Tarea Examen 1

Uriel Balderas Aguilar

October 4, 2024

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 binarios, son balanceadas.

   (a) Cada nodo del árbol es una hoja o tiene dos hijos.
   Es falso, y para verlo, tomemos un árbol-lista que se extiende de un lado, y finalmente hacemos valer la propiedad indicada (cada nodo es hoja, o tendrá 2 hijos).
   Así, notamos que la altura del árbol se expresa de la forma $\frac{n}{2} - 1$, lo cual es una función $O(n)$, cuya complejidad es directamente superior a la función dada en la definición $O(logn)$

   (b) Existe una constante c tal que, para cada nodo en el árbol, las alturas de sus sub-árboles difieren a lo más c.
   Es verdadero, primero porque se tienen ejemplares de esta familia que cumplen con ser balanceados, como los árboles AVL y los R/N, pero no se limita a éstos.
   Primero, supongamos que NO se cumple que el árbol sea balanceado para una constante c, tal que c equivale a la cantidad de nodos en el árbol en dado momento, es decir, se tiene un "árbol lista" que no es balanceado pues la mayor diferencia aceptada entre alturas de nodos es justamente lineal respecto esto.

   Ahora, tomando el mismo árbol, insertemos una gran cantidad de nodos, digamos 2^c, siguiendo la estrategia de rotaciones, en el momento que se supere la tolerancia de diferencia de alturas constante, notaremos que con cada inserción, en el peor caso se requiere hacer una cadena de rotaciones, de modo que en todo

momento se preserve la diferencia máxima de alturas toleradas c, y notaremos que muy a pesar del desequilibrio inicial, conforme la estructura continúa siendo poblada, se logra, inserción a inserción, la altura que en un inicio era expresable de forma lineal, se va volviendo de forma logarítmica, y ya que se logró dicho comportamiento con una cantidad finita de nodos, se puede aseverar por inducción que será válido para un valor n incluso superior.
Por lo anterior, se considera demostrado.

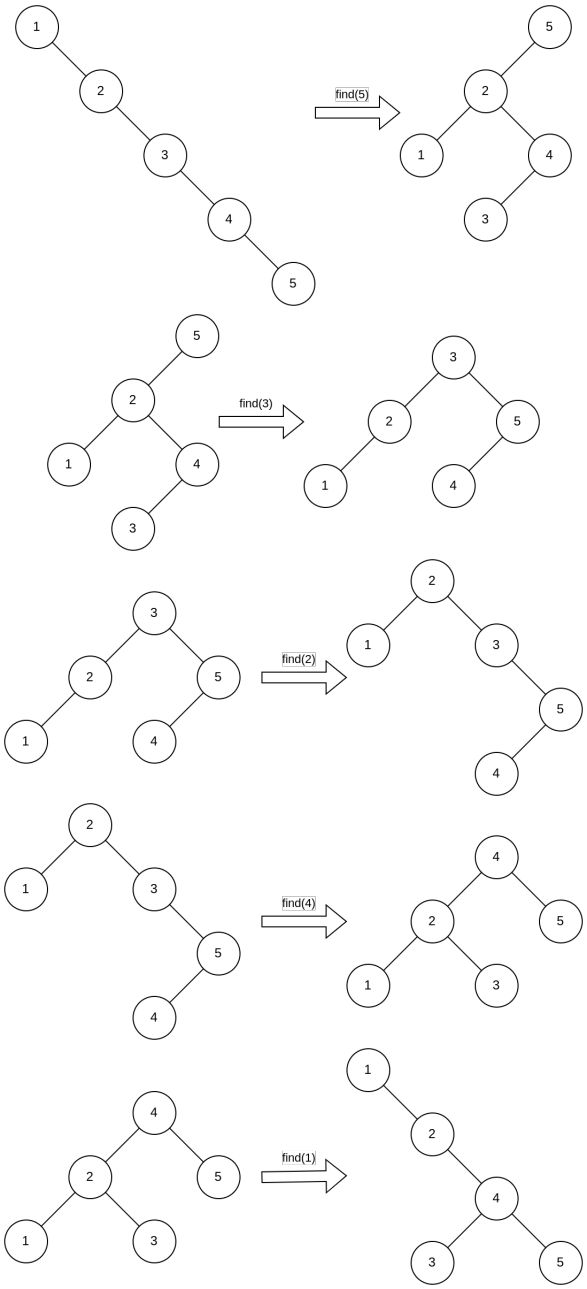
(c) La profundidad promedio de un nodo es $O(log(n))$.
Es falso, y aunque de inicio es contra-intuitivo, puede mostrarse usando como base el ya muy mencionado "árbol-lista".
A partir de éste árbol, se razona que a partir del comportamiento de la función promedio, se puede tener de un lado un valor monstruosamente alto en un lado del árbol, y del otro agregar poco a poco "contrapesos", es decir, se desea maximizar la profundidad de la hoja del árbol lista original, que asumiremos es el subarbol izquierdo de la raíz, y los "contrapesos" indicados, se agregarán en el subarbol derecho a la raíz, con preferencia a agregar hojas con la menor profundidad disponible, de este modo, se presenta un sub-arbol con comportamiento de altura más bien lineal, mientras que el otro sub-arbol realiza inserciones que apuntan a poblar primero el arbol "a lo ancho" que en lo alto

