

Tarea 2

1) a) el algoritmo sera el siguiente , primero recorremos la lista y cada elemento lo metemos en un arbol binario de busqueda balanceado (si no es balanceado , esto deja de funcionar) , entonces metemos cada elemento al arbol , pero cada nodo del arbol tendra una lista para meter los elementos repetidos ahí , luego de recorrer la lista tendremos un arbol binario de busqueda balanceado con $\log n$ elementos , por lo tanto la insercion nos toma $O(\log \log n)$, luego vamos a recorrer el arbol en orden , de esta manera los tendremos los elementos ordenados y conforme recorremos los vamos poniendo en una lista nueva (la ordenada) , entonces vamos a recorrer n elementos (ya que solo tenemos $\log n$ elementos en el arbol , pero algunos estan en las listas de los nodos) , mientras recorremos el arbol , agregamos al final de la lista nueva y terminamos de ordenar

Veamos la complejidad , primero recorremos la lista $O(n)$ y para cada elemento lo agregamos al arbol de $\log n$ elementos $O(\log \log n)$ por lo tanto construir todo el paso nos toma $O(n \log \log n)$,

luego recorremos el arbol en inorder que nos toma $O(n)$, ya que hay n elementos en total en el arbol (solo uno dentro de listas)

y agregar al final de la nueva lista $O(1)$, entonces este paso toma $O(n)$, por lo tanto todo el algoritmo toma $O(n \log n)$

Este algoritmo es correcto ya que al usar inorder en el arbol binario de busqueda balanceado, vamos a ver a los elementos ordenados y de esta manera los ordenamos

b) la cota inferior de $O(n \log n)$ no se vio debido a que primero, en clase vimos que necesitamos cumplir con las condiciones, 1) no sabemos nada de los datos, 2) mala memoria y 3) usar comparaciones

y en este problema si sabemos informacion adicional acerca de los datos y otra cosa es que $O(n \log \log n) \in O(n \log n)$, tenemos

que $C=3$ y $N_0 = 3$, fuente geogebra.

2) ° inversos del arreglo $[2, 3, 8, 6, 1]$

primera $(2, 1)$ ya que $1 < 5$, $A[1] = 2 > 1 = A[5]$

segunda $(3, 1)$ ya que $2 < 5$, $A[2] = 3 > 1 = A[5]$

tercera $(8, 1)$ ya que $3 < 4$, $A[3] = 8 > 1 = A[5]$

cuarta $(8, 1)$ ya que $3 < 5$, $A[3] = 8 > 1 = A[5]$

quinta $(6, 1)$ ya que $4 < 5$, $A[4] = 6 > 1 = A[5]$

• el arreglo con elementos del conjunto $\{1, 2, \dots, n\}$ con más inversiones

es el arreglo $\{n, n-1, \dots, 2, 1\}$, es decir cuando están ordenados

de mayor a menor, ya que el elemento n tendrá inversión con todos

los $n-1$ elementos después de él, el elemento $n-1$ tendrá inversión con

los $n-2$ elementos después de él y así, por lo tanto tiene

$(n-1) + (n-2) + \dots + 1 + 0$ inversiones, que son $\frac{(n-1)(n)}{2}$ inversiones

• algoritmo, vamos a usar un árbol de búsqueda autoequilibrado

(como un AVL), donde cada nodo tendremos la cantidad (tamaño) de

nodos en su hijo derecho (los elementos mayores que aparecen antes en el arreglo)

entonces el algoritmo es así

creamos nuestro arbolito (será nulo de momento) y una variable total

initializada en 0, luego empezamos a recorrer el arreglo de n

elementos, por cada elemento lo insertamos al arbol de esta manera

tendremos una variable mayores initializada en 0, primero buscamos

el lugar del nuevo elemento haciendo insertar en la raíz y comparando

con la raíz, si es menor entonces hacemos insertar en el hijo izquierdo

y le sumamos a mayor el tamaño del hijo derecho más 1, pero si

es mayor hacemos insertar en el hijo derecho, cuando encontramos su

lugar lo ponemos y sumamos mayor a total, ahora nos falta

balancear el arbol y actualizar el tamaño de los nodos entonces

vamos balanceando y actualizando el tamaño (sumando lo de sus hijos más 1)

