entonces el número esperado de elementos para dicho nivel i de la lista, sería de $\frac{n}{2^i}$. Se define ahora una variable indicadora para cada nivel de la lista, suponiendo que los niveles van desde el más bajo que es el 1 hasta el infinito:

$$\delta_i \begin{cases} 1 & L_i \neq \varnothing \\ 0 & e.o.c. \end{cases}$$

De forma que si la lista en el nivel tiene al menos un nodo, entonces $\delta_i = 1$. En consecuencia, se tiene que la altura h de la *skiplist* va a ser:

$$h = \sum_{i=1}^{\infty} \delta_i \quad (1)$$

Pues en los niveles en donde ya no se tengan nodos, se sumaran ceros y solamente una unidad por cada nivel de la lista en donde haya al menos un nodo. Luego, como ya sabemos que el número de

[11] La distribución geométrica supone que se tiene una sucesión infinita de ensayos independientes Bernoulli en cada uno de los cuales la probabilidad de éxito es p y para cada una de las sucesiones se define la variable aleatoria como el número de éxitos antes del primer fracaso; aquí el fracaso sería que el valor buscado es mayor que el elemento con el que se compara y hay que descender en L.

nodos esperados para un nivel i es $\frac{n}{2^i}$ ($E(|L_i| = \frac{n}{2^i})$ y a lo más δ_i es $|L_i|$ cuando $|L_i| = 1$, se tiene por transitividad de la desigualdad que:

$$E(\delta_i) \leq \frac{n}{2^i} \quad (2)$$

Tomando la esperanza de ambos lados en (1):

$$E(h) = E(\sum_{i=1}^{\infty} \delta_i)$$

Al ser la esperanza lineal (curso de probabilidad), tenemos que la esperanza de la suma es la suma de las esperanzas y entonces:

$$E(h) = \sum_{i=1}^{\infty} E(\delta_i)$$

Separando en dos sumas:

$$E(h) = \underbrace{\sum_{i=1}^{\lfloor logn \rfloor} E(\delta_i)}_{(3)} + \underbrace{\sum_{i=\lfloor logn \rfloor + 1}^{\infty} E(\delta_i)}_{(4)}$$

A lo más la esperanza vale uno, por propiedades de la esperanza, entonces podemos reescribir (3) y ahora se tiene una desigualdad:

$$E(h) \leq \underbrace{\sum_{i=1}^{\lfloor logn \rfloor} 1}_{(3)} + \underbrace{\sum_{i=\lfloor logn \rfloor + 1}^{\infty} E(\delta_i)}_{(4)}$$

Usando (2) en (4), se tiene que:

$$E(h) \leq \underbrace{\sum_{i=1}^{\lfloor logn \rfloor} 1}_{(3)} + \underbrace{\sum_{i=\lfloor logn \rfloor + 1}^{\infty} \frac{n}{2^i}}_{(4)}$$

Después tenemos que (3) es $logn$, pues sumamos un número logarítmico de veces una unidad. En (4) se tiene que es igual a $\sum_{i=0}^{\infty} \frac{1}{2^i}$. Esto es:

$$E(h) \leq logn + \sum_{i=0}^{\infty} \frac{1}{2^i}$$

La suma es una serie geométrica con razón común $\frac{1}{2}$ y primer término 1. Así, converge a:

$$\frac{1}{1 - \frac{1}{2}} = \frac{1}{\frac{1}{2}} = 2$$

Finalmente:

$$E(h) \leq logn + 2 = O(logn)$$

$\therefore$ El algoritmo de búsqueda en las skiplists toman tiempo esperado $O(logn)$[12].

$\therefore$ Las operaciones adicionales que fueron agregadas toman tiempo esperado $O(logn)$ para llevar a cabo la búsqueda del i-ésimo elemento y determinar el número de elementos que son menores a un elemento con valor x.

Demostración de la esperanza de la distribución geométrica:

Tenemos la definición de la esperanza de una variable aleatoria:

$$E(Y) = \sum_{y} y f_Y(y)$$

En el caso de la variable aleatoria $X \sim geo(p)$:

$$E(X) = \sum_{x=1}^{\infty} x(1-p)^{x-1}p$$

$$\begin{aligned} E(X) &= \sum_{x=1}^{\infty} x(1-p)^{x-1}p \\ &= p\sum_{x=1}^{\infty} x(1-p)^{x-1} \quad \text{La constante sale de la suma} \end{aligned}$$

Notemos que $\frac{d}{da}a^i = ia^{i-1}$. Entonces, de cálculo se tiene que por la derivada término a término de una serie infinita y nombrando $q = (1-p)$:

$$\begin{aligned} &= p\frac{d}{dq}\sum_{x=1}^{\infty} q^x \\ &= p\frac{d}{dq}\sum_{x=0}^{\infty} q^x && \text{Con } x = 0,\ q^0 = 1 \text{ y la derivada es cero.} \\ &= p\frac{d}{dq}\frac{1}{1-q} && \text{Serie geométrica (*) con } a = 1 \text{ y } r = q. \\ &= p(\frac{1}{(1-q)^2}) && \text{Derivando} \\ &= \frac{p}{(1-(1-p))^2} && \text{Sustituyendo } q \text{ por } 1-p \\ &= \frac{p}{p^2} && \text{Simplificando} \\ &= \frac{1}{p} && \text{Simplificando} \end{aligned}$$

[12] Compyter Engineering, Open Data Structures (2013). Capítulo 4, *Skiplists*.

Para la serie geométrica de (*), tenemos:

$$S = \sum_{k=0}^{n-1} ar^k = \frac{a - ar^n}{1 - r}$$

Demostración:

Consideremos primero:

$$S = a + ar + ar^2 + \cdots + ar^{n-2} + ar^{n-1}$$

Luego multiplicamos lo anterior por r y nos queda:

$$Sr = ar + ar^2 + \cdots + ar^{n-1} + ar^n$$

De donde:

$$S - Sr = S(1 - r) = a + (ar - ar) + (ar^2 - ar^2) + \cdots + (-ar^n) = a - ar^n$$

Esto es:

$$S(1 - r) = a - ar^n = a(1 - r^n)$$

Finalmente, multiplicando ambos lados por $\frac{1}{1-r}$ se tiene:

$$S = \frac{a(1 - r^n)}{1 - r} \quad \blacksquare$$

Y la serie geométrica converge a $\frac{a}{1-r}$ cuando $|r| < 1$, esto es,

$$\lim_{n\to\infty} S = \lim_{n\to\infty} \sum_{k=0}^{n} ar^k = \frac{a}{1 - r}$$

5. Describe una secuencia de accesos a un árbol splay T de n nodos, con $n \geq 5$ impar, que resulta en T siendo una sola cadena de nodos en la que el camino para bajar en el árbol alterne entre hijo izquierdo e hijo derecho.

   Solución:

   Probemos primero el siguiente Lema:

   Lema: Si los n nodos en un árbol splay T son accedidos de forma secuencial[13], entonces el árbol resultante consistirá de una sola cadena de nodos que son hijos izquierdos.

   Demostración:

   Supongamos sin pérdida de generalidad que los nodos del árbol splay T son tales que sus valores van de 1 a n.

   Por inducción natural sobre n, el número de nodos en el árbol splay T:

   Caso Base: Veamos que la aseveración es cierta cuando $n = 1$. Evidentemente al buscar al nodo con el elemento 1, no se tiene que hacer nada debido a que al ser el único elemento ya se encuentra como raíz de T. Se cumple que este único nodo constituye una sola cadena de hijos izquierdos.

   Hipótesis de Inducción: Supongamos que el enunciado se satisface para $n = k$ nodos, i.e., si los nodos en un árbol splay de k con elementos $1, \ldots, k$ son accedidos de forma secuencias $(1, \ldots, k)$, entonces el árbol resultante —llamémosle T_k— es una cadena de hijos izquierdos.

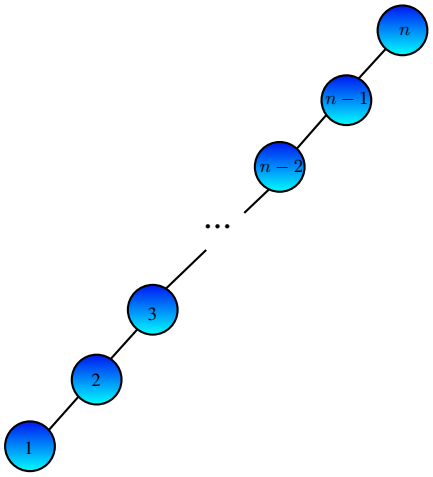
   Paso Inductivo: Sea $n = k+1$. Probemos que el enunciado es cierto en este caso, esto es, el acceso a los nodos $1, \ldots, k, k+1$ en orden secuencial resulta en una sola cadena de descendientes izquierdos. Luego de haber accedido los primeros k en orden secuencial $1, \ldots, k$, tenemos un árbol T'_k en donde los k nodos que fueron accedidos forman una cadena de hijos izquierdos. El último nodo con valor $k+1$ que aún no ha sido accedido solamente puede ubicarse a la derecha de la raíz como su hijo derecho por tratarse de un árbol splay y que esto implique que el último nodo accedido, el nodo k, está ahora en la raíz. En caso de que estuviera a la derecha de cualquier otro nodo distinto del nodo raíz, entonces se violaría la propiedad de árbol binario de búsqueda del árbol splay pues implicaría $k+1 < k$. Entonces, al buscar al nodo con elemento $k+1$ en el último acceso secuencial, se aplica una operación de *zag*, en donde el nodo con valor $k+1$ pasa a ser el padre del nodo que era previamente la raíz; dicho nodo con elemento k es ahora el hijo izquierdo del nodo con valor $k+1$ y se obtiene una cadena de hijos izquierdos pues la cadena que comenzaba desde el nodo con elemento k ya era una cadena de hijos izquierdos en T'_k por hipótesis de inducción. Por tanto, T_{k+1} constituye una sola cadena de hijos izquierdos.