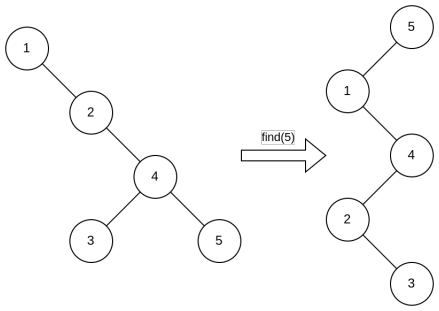
2. Muestra que dado un conjunto T de n nodos $x_1, x_2, \ldots, x_n$ con valores y prioridades distintas, el **árbol treap** asociado a T es único.
Se procede por contradicción, por lo que suponemos es falso, es decir, existen al menos 2 formas distintas de representar el mismo conjunto de datos.
Para esto, los construiremos asumiendo un orden de inserción de nodos diferente.
En su primera fase, podemos lograr únicamente con el orden de inserción que ambos árboles sean distintos, la ejecutar únicamente la sección de inserción BST.
Sin embargo, hay que reconocer que tendrán el mismo "orden plano" (al realizar un recorrido pos-order, se habrán encontrado a los elementos ordenados del mismo modo), debido a la característica de orden implícito propio de los BST.
Luego, llega la segunda fase de inserción, que son las rotaciones.

En el planteamiento se menciona que todos los nodos tienen prioridades distintas entre sí, y ya que las prioridades se manejan con el régimen de HeapMax, las prioridades de mayor valor han de quedar por encima del resto.
Así, sin perder generalidad, conocemos que el nodo con prioridad máxima ha de quedar a la raíz del árbol, así sabemos que ambos árboles, como mínimo, poseerán la misma raíz, y además, al conservar la lógica de orden implícito de BST, todos los nodos con valor menor seguirán en su sub-arbol izquierdo, y todos los nodos con valor superior seguirán en su sub-arbol derecho.
De ésto, reconocemos que se puede seguir un razonamiento recursivo, considerando cada sub-arbol como un árbol en sí mismo, con lo que al concluír con las rotaciones, se tendrá que todos los nodos se encontrarán en las mismas posiciones, gracias a las propiedad en conjunto de las prioridades HeapMax, y los valores ordenados mediante BST.
Es justamente durante esta segunda fase, que se realiza mediante rotaciones, lo que asegura la preservación del orden BST, mientras se hace el acomodo correspondiente de las prioridades a lo largo de los nodos.

3. Considera el treap T después de insertar x, con el algoritmo visto en clase.
   Sea C la longitud del camino derecho del subárbol izquierdo de x.
   Sea D la longitud del camino izquierdo del subárbol derecho de x.
   Demuestre que el número total de rotaciones que se realizaron durante la inserción de x es igual a $C + D$.