por llamada recursiva que haremos (nos vamos moviendo por nodos padres hermanos,

raíz), para balancear depende del arbol usando pero por lo general

es viendo la altura de los hijos y hacer rotaciones (creas),

entonces luego de insertar todos los elementos tendremos el nuevo de insertores

en total

Veamos la complejidad, como usamos un árbol binario de búsqueda autoequilibrado, sabemos que insertar siempre nos toma $O(\log n)$ y como insertamos n elementos, lo hacemos en $O(n \log n)$, como al insertar llevamos un conteo de los elementos mayores al final solo regresamos total, toma $O(1)$, en total toma $O(n \log n)$

y el algoritmo funciona ya que conforme vamos retirando elementos y saber cuantos elementos mayores tiene (gracias al árbol), checamos los elementos mayores que están antes en el arreglo (generan inclusiones)

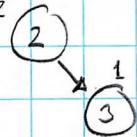
y al final se suma b de cada nodo, obteniendo lo que queremos

Ejemplo [2,3,8,6,1]

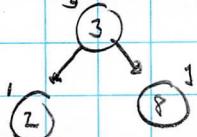
total 0 may 0



total 0 may 0

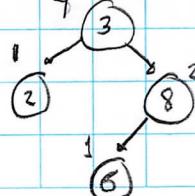


total 0 may 0

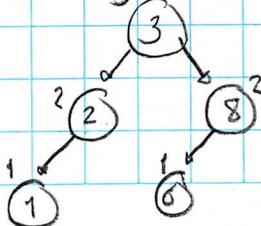


1

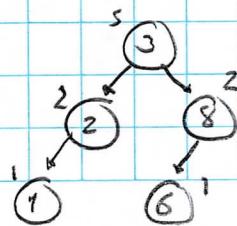
total 0 may 1



total 5 may 4



total 5



3) a) para esto usaremos una lista circular doblemente ligada, (similar a una lista doblemente ligada, solo que la coda apunta a la cabeza) entonces agregar al final, si no hay cabeza se vuelve la cabeza, solo nos toma decir quien sera el proximo de la cabeza y el anterior de la antigua cabeza, asi como el siguiente y anterior del nuevo, por lo tanto esto nos toma $O(1)$

eliminar la cabeza tambien (con 1 elemento la cabeza es null), solo en decir quien sera el anterior del siguiente de la cabeza, el siguiente del anterior de la cabeza y decir borrar cabeza, toma $O(1)$

ahora tambien usaremos un metodo moverse que resive un k entero y move la cabeza al siguiente de la cabeza n veces, esto nos toma $O(k)$

ya que mover la cabeza k veces

ahora veamos como obtener la permutacion

creamos la lista y retengamos los n elementos, luego mientras la cabeza no es null, hacemos

moverse $m-1$, tomamos el valor de la cabeza, lo ponemos

en la permutación, eliminamos la cabeza y repetimos.

al final la cabeza es null, terminamos y tenemos la permutación

Veamos la complejidad, construir la lista nos toma $O(n)$ (insertando

a elemento y agregar toma $O(1)$), para la permutación

movemos la cabeza n posiciones $O(n)$ y lo hacemos siempre antes de

eliminar cada elemento, entonces la permutación nos toma $O(n^2)$

y todo toma $O(n^2)$, pero n es constante, entonces la complejidad es

$O(n)$

Y el algoritmo funciona ya que simulamos el problema gracias a

la lista circular, por lo que recreamos el problema fielmente y de

manera sencilla.