   $\therefore$ Por el principio de inducción, se tiene que para cualquier árbol splay T de n nodos, un acceso secuencial a éstos produce una sola cadena de hijos izquierdos. □

   Continuando con el ejercicio, debemos proporcionar una secuencia de accesos (búsquedas de elementos) para cualquier árbol splay T con n nodos ($n = 2k-1, k \in \mathbb{Z}^+, n \geq 5$) que resulte en una cadena alternante de hijos izquierdos y derechos al bajar desde la raíz. Sin pérdida de generalidad, supongamos como en la demostración del Lema anterior que los valores que almacenan los nodos están numerados desde 1 hasta n, con n impar[14]. Luego, lo que se hace en primera instancia es acceder los elementos secuencialmente; por el Lema anterior, se obtiene un árbol splay T' que es una sola cadena consistente de hijos izquierdos:

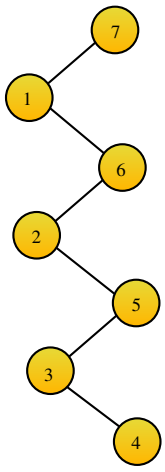
[13] Nos referimos a la operación búsqudda de los nodos, en donde el nodo buscado pasará a ser la raíz de T luego de llevar a cabo el *splaying*.

[14] En cualquier caso sería posible numerar a los elementos de cualquier árbol splay de 1 hasta n de acuerdo con el orden creciente de los valores que almacenan sus nodos y realizar la secuencia de accesos que se describirá.



Independientemente de cómo se encuentra originalmente T, accedemos a los n elementos en orden secuencial $1, 2, \ldots, n$, por lo que hacemos $\Theta(n)$ accesos y obtenemos T'. Ahora podemos proceder a transformar cualquier cadena con un número impar de nodos que son únicamente hijos derechos —excepto por la raíz— a una cadena que alterne entre hijos izquierdos y derechos. Meditemos por un momento cómo deberá ser tal cadena resultante.

El nodo raíz deberá ser el nodo con el mayor elemento, pues de lo contrario por la propiedad de BST, tendría al menos un descendiente derecho y ya no se cumpliría con el requerimiento del problema. Bajo un razonamiento similar, para que el siguiente nodo que es el descendiente izquierdo de la raíz no tenga hijos a la izquierda, entonces deberá ser el menor elemento de los n elementos almacenados en los nodos del splay T'. Siguiendo este proceso iterativamente, entonces los nodos que son hijos izquierdos irán incrementándose en una unidad por cada nivel que se desciende en el splay que buscamos T_F, mientras que los hijos derechos irán decrementándose en una unidad hasta llegar al último nodo que será un descendiente derecho ya que siempre comenzamos con un primer hijo izquierdo y n es impar. Ilustrando para el caso particular con $n = 7$:



Pensando a partir del árbol que queremos obtener y teniendo en cuenta que los accesos (búsquedas) llevan al nodo con el elemento a la raíz del árbol splay, entonces es evidente que los accesos tendrán que irse dando desde abajo hacia arriba, siendo el penúltimo elemento accesado el 1 y el último el mayor elemento del conjunto; en este caso el 7.

Evaluando para el primer caso (caso base), con $n = 5$ impar, tenemos que una secuencia de accesos que transforma T' en T_F es:

2,4,5,4,1,5

Podemos ver los árboles que van resultando luego de cada acceso con el visualizador de la Universidad de San Francisco:

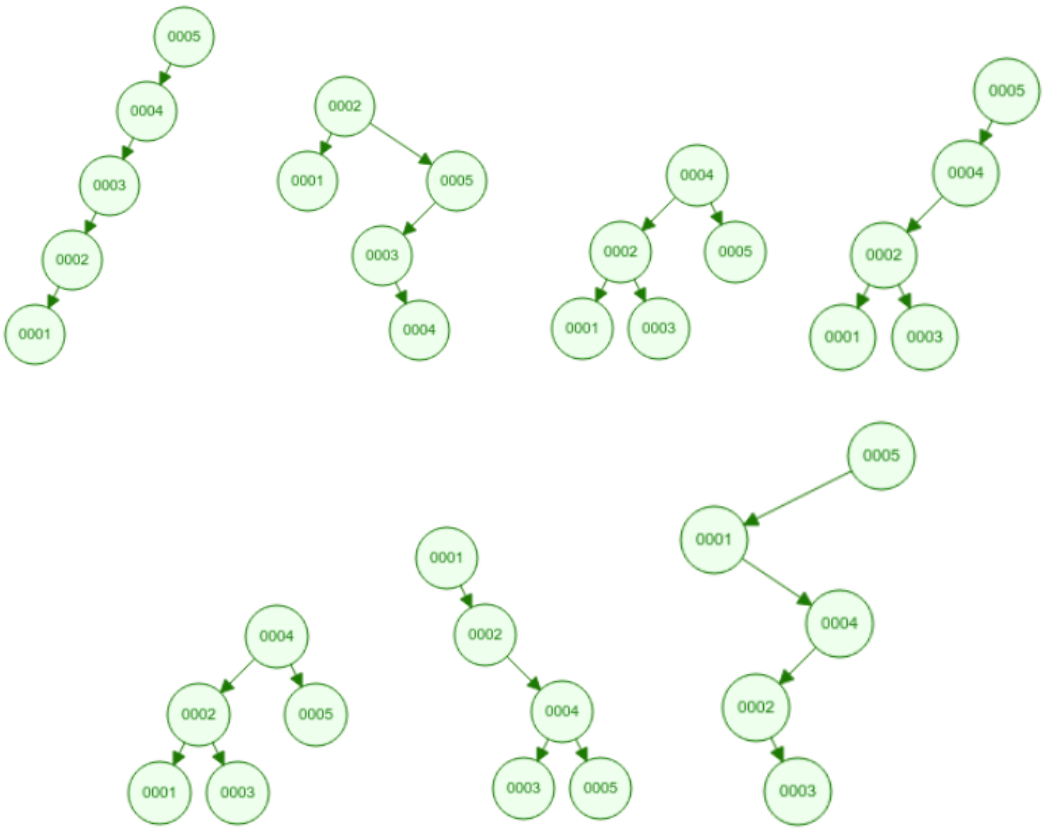


Figura 1: De izquierda a derecha, de arriba a abajo; accesos: 2,4,5,4,1,5.

Aquí notemos que regresamos al mismo árbol splay luego acceder al 4 por segunda ocasión. Sin embargo, ésto se hace debido a que el hijo derecho del 4 en el tercer árbol mostrado no necesariamente tiene almacenado al sucesor de 4, pues bien puede tenerlo en un subárbol izquierdo. Por tanto, lo que aseguramos con esta subsecuencia **5,4** es la de llevar al sucesor del cuatro a la raíz y luego regresar al que estaba antes, para que ahora sí sucede inevitablemente que su sucesor es su hijo derecho luego de ésto. Entonces, la secuencia de accesos la podemos analizar a partir del árbol que queremos obtener como sigue; en este caso lo hacemos para $n = 7$, con 7 impar:

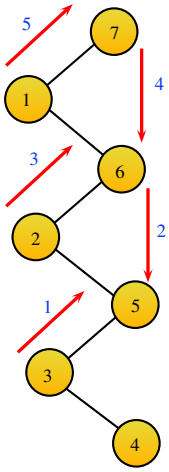
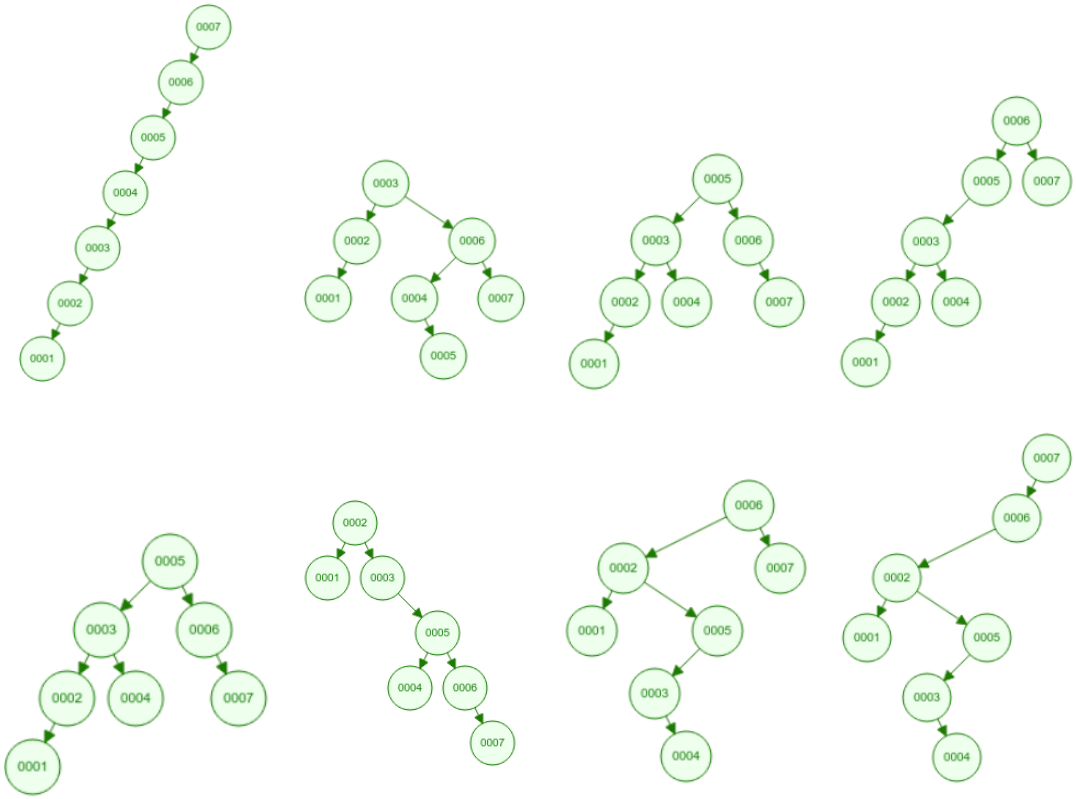


Figura 2: Para $n = 7$, la secuencia de accesos sería: 3,5,6,5,2,6,7,6,1,7.