4. Describe una secuencia de accesos a un **árbol splay** T de n nodos, con $5 \leq n$ impar, que resulte 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.


1
2
3
4
5
find(5)
5
2
1
4
3
5
2
1
4
3
find(3)
3
2
5
1
4
3
2
5
1
4
find(2)
2
1
3
5
4
2
1
3
5
4
find(4)
4
2
5
1
3
4
2
5
1
3
find(1)
1
2
4
3
5



5. Muestra cómo transformar una **skip-list** L en un árbol binario de búsqueda $T(L)$.
   Justifica por qué buscar en $T(L)$ no es más rapido que en L.

6. Demuestra o da un contraejemplo:

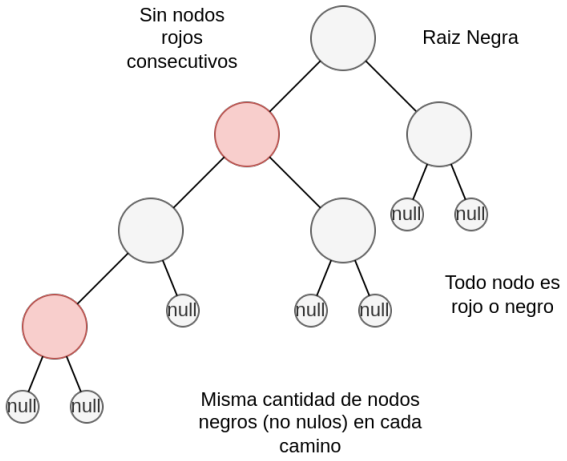
   (a) Los nodos de cualquier **árbol AVL** pueden colorearse de rojo y negro para obtener un árbol rojo-negro válido.
   Esto es cierto, y para demostrarlo, primero hay que mencionar que ambos son arboles BST, por lo que ya tienen el mismo "orden implicito".
   Además, verificando las propiedades de arboles AVL y rojinegros, notamos que la de AVL es mas restrictiva respecto altura; puesto que la mayor diferencia de alturas entre 2 sub-arboles hijos de un mismo nodo, será a lo mas 1. Así, propagandose ésta propiedad por todos los nodos, se tiene que tambien se cumple para la raíz. Entonces, ya que conocemos que en todo momento hay, a lo más 1 nivel de diferencia entre las hojas, procedemos a analizar las propiedades de Rojinegros:

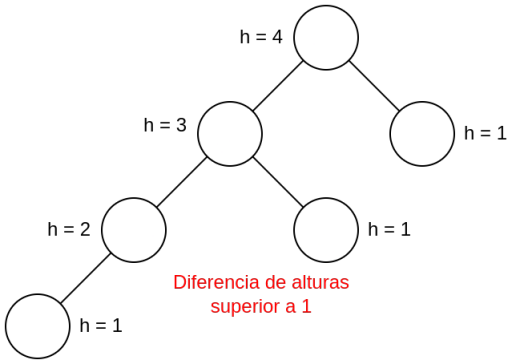
   - Todo nodo es rojo o negro
   - La raiz es negra
   - Todas las hojas (null) son negras
   - Todo nodo rojo ha de tener dos hijos negros/No hay dos nodos rojos contiguos (padre-hijo)
   - Cada camino desde un dado nodo a sus hojas contiene la misma cantidad de nodos negros.

   Ahora que las conocemos, primero dire que todos los nodos del arbol AVL seran negros, y con esto casi cumplimos las restricciones,

excepto por la ultima, debido a que la diferencia de altura puede hacer que haya un nodo negro mas que en el del vecino en algun punto.
Por esto, si el arbol tiene un factor-balance 0, al haber pintado de negro los nodos acabamos; pero en otro caso (factor-balance 1 o -1), la solución que propongo es que todos los nodos en el nivel mas profundo sean re-coloreados de rojo, y además los hijos izquierdo y derecho inmediatos al nodo raíz también sean re-coloreados de rojo.