b) para esto usamos un árbol binario de búsqueda autobalanceado

(como AVL), pero cada nodo tendrá su tamaño ($1 + \text{tamaño de sus hijos}$,

hojas tamaño 1), usaremos los métodos de un árbol AVL para agregar

(buscar su lugar insertar y rebalancear) y eliminar (explicado después),

actualizando el tamaño cada operación, ademas un método especial

para encontrar al ítemo elemento más pequeño y restar su numero

veamos primero como eliminar , dato el valor a eliminar , lo buscamos
y encontramos su nodo , tenemos casos:

caso 1: es hoja entonces solo borramos

caso 2: tiene un hijo , copiamos el valor del hijo al nodo y luego borramos
al hijo

caso 3: tiene 2 hijos , buscamos su sucesor inorden , copiamos su valor en el
nodo y borramos el sucesor inorden

desde esto empezamos donde esta el nodo eliminado y empezamos
a subir a la raiz (actualizando tamaños) , encontramos al primer arbol
desbalanceado y lo rebalanceam (actualizando tamaño, tambien) mediante rotaciones,
luego empezam a subir por el nodo que rebalancean y repetimis este proceso
hasta la raiz (es la eliminacion de un arbol AVL).

ahora para el metodo especial sera buscar(nodo x, indice i)

usamos la variable C con el tamaño de su hijo Izq mas 1 y comparamos i con C
tenemos 3 casos

caso 1: i igual a c , regresamos el valor de X

caso 2: i menor a c , regresamos el resultado de buscar (hijo izq, i)

caso 3: i mayor a c , regresamos el resultado de buscar (hijo der, $i-c$)

ahora veremos como funciona todo junto

primero creamos el arbol y tenemos todos los elementos, ahora

tenemos dos variables, la primera $s=0$ y la segunda $t=n$ (tamaño de la rama),

ahora mientras $t > 0$ hacemos esto $s = (stn-1) \% t$, luego

buscamos (raiz; $s+1$) sin modificar a s , luego ya tendremos el valor

del s -ésimo elemento más pequeño del arbol, el valor lo agregamos a

la permutacion y lo mandamos a eliminar, por ultimo reducimos

t en 1 y repetimos todo hasta que $t > 0$ no se cumple y tenemos la

permutacion

Veamos la complejidad, creamos el arbol toma $O(n \log n)$, ya

que agregamos n elementos y agregar toma $O(\log n)$, buscamos el

s -ésimo elemento nos toma $O(\log n)$ ya que hacemos una operacion

por nivel del arbol y borrar nos toma $O(\log n)$ (todo basado en AVL)

como buscamos y eliminamos 1 veces para la permutacion, esto nos toma $O(n \log n)$, y todo en conjunto nos toma $O(n \log n)$

Y este algoritmo funciona ya que tenemos una forma para tener el indice del siguiente elemento a eliminar (mediante nuestra formula), por eso lo buscamos y eliminamos, todo gracias a que la formula nos da el indice del elemento anterior del proximo a eliminar (por eso buscar(raiz, st1)), entonces al eliminarlo y reducir el tamaño del arbol, los indices de los elementos "se reacomodan" para el siguiente ciclo y asi sacamos al elemento correcto

4) a) algoritmo $O(n^2)$

Tenemos n contenedores de madera, entonces tomamos al primero y lo vamos a comparar con cada uno de los n contenedores de vidrio hasta encontrar el del mismo tamaño (les cabe la misma cantidad), luego pasamos al segundo contenedor de madera y repetimos comparar contenedores de vidrio, hacemos esto hasta terminar con los contenedores de madera y terminamos.