Con el visualizador:



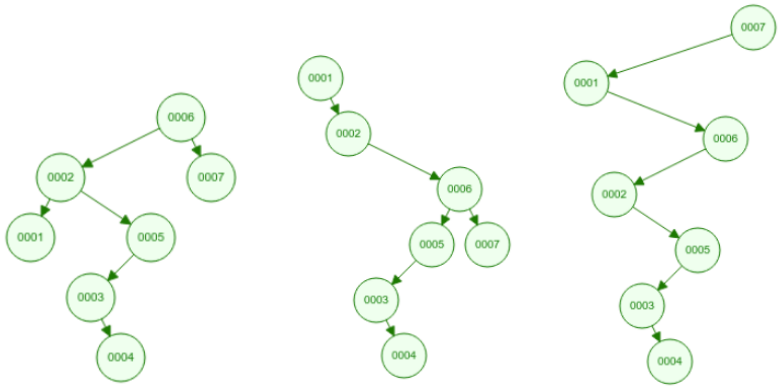


Figura 3: De izquierda a derecha, de arriba a abajo; accesos: 3,5,6,5,2,6,7,6,1,7.

Notemos que de nuevo al efectuar la subsecuencia de accesos **6,7,6** regresamos a un árbol que ya se tenía, pero lo hacemos para uniformar la descripción de la secuencia de accesos genérica que se dará a continuación.

Empezando en el último hijo izquierdo de de arriba hacia abajo en T_F —el árbol splay al que se quiere llegar—, a este nodo le corresponde el elemento $\lfloor \frac{n}{2} \rfloor$. Luego se accede al elemento que es dos unidades mayor que él, esto es, $\lfloor \frac{n}{2} \rfloor + 2$. Posteriormente se efectúa la subsecuencia de accesos $\lfloor \frac{n}{2} \rfloor + 3, \lfloor \frac{n}{2} \rfloor + 2$ y se continúa como se hizo al momento. Es decir, luego se haría $\lfloor \frac{n}{2} \rfloor - 1, \lfloor \frac{n}{2} \rfloor + 3, \lfloor \frac{n}{2} \rfloor + 4, \lfloor \frac{n}{2} \rfloor + 3$, después $\lfloor \frac{n}{2} \rfloor - 2, \lfloor \frac{n}{2} \rfloor + 4, \lfloor \frac{n}{2} \rfloor + 5, \lfloor \frac{n}{2} \rfloor + 4$ y así en lo sucesivo hasta llegar a tener que efectuar solamente dos accesos $1, n$ del menor y el mayor elemento.

Como ejemplos, consideremos los casos para $n = 5$ y $n = 7$:

- $n = 5$: La secuencia de accesos mencionada es en efecto:

$$\underbrace{\lfloor \frac{n}{2} \rfloor}_{\lfloor \frac{5}{2} \rfloor = \mathbf{2}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 2}_{2+2=\mathbf{4}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 3}_{2+3=\mathbf{5}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 2}_{2+2=\mathbf{4}}, \mathbf{1}, \mathbf{5}$$

- $n = 7$: La secuencia de accesos mencionada es en efecto:

$$\underbrace{\lfloor \frac{n}{2} \rfloor}_{\lfloor \frac{7}{2} \rfloor = \mathbf{3}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 2}_{3+2=\mathbf{5}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 3}_{3+3=\mathbf{6}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 2}_{3+2=\mathbf{5}}, \underbrace{\lfloor \frac{n}{2} \rfloor - 1}_{3-1=\mathbf{2}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 3}_{3+3=\mathbf{6}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 4}_{3+4=\mathbf{7}}, \underbrace{\lfloor \frac{n}{2} \rfloor + 3}_{3+3=\mathbf{6}}, \mathbf{1}, \mathbf{7}$$

Demostremos por inducción que la secuencia de accesos sugerida logra lo que se busca. Lo haremos por inducción natural sobre n el número de nodos en el árbol splay T' (ya en cadena de hijos izquierdos), con $n = 2r - 1, r \in \mathbb{Z}^+$ y $n \geq 5$.

Demostración: Por inducción natural sobre n con cambio de base.

Caso Base: Veamos que se cumple que la secuencia de accesos funciona para un árbol splay T' con $n = 5$ nodos. Ya se vio anteriormente que sí se cumple.

Hipótesis de Inducción: Supongamos que la secuencia de accesos para un árbol splay T' con $n = k = 2r' - 1$ nodos, $r' \in \mathbb{Z}^+$:

$$\lfloor \frac{k}{2} \rfloor, \lfloor \frac{k}{2} \rfloor + 2, \lfloor \frac{k}{2} \rfloor + 3, \lfloor \frac{k}{2} \rfloor + 2, \lfloor \frac{k}{2} \rfloor - 1, \lfloor \frac{k}{2} \rfloor + 3, \lfloor \frac{k}{2} \rfloor + 4, \lfloor \frac{k}{2} \rfloor + 3, \lfloor \frac{k}{2} \rfloor - 2, \lfloor \frac{k}{2} \rfloor + 4, \lfloor \frac{k}{2} \rfloor + 5, \lfloor \frac{k}{2} \rfloor + 4, \ldots, 1, k$$

da como resultado una sola cadena que alterna entre un hijo izquierdo y un hijo derecho.

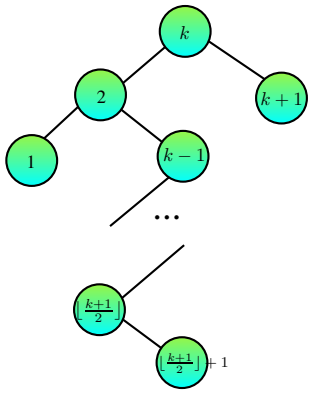
Paso Inductivo: Veamos que la secuencia se accesos funciona para un árbol splay T' con $n = k+2 = 2r'' - 1$ y $r'' = r' + 1$ para que sea un número impar de nodos. Sin considerar a los últimos dos nodos que deben aparecer como la raíz del árbol al tener al mayor elemento y su hijo izquierdo con el elemento más pequeño, el resto de valores son un total de k, mismos que pueden enumerarse de 1 a k y aplicar la secuencia para k con k impar de la hipótesis de inducción:

$$\lfloor\frac{k}{2}\rfloor, \lfloor\frac{k}{2}\rfloor+2, \lfloor\frac{k}{2}\rfloor+3, \lfloor\frac{k}{2}\rfloor+2, \lfloor\frac{k}{2}\rfloor-1, \lfloor\frac{k}{2}\rfloor+3, \lfloor\frac{k}{2}\rfloor+4, \lfloor\frac{k}{2}\rfloor+3, \lfloor\frac{k}{2}\rfloor-2, \lfloor\frac{k}{2}\rfloor+4, \lfloor\frac{k}{2}\rfloor+5, \lfloor\frac{k}{2}\rfloor+4, \ldots, 1, k$$

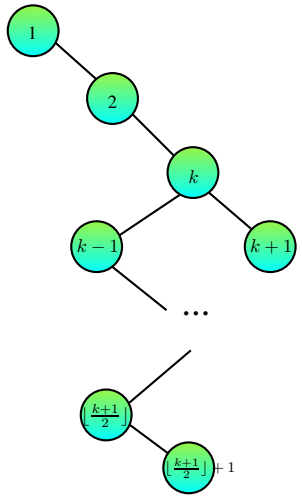
Luego de aplicar esta secuencia de accesos, el elemento más pequeño de los $k+1$ estará almacenado en el nodo que es descendiente izquierdo del hijo izquierdo del que ahora se encuentra en la raíz. Procedamos por contradicción y supongamos que no es así. De lo contrario aparecería del lado derecho de algún nodo o bien del lado izquierdo de algún nodo distinto del segundo más pequeño que actualmente es el hijo izquierdo de la raíz. Esto implica que hay un elemento más pequeño que el que supusimos menor que los demás, llegando a una contradicción ※ . Ergo, el más pequeño de los $k+1$ elementos cumple con estar en el nodo izquierdo del que actualmente es hijo izquierdo de la raíz.

De manera similar, el elemento más grande —elemento $k+2$— está almacenado en el nodo que es hijo derecho de la raíz. Nuevamente supongamos que no es así y lleguemos a una contradicción. Si estuviera ubicado en otra parte, sería hijo izquierdo de alguien o bien como hijo derecho de un nodo que aparece del lado izquierdo del que actualmente está en la raíz, así que en ambos casos esto implicaría que hay un elemento más grande que el que supusimos mayor ※ .