(b) Cualquier **árbol rojo-negro** satisface las propiedades de **árbol AVL**.
Esto es falso, debido a que el arbol AVL tiene una "tolerancia" de diferencia de alturas en sus sub-arboles claramente marcada como máximo 1, mientras que el arbol rojinegro fácilmente puede sobrepasar esta limitación, un ejemplo de lo mencionado es el siguiente:





7. Decimos que un **BST** T_1 puede ser convertido por la derecha a un **BST** T_2 si es posible obtener T_2 como resultado de ejecutar una sucesión de operaciones ***RotacionDerecha*** sobre un arbol T_1.
   Proporciona un ejemplo de arboles T_1 y T_2 tal que T_1 *no* pueda ser convertido por la derecha en T_2.
   Además, demuestra que si un arbol T_1 puede ser convertido en T_2, se hará en $O(n^2)$ ejecuciones ***RotacionDerecha***.

8. Supongase un **arbol 2-4** T, con n_l hojas y n_i nodos internos.

   (a) ¿Cuál es el mínimo valor de n_i como función de n_l?
   Ya que se estan considerando la menor cantidad de nodos n_i, se asume que, si no todos, la mayoría dominante de estos serán nodos4 (ya que de no ser así, se requieren más nodos como padres de los nodos n_l, lo que resultaría en un aumento mas notable de nodos), entonces, suponiento que se tienen $4k + c$ nodos n_l, con $0 \leq c < 4$, se tendrán a lo más $k + 1$ nodos padres de n_l, luego, en nuestro afan de ahorrar nodos, supondré que se siguen usando nodos4 en n_i, así, notamos que se van reduciendo los nodos padre en razon de 4, lo cual nos recomienda el comportamiento logaritmico observado en los arboles binarios ya estudiados previamente.

   Entonces, si se tienen 64 hojas, hay 16 padres de estos, que a su vez tienen 4 padres, que a su vez tienen 1 padre (raiz), lo que corresponde a que, si se tienen 64 n_l, se tienen $16 + 4 + 1$ nodos n_i.
   Entonces $4^3 n_l$ requieren $4^2 + 4^1 + 4^0$ nodos n_i

Así, si tengo $4k + c = nl \rightarrow \Sigma_{x=0} \log_4(4k + c) - 14^x = n_i$
Con lo que tengo que $n_i = \frac{4^{log4(4k+c)-1+1}-1}{4-1} = \frac{(4k+c)-1}{3}$
Ahora, compruebo que la funcion corresponda a lo expresado originalmente:

$$n_l = 64; k = 16, c = 0$$

$$n_i = \frac{(4 * 16 + 0) - 1}{3} = \frac{64 - 1}{3}$$

$$n_i = \frac{63}{3} = 21$$

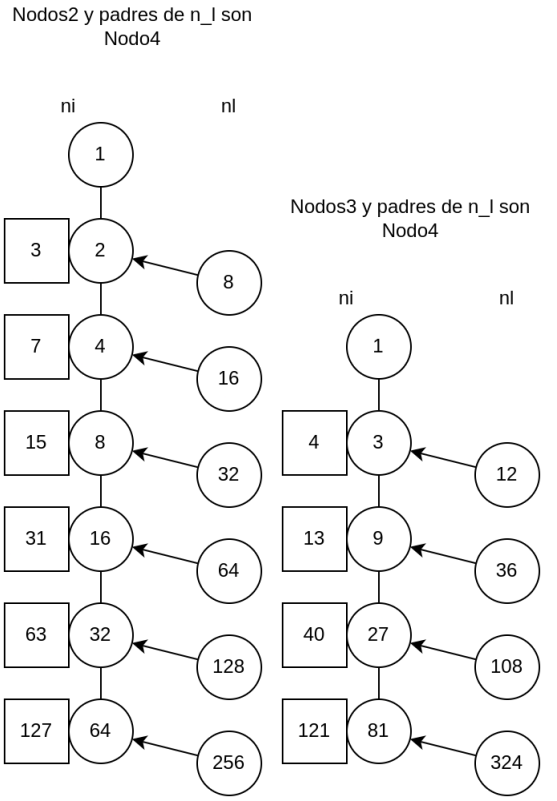
Lo que concuerda con el valor esperado, por lo que concluyo que:

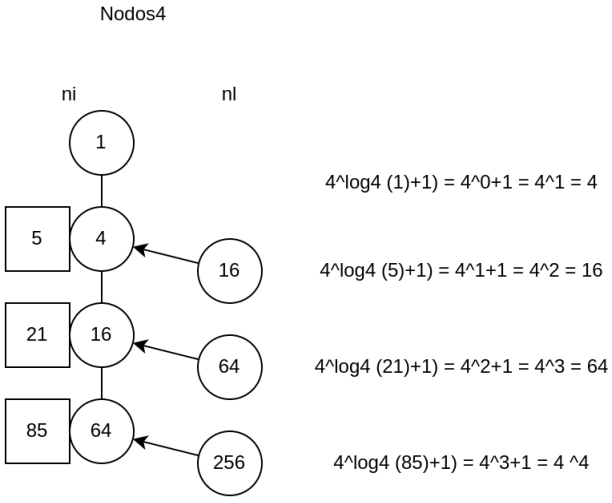
$$n_i = \frac{(4k + c) - 1}{3}$$

con $n_l = 4k + c$ y $0 \leq c < 4$

(b) ¿Cuál es el máximo valor de n_i como funcion de n_l?

Ahora queremos maximizar el valor de n_l en base n_i... para esto, asumo que los padres de los nodos n_l siempre son nodos4, sin embargo tengo que analizar como considerar el resto de nodos n_i.
En este analisis visual, el circulo izquierdo corresponde a la cantidad de nodos ni en ese nivel, mientras que el circulo derecho representa la cantidad de nodos n_l total que se obtendria de maximizar en ese punto los nodos n_l, finalmente, el cuadrado representa la cantidad total de nodos ni acumulados en el arbol, de modo que se puede notar la cantidad de nodos n_i y cuantos nodos n_l producen con el razonamiento dado.





Del previo analisis, notamos que tomando que todos los nodos n_i sean nodos4, maximizamos la cantidad de nodos n_l, ademas que

requerimos de una proporcion menor de nodos n_i que con otras configuraciones.
Entonces, notamos que los nodos nl aumentan en forma de potencias de 4, mientras que los nodos n_i aumentan por capas, es decir, que la cantidad de nodos n_l depende del nivel alcanzado por los nodos n_i. Así, conocemos que la cantidad de nodos n_l será de la forma $4^{floor(log4(n_i))+1}$.

(c) Si T' es un **rojinegro** que representa a T, ¿Cuántos nodos rojos tiene T'?

Suponiendo que conocemos la cantidad de nodos que son nodo2 , nodo3 y nodo4, podemos usar una guía de conversion similar a la mencionada en clase, donde a cada nodo2, le corresponde convertirse en nodo negro, a cada nodo4, su valor central será negro, mientras que sus laterales seran rojos, y finalmente, los nodo3, serán un nodo rojo y uno negro, por lo que se tendrán:

$$2(n_4) + 1(n_3)$$

Ideas inspiradas de:
`https://stackoverflow.com/questions/35955246/converting-a-2-3-4-tree-int`
`https://stackoverflow.com/questions/70251698/converting-2-3-4-to-red-bla`
`https://faculty.cs.niu.edu/~freedman/340/340notes/340redblk.htm`

9. Diseña e implementa un **Treap** que incluya al menos las siguientes operaciones:

- *insert(V,P)*: Agrega el valor V al **Treap** con su respectiva prioridad **P**.
- *delete(V)*: Elimina la primera ocurrencia de V en el **Treap**, si existe.
- *get(**V**)*: Obtiene el nodo con valor V que se encuentra en el **Treap**, si existe
- *get(**i**)*: Obtiene el nodo cuyo valor sea el i-ésimo más grande en el **Treap**, si i está en el rango de elementos.
- *peek()*: Regresa el nodo hasta el tope del **Treap**.
- *pop()*: Elimina y regresa el primer elemento al tope del **Treap**.

https://github.com/Xpartiel/EstructurasDatosAvanzadas