Vemos la complejidad, a cada cont de madera lo comparamos con a lo más n cont de vidrio, y como son n de madera entonces son $O(n^2)$ comparaciones. Y el algoritmo sigue, ya que comparamos a todos los cont de madera con todos los de vidrio para encontrar su pareja.

b) hacer en $2n-2$ comparaciones

Veamos como hacerlo en a lo más $2n-2$ comparaciones

primero ponemos a los cont de madera en una fila y otra para los de vidrio, luego tomamos al primero de la de madera como el mayor de su lista y al primero de vidrio como el mayor de su lista,

(el de madera es el maximo 1
y si el de vidrio tiene tamano igual, ese es el maximo 2)

luego los comparamos y al mayor lo llamemos maximo, supongamos

sin perdida de generalidad que el de madera sea el mayor,

entonces lo compararemos con el siguiente de vidrio y tendremos 3 casos:

caso 1: el maximo (madera) es mayor al de vidrio, entonces seguimos comparando con el maximo

caso 2: el maximo (madera) es igual al de vidrio, entonces guardamos al de vidrio como el maximo 2 y seguimos comparando con el maximo

caso 3: el maximo (madera) es menor al de vidrio, entonces hacemos al de vidrio como el nuevo maximo y quitamos aquello que estaba como maximo 2

este proceso es análogo para el caso de vidrio

De esta manera vamos comparando hasta comparar con el ultimo

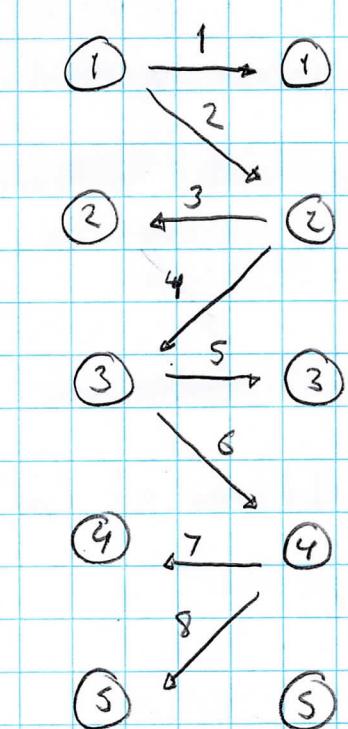
elemento de madera o vidrio, aquí hay dos casos, si tenemos un maximo 2 ya acabamos, pero si no tenemos entonces hay dos casos si solo nos falta un elemento por comparar en contenedor de los que no

terminan lo volvemos maximo 2 (no acabamos la comparacion), pero si hay varios contenedores, tenemos que compararlos con maximo y seguir

chequeando estos dos últimos casos (para ahorrar 1 comparación)

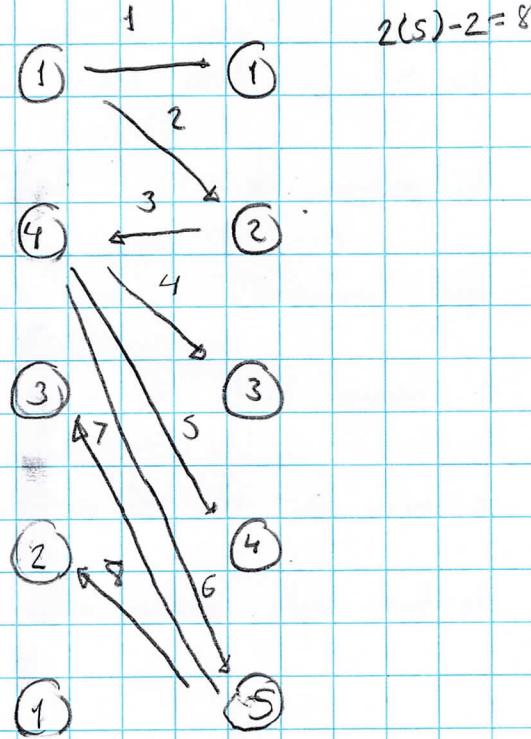
Si seguimos estos pasos, nuestro peor caso será cuando tenemos que de las listas de cont este ordenada, ya que cada 2 comparaciones cambiaremos al máximo, en una cambiaremos al mínimo, en otra te la vez al máximo 2, esto lo hacemos $n-1$ veces, por lo tanto toma $2(n-1) = 2n-2$ comparaciones

Ejemplo peor caso



$$2(5)-2=8$$

Ejemplo peor caso



$$2(5)-2=8$$

c) Mejorar a?