Esquemáticamente:



Finalmente, el penúltimo acceso de la secuencia para el menor elemento 1 ocasionará que se lleve a cabo la operaciones de *zig-zig* al hacer *splaying*. Esto dará lugar al siguiente árbol:



Así que con el último acceso de la secuencia al mayor elemento $k+1$ se harían dos operaciones en el *splaying*: primero un *zag-zag* y finalmente otro *zag*:

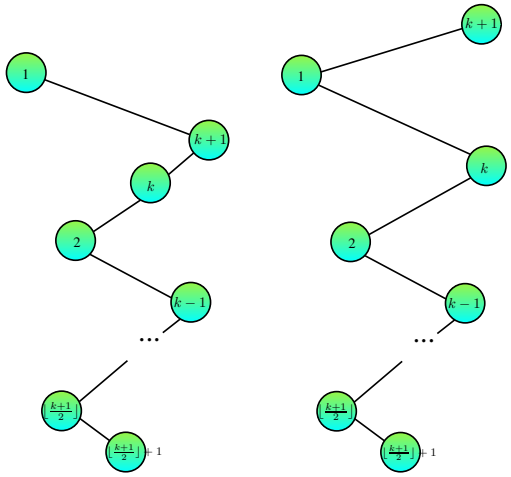


Figura 4: *zag-zag* y luego otro *zag*.

Por tanto, la secuencia obtiene el árbol T_F cuando $n = k + 2$.

$\therefore$ Por el principio de inducción, se cumple que la secuencia funciona para todo árbol splay con n nodos, con n impar y $n \geq 5$, en donde el árbol T original consiste de una cadena de hijos izquierdos. $\square$

Notemos que en por cada descendiente izquierdo en la cadena *zigzagueante* resultante se llevan a cabo cuatro accesos —menos para el último en donde solamente hay dos accesos– y hay un total de $\lfloor \frac{n}{2} \rfloor$ hijos izquierdos. Por tanto, el número de accesos en la secuencia es $\lfloor \frac{n}{2} \rfloor \cdot 4 - 2$, que es del orden lineal $O(n)$.

Para convertir cualquier árbol splay T de n nodos, con n impar y $n \geq 5$, la secuencia completa de accesos sería entonces:

$$1, 2 \ldots, n,$$
$$\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor + 2, \lfloor \frac{n}{2} \rfloor + 3, \lfloor \frac{n}{2} \rfloor + 2, \lfloor \frac{n}{2} \rfloor - 1, \lfloor \frac{n}{2} \rfloor + 3, \lfloor \frac{n}{2} \rfloor + 4, \lfloor \frac{n}{2} \rfloor + 3, \lfloor \frac{n}{2} \rfloor - 2, \lfloor \frac{n}{2} \rfloor + 4, \lfloor \frac{n}{2} \rfloor + 5, \lfloor \frac{n}{2} \rfloor + 4, \ldots, 1, k$$

Ya tomando en cuenta en obtener primero el árbol como cadena de hijos derechos T' y luego aplicar la otra subsecuencia de accesos. Entonces, por lo anterior el número de accesos sería del orden $\Theta(n) + O(n) = O(n)$.

6. Demuestra o da un contraejemplo:

   a) Los nodos de cualquier árbol AVL pueden colorearse de rojo y negro para obtener un árbol rojo-negro válido.
   b) Cualquier árbol rojo-negro satisface las propiedades de árbol AVL.

   Soluciones:

   Antes de comenzar con los ejercicios, recordemos brevemente que un árbol rojo-negro es un árbol binario de búsqueda que satisface las siguiente propiedades:[15]

   - Cada nodo del árbol es rojo o negro.
   - La raíz del árbol es negra.
   - Cada hoja (NIL) es negra.
   - Si un nodo es rojo, entonces sus dos descendientes son negros.
   - Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

   Llamemos *profundidad negra* de un camino desde a un nodo a alguna hoja como este número de nodos de color negro encontrados al bajar por dicho camino.

   Por su parte, en el caso de los árboles AVL —otra particularización de BSTs— se da una definición recursiva:

   - Un árbol vacío es un árbol AVL.
   - Si T es un árbol no vacío y T_i y T_d sus dos subárboles, entonces T es AVL si y sólo si:
     - T_i es AVL.
     - T_d es AVL.
     - $|H(T_i) - H(T_d)| \leq 1$ con la función de altura $H : BST \to \mathbb{N}$ para un árbol binario de búsqueda T con subárboles izquierdo y derecho T_i y T_d definida como:

$$H(T) = \begin{cases} 0 & si\ T\ tiene\ solamente\ un\ nodo \\ 1 + max(H(T_i), H(T_d)) & e.o.c \end{cases}$$

   Cabe mencionar que la altura para las hojas de define para estos árboles como una unidad, mientras que la altura para un nodo v se define como:

$$max(v.i.h, v.d.h) + 1$$

   Donde i y d son los nodos izquierdos y derechos del nodo v y h es la altura del nodo.

   a) Ralf Hinze publicó en el artículo Constructing Red-Black Trees un algoritmo de coloración de árboles binarios de búsqueda que generalizan un algoritmo para poder colorear árboles AVL como árboles rojo-negros; en este caso se dará un algoritmo particular de coloración de árboles AVL en árboles rojo-negros y se probará la correctud del algoritmo.

      Tengamos presente que los nodos de los árboles AVL tienen además del elemento y sus nodos izquierdo y derecho, una altura. Para poder colorearlos en el algoritmo se les agrega un atributo adicional de color.
      EL algoritmo es recursivo y el pseudocódigo es el siguiente:

[15] *CLRS, Introduction to Algorithms, Tercera ed., p. 308.*

```
1  COLOREA_AVL(r) //raiz del arbol
2    if  r == NIL
3      return      // Hemos terminado o no hay nada que hacer.
4    r.color = BLACK
5    COLOREA_AVL(r.i)
6    COLOREA_AVL(r.d)
7    if  r.h  % 2 == 1
8      if  r.i  != NIL AND r.i.h  % 2 == 0
9        r.l.color = RED
10     if  r.d != NIL AND r.d.h  % 2 == 0
11       r.d.color = RED
```

En las líneas 7-11 se verifica si la altura del nodo actual es impar, en cuyo caso se corroboran las alturas de los dos hijos izquierdo y derecho; si la altura de uno de ellos —que no debe ser vacío— resulta ser par, entonces se colorea de rojo.

Vemos que el algoritmo fácilmente satisface las primeras dos propiedades de los árboles rojo-negros debido a que la raíz de entrada se colorea de color negro y además las llamadas recursivas sobre los dos subárboles AVL se encargan de colorear a todos los nodos de negro y en algunos casos los hijos que ya habían sido en principio coloreados de negro son repintados de rojo cuando ocurre la condición antes descrita. De esta forma, se cumple que cada nodo del árbol es rojo o negro.

Luego, las hojas que tienen una altura de una unidad son coloreadas de negro debido a que $1 \equiv 1(mod 2)$, esto es, es impar y por tanto no se colorean de rojo. Los nodos rojos son aquellos que tenían una altura par para que de entrada fueran recoloreados, por lo que sus descendientes tendrán una altura impar y por ende no se les coloreará nuevamente de rojo; así, los hijos de los nodos de color rojo son negros. Falta ver únicamente que cada camino de cada nodo a las hojas tiene la misma *altura negra* para que el algoritmo sea correcto y es fácil ver que su complejidad será del orden $O(n)$ pues se trata de un recorrido en preorden del árbol que tiene n nodos y en cada paso se llevan a cabo un número constante de operaciones de orden constante. Formalmente, La recurrencia sería $T(n) = 2T(\frac{n}{2})+1$ al tener cada nodo a lo más dos descendientes (izquierdo y derecho) y en cada paso hacer un número constante de operaciones del orden $O(1)$. Desarrollando:

$$T(n) = 2T(\frac{n}{2}) + 1$$

$$T(n) = 2(2T(\frac{n}{4}) + 1) + 1 = 4T(\frac{n}{4}) + 3$$

$$T(n) = 2(2(2T(\frac{n}{8}) + 1) + 1) + 1 = 6T(\frac{n}{8}) + 5$$

$$\vdots$$

$$T(n) = nT(1) + (n - 1)$$

$$T(n) = n + (n - 1)$$

$$\Rightarrow T(n) = O(n)$$

La demostración que se cumple la propiedad de las alturas negras luego de ejecutar el algoritmo en un árbol AVL se hará por inducción sobre la altura de un nodo en el árbol v desde el cual se manda llamar la función `COLOREA_AVL`; se probará que:

- Si la altura de v es impar, i.e., $v.h = 2q - 1$, $q \in \mathbb{Z}^+$, entonces se obtiene un subárbol rojo-negro enraizado en v y la profundidad negra de v es q.
- Si la altura de v es par, i.e., $v.h = 2q, q \in \mathbb{Z}^+$, entonces se obtiene un subárbol rojo-negro enraizado en v y la profundidad negra de v es:
  - q si se colorea de rojo.
  - $q + 1$ si se colorea de negro.

Tratemos los dos casos por separado:

- Demostración para $v.h$ impar: Por inducción natural sobre $v.h$ que se abreviará como h, con $h = 2q - 1, q \in \mathbb{Z}^+$.

  Caso Base: $h = 1$, con $1 = 2(1) - 1$ impar. Por definición de altura de nodos en árboles AVL, se tiene que v es una hoja; entonces el nodo es coloreado de negro solamente, nos es recoloreado de rojo al no ser su altura par y se cumple que la *profundidad negra* de v es entonces $h = 1$ al haber solamente un nodo negro.
  Hipótesis de Inducción: Supongamos que para $h = k$, con $k = 2m - 1$, $m \in \mathbb{Z}^+$ el llamar la función de `COLOREA_AVL` con parámetro del nodo v obtiene una *profundidad negra* para v de m.
  Paso Inductivo: Veamos que para $h = k + 2 = 2m - 1 + 2 = 2m + 1 = 2(m + 1) - 1$ se cumple que al mandar llamar la función de `COLOREA_AVL` desde v se obtiene para v una *profundidad negra* para v de $m + 1$. Hay dos casos:
  - Los dos descendientes de v tienen altura par $2m$, i.e., $v.i.h = v.d.h = 2m$. Entonces se entra al caso de recoloración de los dos hijos izquierdo y derecho a color rojo y por la demostración que se da abajo[16], se tiene que la *profundidad negra* para $v.i$ y $v.d$ es de m. Luego, como se colorea de negro a v, entonces la *profundidad negra* de v es de $m + 1$, que es lo que se buscaba.
  - Uno de los hijos tiene altura $2m$ y el otro tiene altura $2m-1$[17]. Supongamos sin pérdida de generalidad que $v.i.h = 2m$ y $v.d.h = 2m - 1$. Entonces, para el hijo izquierdo se entra al caso en donde es necesario recolorear de color rojo, mientras que al derecho no por no ser par su altura. Por la siguiente demostración que se da, la *profundidad negra* resultante para $v.i$ sería de m, mientras que para el hijo derecho, por la hipótesis de inducción se obtiene una *profundidad negra* de m. Entonces la *profundidad negra* obtenida para v luego del proceso sería de $m + 1$ al quedarse de color negro, que es lo que se buscaba.

  $\therefore$ Por el principio de inducción, luego de llamar `COLOREA_AVL` desde un nodo v cuya altura es $2q-1$, con $q \in \mathbb{Z}^+$, entonces se obtiene un árbol rojo-negro enraizado en v y la *profundidad negra* para v es de q.

- Demostración para $v.h$ par: Por inducción natural sobre $v.h$ que se abreviará como h, con $h = 2q, q \in \mathbb{Z}^+$.

  Caso Base: $h = 2$, con $1 = 2(1)$ par. Esto implica que v es la raíz y que tiene dos hijos NIL (hojas) en un nivel anterior y como la altura de v es par, la de sus hijos es impar. Entonces sus hijos son coloreados de negro solamente. Si v se colorea de rojo, entonces la *profundidad negra* de v cumpliría con ser de 1; por otro lado, si se colorea de negro, cumpliría con ser de $1 + 1 = 2$.

[16] Como la demostración de abajo se vale de ésta, nada más es necesario tomarse la hipótesis de inducción luego de ver que se cumple el caso base; lo mismo se hará en la segunda demostración con respecto a ésta. Las dos demostraciones pueden verse como una sola que se hace en paralelo, pero decidí separar en dos por claridad de todos los subcasos que se presentan.

[17] Notemos que no pueden diferir sus alturas en más de una unidad porque es un árbol AVL.

Hipótesis de Inducción: Supongamos que para $h = k$, con $k = 2m$, $m \in \mathbb{Z}^+$ el llamar la función de `COLOREA_AVL` con parámetro del nodo v obtiene una *profundidad negra* para v de m si se colorea a v de rojo y de $m+1$ si se colorea de negro.
Paso Inductivo: Veamos que para $h = k+2 = 2m+2 = 2(m+1)$ se cumple que al mandar llamar la función de `COLOREA_AVL` desde v se obtiene para v una *profundidad negra* para v de $m+1$ si se colorea de rojo y de $m+2$ si se colorea de negro. Hay dos casos:

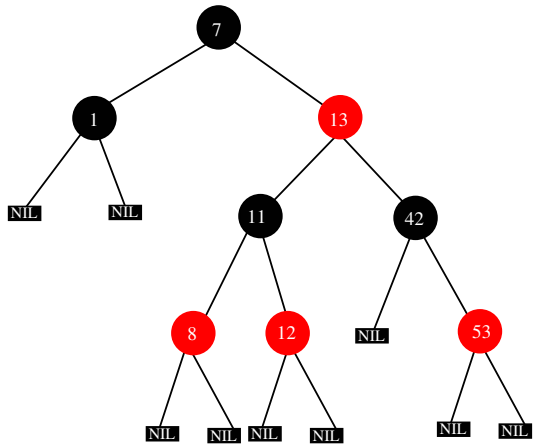
- ◇ Los dos descendientes de v tienen altura par $2(m+1)-1$, i.e., $v.i.h = v.d.h = 2(m+1)-1$. Entonces no se tienen que recolorear los hijos, pero al llamar la función sobre ellos por la hipótesis de inducción de la demostración anterior se tiene que sus *profundidades negras* son de $m+1$. Luego, si a v se le colorea de rojo, entonces no se modifica la *profundidad negra* que sería de $m+1$ como se buscaba. Por otro lado, si se mantiene de color negro, entonces incrementa en una unidad a $m+1$ como se quería ver.
- ◇ Si la altura de uno de los hijos es de $2(m+1)-1$ y la del otro es $2m$ —pueden diferir a lo más en una unidad por la propiedad de AVL—, supongamos sin pérdida de generalidad que $v.d.h = 2m$. En las llamadas recursivas de la función con ambos hijos, no se les recolorea dado que partimos de un nodo con altura par. Para el hijo izquierdo al tener una altura impar de $2(m+1)-1$, tenemos por lo ya visto que su *profundidad negra* sería de $m+1$. Por otro lado, por la hipótesis de inducción en el caso del hijo derecho que tiene la altura $2m$ y como su color es negro, la *profundidad negra* es de $m+1$. Entonces al colorear a v de rojo no se incrementa la *profundidad negra* para v y sería de $m+1$ como se quería ver; por el otro lado, si se pinta de negro, sería de $m+2$.

$\therefore$ Por el principio de inducción, luego de llamar `COLOREA_AVL` desde un nodo v cuya altura es $2q$, con $q \in \mathbb{Z}^+$, entonces se obtiene un árbol rojo-negro enraizado en v y la *profundidad negra* para v es de $q+1$ al colorearse de negro y de q al colorearse de rojo.

Por las dos demostraciones en paralelo presentadas, entonces se tiene que la *profundidad negra* para todo nodo en el árbol AVL transformado es la misma en cualquier camino hacia las hojas, por lo que se cumple la última propiedad de los árboles rojo-negros. Así, hay un algoritmo para transformar árboles AVL en árboles rojo negros.

$\therefore$ Los nodos de cualquier árbol AVL pueden colorearse de rojo y negro para obtener un árbol rojo-negro válido. □

b) Consideremos por contraejemplo el siguiente árbol rojo-negro:



En efecto es rojo-negro al ser un árbol binario de búsqueda en donde todos los nodos están coloreados de negro o de rojo. Además, cada nodo rojo cumple con tener dos hijos de color negro y la raíz es negra. Las hojas nulas cumplen con ser negras. Finalmente, se cumple con la condición de que todo camino de cualquier nodo a las hojas tiene la misma cantidad de nodos negros:

- Del nodo con elemento 7 a las hojas, hay dos nodos negros.
- Del nodo con elemento 1 a las hojas, hay un nodo negro.
- Del nodo con elemento 13 hay dos nodos negros.
- Del nodo con elemento 11 hay un nodo negro.
- Del nodo con elemento 42 hay un nodo negro.
- Del nodo con elemento 8 hay un nodo negro.
- Del nodo con elemento 12 hay un nodo negro.
- Del nodo con elemento 53 hay un nodo negro.

No obstante, el árbol no cumple con las propiedades de AVL puesto que la altura del subárbol izquierdo $H(T_i) = 1$, mientras que la altura del subárbol derecho $H(T_d) = 3$ y entonces:

$$|H(T_i) - H(T_d)| = |1 - 3| = |-2| = 2 \not\leq 1$$

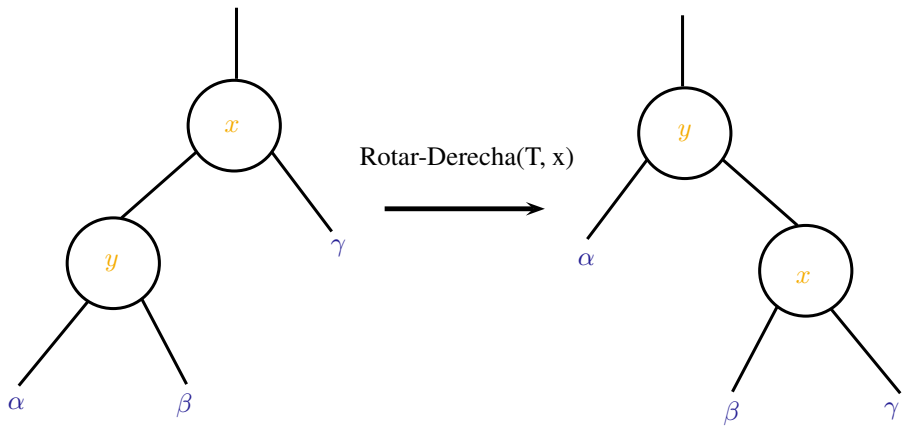
7. Se dice que un árbol binario de búsqueda T_1 puede ser *convertido-a-la-derecha* (*right-converted*) en un árbol binario de búsqueda T_2 si es posible obtener T_2 a partir de T_1 por medio de una serie de llamadas a la función **Rotar-Derecha**. Dar un ejemplo de dos árboles T_1 y T_2 tales que T_1 no pueda ser *convertido-a-la-derecha* en T_2. Después, mostrar que si un árbol T_1 puede ser *convertido-a-la-derecha* en un árbol T_2, entonces puede ser *convertido-a-la-derecha* usando $O(n^2)$ llamadas a la función **Rotar-Derecha**.