lo que podemos hacer es intentar ordenar los cont de madera y vidrio para así solo ir tomando parejitas en $O(n)$ para ordenarlos podemos hacer algo como un tipo quicksort , tomamos un cont de madera aleatorio m_1 y lo usamos para partir a los contenedores de vidrio al compararlo con todos los de vidrio y así encontrar la posición del cont de vidrio que tiene la misma cantidad (v_1) que m_1 , luego usamos a v_1 para repetir este proceso y encontrar la posición de m_1 , luego repetimos todo el proceso en las particiones que generamos , muy similar a quicksort si hacemos esto de esta manera tendremos $O(n \log n)$ comparaciones a más que sea el peor caso (sería la complejidad del merge a) , por lo tanto esto podría mejorar al más a , tal vez

5) a) para la primera estructura, nos basamos en un arreglo de arreglos o matriz, entonces tenemos una matriz $n \times n$ b que contiene cada posición sera esto , la posición $\text{mat}[i][j]$ tendrá al elemento mínimo del intervalo $\{x_i, \dots, x_j\}$, que se obtendrá comparando los elementos del intervalo $O(n)$, pero si $j < i$ dejamos vacía la posición, (llenamos media matriz), ahora cuarto tenemos que responder la pregunta solo vemos que hay en la posición que nos piden, si nos dan a, b entonces devolvemos $\text{mat}[a][b]$ y por propiedades de los arreglos salimos que esto nos toma $O(1)$ y para el espacio que ocupa , tenemos una matriz de $n \times n$, por lo tanto hay n^2 posiciones y ese es $O(n^2)$ espacio

b) para la segunda estructura, nos basamos en un árbol binario autobalanceado, tendremos nuestro árbol de n hojas , en las hojas estaran los valores x_1, \dots, x_n y su rango $[1:1], \dots, [n:n]$, en los nodos internos tendremos al mínimo valor de los nodos inferiores y el rango que lo cubre , como tiene n hojas sabemos que va a tener

$2n-1$ nodos en total, por lo tanto el espacio es $O(n)$

y ahora cuando recibimos i y j veamos como encontramos el minimo del intervalo, realizaremos este proceso

dado un nodo x y un intervalo i, j entonces hay 3 casos:

caso 1: si el rango del nodo x esta entre i y j entonces regresamos el valor minimo dentro del nodo x

caso 2: si el rango del nodo x esta completamente fuera de i y j entonces regresamos un positivo muy grande

caso 3: en otro caso regresamos el minimo de los valores obtenidos de aplicar este proceso a los nodos hijos de x

este proceso lo empezamos en la raiz con el intervalo que nos piden

(otra manera seria empezar a la raiz e ir bajando a las hojas,

i y j , y luego regresar el minimo de lo que encontramos a su camino, aunque no uso este metodo porque se me dificulta algo como explicarlo),

de esta manera vamos bajando por los niveles del arbol procediendo

a lo mas a los nodos por nivel, como tenemos n hojas y vamos

bajando por el arbol, encontrar al minimo del rango dado nos toma $O(\log n)$

6) el algoritmo es este

Tenemos los arreglos A, B y el entero k ($1 \leq k \leq 2n$)

el caso base sera cuando un arreglo sea el vacio , sin perdida de generalidad supongamos que es A , entonces regresamos $B[k]$, de manera anloga si B es el vacio

ahora veamos el paso recurrente

sean midA y midB los indices de los elementos medios de A y B

respectivamente , entonces compararemos los valores $A[\text{midA}]$ y $B[\text{midB}]$ (sus valores medios) y tenemos estos casos:

Caso 1: se cumple $\text{midA} + \text{midB} < k$ y $A[\text{midA}] > B[\text{midB}]$, entonces descartamos la primera mitad de B ($B[1], \dots, B[\text{midB}]$) y actualizamos a k como $k - \text{midB} - 1$, y repetimos este proceso