   Solución:

   Recordemos primeramente cómo se define la función **Rotar-Derecha** que toma un nodo de un árbol binario de búsqueda T sobre el que se lleva a cabo la rotación[18]:

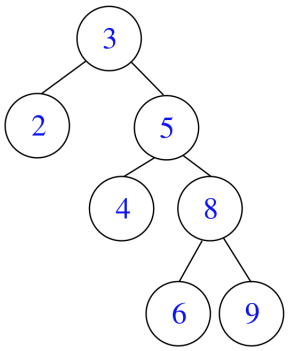
```
1  ROTAR−DERECHA(T,x)
2  y = x. left
3  x. left  = y. right
4  if  y. right  != T. nil
5    y. right .p = x
6  y.p = x.p
7  if  x.p == T.nil
8    T.root = y
9   elseif  x == x.p.left
10   x.p. left  = y
11  else  x.p. right  = y
12 y. right  = x
13 x.p = y
```

   Lo cual se ilustra para un árbol T binario de búsqueda como:

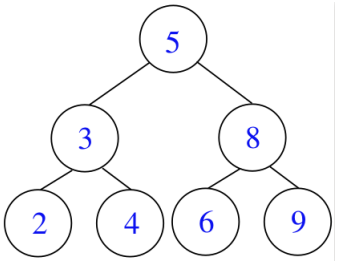


[18] La definimos análogamente a la función **LEFT-ROTATE(T.x)** de *CLRS, Introduction to Algorithms, Tercera ed., p. 313.*

Para la primera parte, consideremos a T_1 y T_2, respectivamente, como:



Por otro lado, consideremos a T_2 como:



No hay manera de llevar al nodo con el elemento 5 a la raíz de T_1 por medio de rotaciones a la derecha para que quede en la misma posición en que aparece el mismo elemento en T_2 dado que está en el subárbol derecho de T_1; las rotaciones a la derecha no llevan elementos del subárbol derecho al izquierdo.

Para la segunda parte, en primera instancia veamos que cualquier algoritmo que *convierta-a-la-derecha* un BST T_1 en otro BST T_2 usará un número de llamadas a la función de rotar a la derecha del orden cuadrático sobre el número de nodos en el árbol, n.

Definamos para un árbol binario de búsqueda T con n nodos:

$$L(v) = [\text{La cantidad total de referencias izquierdas que se siguen de la raíz de } T \text{ al nodo } v]$$

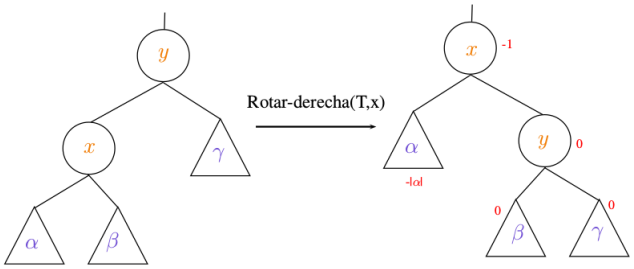
A partir de lo anterior, definimos:

$$L(T) = \sum_{v \in T} L(v)$$

Esto es, la suma sobre todos los nodos en el BST T de la cantidad de veces que nos vamos por la rama izquierda en el camino de la raíz a cada nodo considerado. Véase que el número total de descendientes izquierdos que puede tener un nodo es del orden lineal $O(n)$ (a lo más los $n-1$ nodos distintos a él) y entonces se sigue que para todo nodo $v \in T$, $0 \leq L(v) < n$. Ahora, considerando los n nodos en el árbol, cada uno con $L(v)$ del orden $O(n)$ se tiene que $L(T)$ es del orden cuadrático $O(n^2)$[19].

[19]El peor caso es cuando se tiene una cadena de nodos donde cada uno tiene un hijo izquierdo. De esta forma, para el último nodo se tiene que $v(x) = n-1$, para su padre sería $n-2$ y así sucesivamente y sabemos que $(n-1)+(n-1)+\cdots+0 = O(n^2)$.

Veamos ahora que cada rotación a la derecha (llamada a la función **Rotar-Derecha**) sobre un nodo x en el BST T decrementa el valor de $L(T)$. Consideremos un esquema genérico de la rotación a la derecha sobre tal nodo x con subárboles izquierdo y derecho α y β y padre y cuyo descendiente derecho es el subárbol γ.



Los números en color rojo indican cómo difiere $L(v)$ para los nodos y subárboles luego de la rotación. En el caso del nodo x, el total de referencias izquierdas desde la raíz a x se ve reducido en una unidad y en el caso de la suma de $L(w) \forall w \in \alpha$, entonces también se decrementa en una unidad; para todo ese subárbol izquierdo que tiene un total de $|\alpha|$ nodos, entonces se decrece en $|\alpha|$ unidades el valor de $L(\alpha)$. Finalmente, para el resto de los nodos no se ve alterado su valor de $L(v)$ pues no se incrementan o decrementan el número de referencias a hijos izquierdos que tienen que seguirse para llegar a ellos en un camino desde la raíz luego de la rotación a la derecha.

De esta forma, luego de una llamada a la función de rotación a la derecha, se tiene que $L(T)$ disminuye en un valor $d = -1 + (-|\alpha|) = -(1 + \alpha) \leq -1$. Como $L(T)$ es una cantidad no negativa que va decrementándose en al menos una unidad luego de cada rotación de la derecha e inicialmente es del orden cuadrático, entonces el número de veces que se manda llamar a lo más es del orden $O(n^2)$[20].

$\therefore$ Si un árbol binario de búsqueda T_1 puede ser *convertido-a-la-derecha* en un árbol binario de búsqueda T_2, puede convertirse usando $O(n^2)$ llamadas a la función de rotación a la derecha.

Un procedimiento para llevar a cabo dicha conversión en caso de ser posible sería llamar a la función **Rotar-Derecha** sobre el nodo x que debe estar presente en el subárbol izquierdo de T_1(si no es que ya está en la raíz de T_1) que es el mismo elemento que es la raíz del BST T_2; en caso de no ser así, entonces estaría en el subárbol derecho y ninguna secuencia de llamadas a la función podría llevarlo a la raíz contradiciendo la hipótesis de que se puede *convertir-a-la-derecha*.

Para llevar a x a la raíz de T_1 por los argumentos previos de que $L(x)$ es del orden $O(n)$, se tendrían que usar $O(n)$ llamadas a la función en cuestión. Luego, una vez en la raíz de T_1, se procede a llamar esta operación de conversión recursivamente para *convertir-a-la-derecha* el subárbol izquierdo de T_1 en el subárbol izquierdo de T_2 y el subárbol derecho de T_1 en el subárbol derecho de T_2.

La recurrencia sería entonces:

$$T(n) = T(i) + T(d) + n$$

Pues nos toma tiempo $O(n)$ y recursamos en ambos subárboles, donde $i + d = n - 1$, pues ya la raíz queda en la posición adecuada luego de la primera parte. Entonces podemos expresar la recurrencia como:

[20]Nos interesa el peor escenario que es cuando se decrementa en una unidad solamente luego de cada rotación a la derecha.

$$T(n) = T(n-1) + n$$

La solución de la recurrencia la podemos obtener con el método de sustitución.

Supongamos inductivamente así que $T(n) \leq c \cdot n^2, \forall n \geq n_0$, con $n_0, c \in \mathbb{N}^+$; claramente $T(1) = c \cdot 1^2$, con $c = 1 \in \mathbb{N}^+$. Entonces:

$$\begin{aligned}
T(n) &= T(n-1) + n && && (4)\\
&\leq c(n-1)^2 + n && \text{Hipótesis inductiva.} && (5)\\
&= c(n^2 - 2n + 1) + n && \text{Desarrollando el binomio al cuadrado.} && (6)\\
&= cn^2 - 2cn + c + n && \text{Distribuyendo.} && (7)\\
&\leq cn^2 - n(2c+1) + c && \text{Factorizando } n \text{ y ahora sustrayendo } n. && (8)\\
&\leq cn^2 - cn + c && \text{Tomando } c > 1 \text{ SPG, de donde } 2c - 1 > c. && (9)\\
&= cn^2 - c(n-1) && \text{Factorizando.} && (10)\\
&\leq cn^2 && \text{(n-1)} \geq 0 \text{ pues } n \geq 1 \text{ y como } c \geq 1, \text{ entonces } -c(n-1) \leq 0. && (11)
\end{aligned}$$

Entonces $\exists c \in \mathbb{N}^+ \ni T(n) \leq cn^2$, así que por definición $T(n) \in O(n^2)$.
$\therefore T(n) \in O(n^2)$.

Quizás una forma más sencilla de resolver recurrencias de este estilo es a través del proceso iterativo que no requiere aventurarse en conjeturar una solución como antes. En este caso:

$$\begin{aligned}
T(n) &= T(n-1) + n && && (12)\\
&= T(n-2) + (n-1) + n && \text{Desarrollando } T(n-1) && (13)\\
&= T(n-3) + (n-2) + (n-1) + n && \text{Desarrollando } T(n-2) && (14)\\
&\vdots && && (15)\\
&= T(1) + (2 + 3 + \cdots + n) && \text{Luego de varios pasos.} && (16)\\
&= T(0) + (1 + 2 + 3 + \cdots + n) && \text{Desarrollando } T(1). && (17)\\
&= T(0) + \frac{n(n-1)}{2} && \text{Suma de los primeros } n \text{ naturales.} && (18)\\
&= 1 + \frac{n(n-1)}{2} && \text{Para } T(0) = 1, \text{ pues no hay nodos y terminamos.} && (19)\\
&= 1 + \frac{n^2 - n}{2} && \text{Distribuyendo} && (20)\\
&= \frac{1}{2}(2 + n^2 - n) && \text{Reacomodando.} && (21)
\end{aligned}$$

De donde $T(n) \in O(n^2)$, pues $\exists c = 1, n_0 = 1 \in \mathbb{N}^+$ y se tiene que $\frac{1}{2}(2 + n^2 - n) \leq n^2 \forall n \geq 1$ como podemos ver por inducción natural:

Caso base: Para $n = 1$ se cumple que $\frac{1}{2}(2 + 1^2 - 1) = \frac{1}{2}(2 + 1 - 1) = \frac{1}{2}2 = 1 \leq 1^2 = 1$.

Hipótesis de Inducción: Supóganse que para $n = k$ se cumple que $\frac{1}{2}(2 + k^2 - k) \leq k^2$.

Paso Inductivo: Veamos que se tiene que para $n = k+1$ es cierto que $\frac{1}{2}(2+(k+1)^2-(k+1)) \leq (k+1)^2$.

$$\begin{aligned}
\frac{1}{2}(2 + (k+1)^2 - (k+1)) &= \frac{1}{2}(2 + k^2 + 2k + 1 - k - 1) && \text{Desarrollando.} && (22)\\
&= \frac{1}{2}(2 + k^2 + 1) && \text{Simplificando} && (23)\\
&\leq k^2 && \text{Hipótesis de Inducción} && (24)\\
&\leq k^2 + 2k + 1 && \text{Sumando } 2k + 1 \geq 0. && (25)\\
&\leq (k+1)^2 && \text{Expresando el producto notable.} && (26)
\end{aligned}$$

$\therefore$ Por transitividad de la desigualdad $\frac{1}{2}(2 + (k+1)^2 - (k+1)) \leq (k+1)^2$.

$\therefore$ Por el principio de inducción se sigue que $T(n) \in O(n^2)$.

8. Muestra que los métodos inserta y borra de los árboles B vistos en clase usan proporcionalmente $Blog_B(n)$ de memoria. ¿Existe manera de mejorar a $O(B + log_B(n))$?. Explique.

   Solución

   Recordemos que los árboles B fueron diseñados para trabajar adecuadamente en discos y demás sistemas de dispositivos de almacenamiento secundario(más económicos que la memoria principal) [21] y dado que minimizan el número de operaciones de lectura-escritura en disco tienen varias aplicaciones en bases de datos. Debido a que dichos accesos al almacenamiento secundario son lentos —generalmente más que el examinar la información traída a RAM—, el análisis de este tipo de árboles en cuanto a tiempo de ejecución suele verse desde dos ámbitos: número de accesos a disco y por otra parte del tiempo de ejecución del procesador. Ahora, como estos accesos son costosos, se busca traer la mayor cantidad posible a la memoria principal en cada acceso para llevar a cabo las diversas operaciones.

   Recordemos que por lo visto en clase, la principal diferencia entre este tipo de árboles —que veremos son balanceados— con respecto a otras familias de árboles balanceados que conocíamos es su factor de ramificación, que puede ser muy grande. Entonces, los nodos del árbol son en sí bloques que tienen los siguientes campos:

   Para un nodo x (un bloque):

   - El número de elementos almacenados en el bloque x, que denotaremos por $n[x]$.
   - Los $n[x]$ elementos propiamente, almacenados en orden creciente.
   - Un valor de verdad que nos dice si x es una hoja o no.
   - Un total de $n[x] + 1$ apuntadores a los hijos de los elementos de x.

   Y se tienen propiedades:

   - Como consecuencia de lo anterior, el i-ésimo elemento de un bloque x (nodo) particiona el rango de elementos que se almacenan en cada subárbol.
   - Todas las hojas tienen la misma profundidad, que es la altura del árbol, h.
   - Cada nodo deberá tener un mínimo y un máximo de elementos y descendientes a partir de un grado mínimo fijo B, con $B \geq 2$:
     - La cota mínima establece que todo nodo distinto a la raíz deberá tener al menos $B - 1$ elementos y por ende B hijos como mínimo.
     - Cada nodo puede tener a lo sumo $2B - 1$ elementos y en consecuencia a lo más $2B$ hijos.

   Antes de proceder a ver que la complejidad en espacio para las operaciones de inserciones y eliminaciones son del orden $O(Blog_B n)$, probemos que la altura de un árbol B es del orden $O(log_B n)$[22]:

   Teorema: Para todo árbol B T con n nodos, $n \geq 1$ y con grado mínimo $B \geq 2$, se tiene que la altura h de T es:

$$h \leq log_B(\frac{n+1}{2}) \in O(log_B n)$$

   Como $n \geq 1$, entonces la raíz deberá tener al menos un elemento; para el resto de los nodos (bloques), mínimamente deberán tener $B - 1$ elementos por la restricción antes mencionada. Se

[21] *CLRS, Introduction to Algorithms, Tercera ed., pp. 484-504.*
[22] La demostración puede ser consultada en el *CLRS* en la página 489.

sigue entonces que tendrá al menos dos nodos en el nivel de profundidad 1, al menos $2B$ nodos en el nivel de profundidad 2 (pues cada uno de los del anterior deberá tener al menos B hijos), luego mínimamente habrá $2B^2$ en el tercer nivel y así sucesivamente. De esta forma:

$$
\begin{aligned}
n &\geq \underbrace{1}_{\text{raíz}} + \underbrace{(B-1)}_{\text{Cada uno de esos nodos al menos tiene B-1 elementos}} + \sum_{i=1}^{h} 2^{B-1} \\
&= 1 + 2(B-1)\sum_{i=1}^{h} B^{i-1} && \text{Sacando el 2 de la suma} \\
&= 1 + 2(B-1)\frac{B^h - 1}{B-1} && \text{Geométrica} \\
&= 1 + 2(B^h - 1) && \frac{B-1}{B-1} = 1 \\
&= 1 + 2B^h - 2 && \text{Desarrollando} \\
&= 2B^h - 1 && \text{aritmética}
\end{aligned}
$$

Entonces, por transitividad se tiene que:

$$n \geq 2B^h - 1$$

De donde:

$$
\begin{aligned}
n + 1 &\geq 2B^h && \text{Sumando 1 (positivo) de ambos lados de la desigualdad} \\
\frac{n+1}{2} &\geq B^h && \text{Dividiendo entre 2 (positivo) de ambos lados}
\end{aligned}
$$

Entonces, tomando el logaritmo de ambos lados, que es una función creciente, obtenemos:

$$h \leq log_B \frac{n+1}{2}$$

$\therefore$ La altura h del árbol B es del orden $O(log_B n)$ $\square$

Con esto en mente veamos:

a) La complejidad en espacio para el algoritmo de inserción es $O(Blog_B n)$ y cómo puede mejorarse a $O(B + log_B n)$.

b) La complejidad en espacio para el algoritmo de eliminación es $O(Blog_B n)$ y cómo puede mejorarse a $O(B + log_B n)$.

Veamos ambas partes para un árbol-B T de n nodos (bloques) y con grado mínimo $B \geq 2$:

a) Para el algoritmo de inserción, recordemos que se hace recorre hacia abajo en el árbol T, en donde en cada nodo (bloque) se hace una búsqueda logarítmica para encontrar la posición que le corresponde al nuevo elemento; sin embargo, puede darse el caso de que al insertarlo, entonces el árbol resultante ya no satisfaciera la propiedad del máximo número de elementos en un bloque puesto que podría ya encontrarse lleno con $2B-1$ elementos. Es por ésto que se procede

a insertar el nodo en un nodo hoja, pero en caso de que este nuevamente esté lleno con $2B-1$ elementos, se procede a partir el nodo (bloque) con la función auxiliar de `SPLITTING` que lo que hace partir al nodo y se obtienen dos nodos, cada uno con $B-1$ elementos. Luego, el elemento que es la mediana del nodo original previo a la partición sube al padre del nodo original para poder identificar el punto de división de los dos nodos resultantes. No obstante, puede ocurrir de nueva cuenta que dicho nodo padre ya tuviera $2B-1$ elementos, así que se tendría que repetir la operación previa luego de subir la mediana y esto podría continuarse hasta la raíz del árbol T.

Generalmente se sugiera la alternativa de no tener que esperar a ver que se tiene que partir un nodo que ya estaba lleno para poder insertar, sino que se baja desde la raíz del árbol a una hoja en donde será insertado el nuevo elemento; en cada paso hacia abajo, si el nodo encontrado ya está lleno, i.e., tiene $2B-1$ elementos, se procede a partirlo para asegurarnos de que al llegar a la hoja, podremos insertarlo sin ningún inconveniente dado que ya habremos partido todos los nodos llenos en el camino y habrá espacio disponible en el nodo.