Caso 2: se cumple $\text{midA} + \text{midB} < k$ y $A[\text{midA}] \leq B[\text{midB}]$, entonces descartamos la primera mitad de A ($A[1], \dots, A[\text{midA}]$) y actualizamos a k como $k - \text{midA} - 1$, y repetimos este proceso

Caso 3: se cumple $\text{midA} + \text{midB} \geq k$ y $A[\text{midA}] > B[\text{midB}]$, entonces

descartamos la segunda mitad de A ($A[\text{midA}+1], \dots, A[n]$) , y repetimos este proceso

caso 4: se cumple $\text{midA} + \text{midB} \geq k$ y $A[\text{midA}] < B[\text{midB}]$, entonces

descartamos la segunda mitad de B ($B[\text{midB}+1], \dots, B[n]$). , y repetimos

de este manera obtenemos al k-ésimo elemento

Veamos la complejidad, estamos haciendo algo similar a binary search, ya que en cada "ciclo" estamos descartando una mitad de los de los arreglos , por lo tanto es como si hicieramos binary search en ambos arreglos y esto nos tomaría $O(\log n)$ cada uno entonces todo toma $O(2\log n)$ y eso es $O(\log n)$

Y funciona ya que usamos un proceso similar a binary search para ir reduciendo los posibles candidatos , de manera que en cada paso tenemos menos y al final nos encontramos con un caso trivial que ya sabemos resolver , ademas nos basamos en $\text{midA} + \text{midB}$, el menor para ver que mitad desechar , siempre quitamos la mitad donde no puede estar el k-ésimo elemento , si $\text{midA} + \text{midB} < k$ es la mitad menor al menor

elemento de las mitades , ya que el k -ésimo elemento tiene que estar desde de $A[\text{midA}] \circ B[\text{midB}]$, y si $\text{midA} + \text{midB} \geq k$, entonces quitamos la mitad mayor del elemento mayor , ya que el k -ésimo elemento debe estar antes de $A[\text{midA}] \circ B[\text{midB}]$ (o ser uno de ellos.)

7) Vamos a usar un árbol binario, primero diremos como servirá el árbol, los n elementos de A serán las hojas de nuestro árbol, tendrá n hojas, ahora cada nodo en el árbol tendrá la suma de sus hijos (llamaremos a esto el valor del nodo, en el caso de las hojas es su valor) y también tendrá la mitad del rango de sus hijos (lo llamaremos rango, en el caso de la i -ésima hoja es $[i:i]$), ahora veamos como simularemos el árbol, usaremos los dos arreglos para esto, para facilitar esto "uniremos" los arreglos (también podríamos solo usar el árbol creo, pero por alguna razón decidí usar un arreglo $-L(\cdot)\cdot\cdot\cdot$) y tendremos un arreglo de tamaño $2n$, entonces la raíz del árbol está en el índice 1, la i -ésima hoja está en el índice $nti-1$, el padre del i -ésimo nodo es $\frac{i}{2}$ (si i es par) o $\frac{i}{2}-1$ (si i es impar), así será el árbol veamos como construiremos el "árbol", primero empezaremos recordando el arreglo A de otros hacer debajo, en cada elemento i de A haremos esto, tomamos su valor, y "almacenamos" $A[i]$ y al índice $nti-1$ del arreglo que simula al árbol le ponemos el valor que tomamos

$\text{Arbol}[n:i-1] = \text{valor que estaba en } A[i]$, cuando terminemos de recorrer

A (todo este proceso se puede "ahorrar" poniendo el arreglo A como

la segunda mitad del arreglo que simula el arbol) , teniendo la raiz como

el indice 2 y invirtiendo los casos para encontrar el .padre i , creo),

nos paramos en la raiz del arbol (indice 1) , lo actualizamos , es decir

actualizamos su valor con la suma del valor de sus hijos y actualizamos

su rango con la union de los rangos de sus hijos , como nuestro arbol

empezu vacio (menos las raizes) antes de actualizar un nodo actualizamos

a sus hijos , y asi tenemos nuestro arbol

ahora como son las operaciones

para Add(i,y) , hacemos esto , vamos al indice n:i-1 del arbol

arreglo y hacemos $\text{Arbol}[n:i-1] += y$, luego vamos a su padre

y lo actualizamos , repetimos actualizar al padre hasta la raiz

para partial sum(i) , hacemos esto, tendremos el metodo bisecc(i,nodo x)