En la primera forma de inserción, que fue la comentada en clase, se llevan a cabo llamadas recursivas a la función de inserción por cada nivel hasta llegar a un nodo hoja, después de lo cual se tiene que llevar a cabo el efecto de propagación de partición en las medianas de los nodos hasta potencialmente la raíz; justamente por eso es que se tienen que tener las referencias a los padres que se logra con las llamadas recursiva al tener activos los marcos en la pila de ejecución. Dado que la altura del árbol T es del orden $O(log_B n)$ y cada nodo (bloque) por el que descendemos tiene un número de elementos del orden $O(B)$ (entre $B-1$ y $2B-1$), entonces la profundidad del árbol de recursión será de tamaño $O(log_B n)$ y en cada llamada almacenamos espacio en memoria principal del orden $O(B)$[23]. Se sigue que la operación de inserción tiene una complejidad en espacio del orden $O(Blog_B n)$.

La segunda alternativa para inserción posee la ventaja de que al no tener que propagar el efecto de las particiones hasta la raíz porque ya se hicieron de antemano cada vez que se encontraba un nodo ya lleno en el recorrido hace abajo, entonces no es necesario almacenar la información sobre los padres de los nodos por los que vamos descendiendo y puede ser más fácilmente convertido a un algoritmo iterativo al ir actualizando una referencia al bloque al que se mueve en cada iteración. De esta manera, si se inserta en un árbol T con un solo nodo que ya es una hoja, bastaría con hacer el recorrido lineal o bien la búsqueda logarítimica sobre el nodo (bloque) aprovechando que los elementos ya están ordenados y para ésto en memoria solamente tendría que tenerse dicho nodo que es del orden $O(B)$. Por otro lado, suponiendo que $n \geq 1$, entonces hay que descender un número logarítmico $O(log_B n)$ de niveles hasta encontrar la raíz donde deberá ser insertado el nuevo elemento; en cada paso en memoria tendremos solamente el bloque actual, y después de encontrar el apuntador al hijo de un elemento en el bloque que debemos seguir, el bloque actual ya no permanece en memoria principal. De esta forma, el espacio usado es del orden $O(B)$ de igual forma. Como se tiene que $B \leq B + log_B n$, entonces la complejidad en memoria es esta modificación sería del orden $O(B + log_B n)$.

b) Recordemos brevemente el algoritmo de eliminación estudiando en clase y analicemos luego su complejidad en espacio para ver que en efecto es del orden $O(log_B n)$. Hay varios casos que deben verse al eliminar un elemento k de un árbol-B T cuyo grado mínimo es $B \geq 2$:

   - En caso de que k esté en un nodo (bloque) hoja x y se tenga que $n[x] > B-1$ (tiene estrictamente más elementos que el mínimo permitido), entonces simplemente se elimina a k del bloque x dado que no se violará la propiedad al quitarlo.

[23] La función de `SPLITTING` tiene una complejidad en memoria del orden $O(B)$, pues únicamente se crea un nuevo nodo con un número de elementos del orden $O(B)$.

- En caso de que k está en un nodo (bloque) hoja x y se tenga que $n[x] = B - 1$ (mínimo permitido de elementos), entonces se presentan dos subcasos:
  - Si el padre del nodo k tiene otro hijo distinto de k (hermano de x) que llamamos y con $n[y] \geq B$ (no tiene el mínimo), entonces se toma el elemento en el extremo apropiado de y dependiendo del valor a eliminar k; dicho elemento es transferido al nodo del padre y el nodo apropiado del padre es transferido del padre a x para que siga cumpliendo con el menor número de elementos al borrar a k.
  - Si todos los hermanos de k tienen también el mínimo número de elementos $B - 1$, entonces es necesario mezclar x con uno de sus hermanos al bajar la mediana del nodo padre para completar la mezcla y se procede a eliminar. Como se mezclaron dos nodos con $B - 1$ elementos y además se agregó un elemento del nodo padre, ahora se tienen $2(B - 1) + 1 = 2B - 1$ elementos, así que al borrar a k, se tienen $2B - 2$ que cumple con las restricciones.
- Si el elemento k a eliminar aparece en un nodo (bloque) interno (no una hoja), entonces hay tres subcasos:
  - Si el hijo izquierdo y del elemento k tiene al menor B elementos, entonces el mayor valor de dicho hijo (predecesor de k, k') es transferido al padre para reemplazar a k para luego eliminarlo sin problemas. Para ésto se tiene que llamar recursivamente a la función de eliminación con dicho valor k' antes de poder reemplazar a k por k'.
  - Análogamente si el hijo derecho z del elemento k tiene al menor B elementos, entonces su valor más pequeño (sucesor de k, k') puede ser transferido al padre para sustituir a k que luego será eliminado sin problemas. Para ésto se tiene que llamar recursivamente a la función de eliminación con dicho valor k' antes de poder reemplazar a k por k'.
  - Si ninguno de los hijos de k tienen al menos B elementos (tienen entonces el mínimo de $B - 1$), entonces los hijos deben mezclarse, se borra a k de su nodo y el nodo derecho del elemento que estaba a la izquierdo de k tendrá como hijo derecho al resultado de la mezcla; en el caso del elemento que estaba a la derecho de k, su hijo izquierdo será el resultado de la mezcla.

Por el tercer caso, puede darse el caso en que el elemento a eliminar sea encuentre en la raíz de T y se presente el primero o segundo subcaso, en donde se llama recursivamente con un nodo de un nivel de profundidad mayor y así podría seguirse este proceso hasta llegar al caso en que se tiene que eliminar a un nodo de una hoja. Por tanto, el número de llamadas recursivas a la función de eliminación hace que la altura del árbol de recursión sea de altura igual a la altura del árbol-B T, que es del orden $O(log_B n)$; esto es necesario bajo esta implementación ya que las referencias a los nodos padres son necesarias para poder hacer las transferencias de los sucesores o predecesores de acuerdo con subcasos uno y dos del tercer caso. Como en cada llamada se mantiene un nodo (bloque) de tamaño B en memoria principal, se tiene que la complejidad en espacio con este implementación sería del orden $O(Blog_B n)$.

Es posible mejorar a $O(B + log_B n)$ de manera parecida a como se hizo en el caso de las inserciones: no tendremos que regresar a niveles por los que ya vinimos si cada vez que se llame a la función recursiva sobre un nodo aseguramos la condición fuerte de que dicho nodo tiene como mínimo B elementos en lugar del mínimo permitido de $B - 1$. Así, será necesario en ocasiones llevar un elemento del padre a un nodo hijo antes de poder aplicar el proceso de eliminación sobre el hijo y de esta forma eliminar un nodo en una sola pasada hacia abajo en el árbol. Luego, al convertir la función más fácilmente a un proceso iterativo, sería suficiente con almacenar en memoria principal a lo más tres bloques a la vez[24], lo que es del orden

[24]Se debe al caso en que es necesario encontrar a k a eliminar en un bloque y luego checar a los hijos para ver si se deberán mezclar (2c) o bien buscar al predecesor o sucesor en ellos (2a y 2b).

$O(B) = O(log_B n + B)$. El algoritmo es similar al anterior en su versión recursiva previa a la transformación iterativa:

1) Si el elemento a eliminar k está en un nodo x que es una hoja, entonces se elimina k de x.
2) Si el elemento a eliminar k está en un nodo interno:
   a) Si el hijo y que precede a k en el nodo x tiene al menos B elementos, entonces se encuentra al predecesor k' de k en y, se reemplaza a k por k' en x y se recursa eliminando a k' en y.
   b) Si y tiene menos de B elementos, entonces se examina al hijo z que sigue al elemento k en el nodo x. Si z tiene al menos B elementos, entonces se encuentra al sucesor k' de k en z, recursivamente se elimina a k' de z y se sustituye a k por k' en x.
   c) Si ambos y y z tienen $B - 1$ elementos (el mínimo posible), entonces se mezclan k y todo el nodo z en el nodo y para que x ya no tenga al elemento k ni el apuntador a z y ahora y tendrá $B - 1 + 1 + B - 1 = 2(B - 1) + 1 = 2B - 1$ elementos. Después de recursa eliminando k en y.
3) Si el elemento k no está presente en un nodo interno x, entonces se determina la raíz del subárbol-B donde deberá estar en caso se encontrarse en T. En caso de que este subárbol $x.c_i$ determinado tenga solamente $B - 1$ elementos, entonces se ejecutan los casos 2a) o 2b) para garantizar que descendemos a un nodo que tiene al menos B elementos como buscamos y recursamos en el hijo.

A pesar de que ambos algoritmos son similares, el hecho de no tener que regresar por la condición fuerte que aseguramos permite una transformación más sencilla al algoritmo iterativo que permite mejorar la complejidad en tiempo al ir llevando un apuntador del nodo en que nos encontramos en cada paso, y luego de buscar al elemento a eliminar en este nodo tener solamente en memoria:

- El nodo actual y sus nodos izquierdo y y derecho z para hacer el intercambio conveniente.
- El siguiente nodo en donde debemos continuar con el proceso de acuerdo con el tercer caso en caso de que no se encuentre en el nodo x, así que en este punto se tendría solamente al descendiente $x.c_i$ en memoria.

Y para el caso de la mezcla de 2c), una vez mezclados, en lugar de recursar sobre y para eliminar a k, se tiene ahora en memoria en el proceso iterativo a y.

$\therefore$ Las complejidades en espacio para las operaciones de inserción y eliminación vistas en clase son del orden $O(Blog_B n)$ y hay formas de mejorarlas a $O(B + log_B n)$.