entonces sea $[a:i:b]$ el rango del nodo x , $[c:i:d]$ el rango del hijo izq

del nodo x y $[e:f]$ el rango del hijo der ; tenemos 4 casos:

caso 1: $d = i$, entonces regresamos el valor del hijo izq

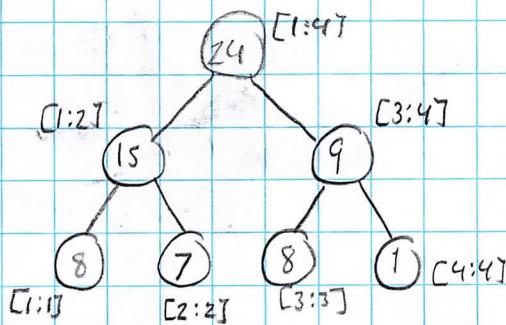
caso 2: $f = i$, regresamos la suma del valor del hijo izq y el valor del hijo der

caso 3: $i > d$, entonces regresamos la suma del valor del hijo izq y el resultado de buscar (i , hijo der)

caso 4: $i < d$, entonces regresamos el resultado de buscar (i , hijo izq)

entonces cuando recibamos partial sum (i) regresamos el resultado de buscar (i , root)

ejemplo del arbol



la complejidad de ambas operaciones es $O(\log n)$ debido que en cada operación hacemos operaciones constantes (en cada actualizar y en partial sum comparar) en cada nivel del arbol (que son $\log n$ niveles) entonces la complejidad es $O(\log n)$

y el algoritmo funciona, ya que usamos el arbol binario para

ayudarnos a no hacer tantas sumas en las operaciones y solo lo actualizamos para que siga sirviendo

* nota todos los indices los hacemos modulo n (+1?) para evitar index out of bounds

- 8) vamos a hacer un metodo busmin para encontrar el indice del minimo elemento, el metodo busmin recibe un arreglo A, un indice i menor y indice mayor j, el metodo funcionara asi
- si $A[i] \leq A[j]$, entonces regresamos i, en otro caso mientras $i <= j$, hacemos esto tomamos a mid como $mid = \frac{i+j}{2}$
- ahora vemos si mid es el minimo, si $A[mid] < A[mid-1]$ entonces regresamos mid, en otro caso revisemos si alguna mitad estan ordenadas, para esto comprparamos $A[mid]$ y $A[j]$, si $A[mid] > A[j]$ significa que el minimo esta en la derecha, por lo tanto actualizamos a i como mid+1, pero si $A[mid] > A[j]$ no pasa, es qe el minimo esta a la izquierda, actualizamos j como mid-1 y repetimos esto hasta qe $i <= j$ no se cumpla
- ya qe tenemos el indice del elemento minimo (i-ésimo elemento) llamemoslo p,
revisamos a k, si $k < 1$ o $k > n$ entonces regresamos error, y en otro caso regresamos el elemento en el indice $(p+k) \% n$ en el arreglo A, qd decir el elemento del indice $p+k-1$, solo nos

Fijons que no nos salgamos del arreglo, es decir $p+k-1$ si
 $p+k-1 \leq n$ o $p+k-1-n$ si $p+k-1 > n$

Este algoritmo toma $O(\log n)$, debido a que cuando buscamos

el índice del mínimo elemento vamos partiendo el arreglo a la mitad por lo tanto lo partimos a lo mas $\log n$ veces, y en cada

partición hacemos operaciones constantes (comparaciones) por lo tanto

todo nos toma $O(\log n)$ y devolver el valor es $O(1)$

El algoritmo funciona ya que primero buscamos al mínimo elemento

usando algo similar a búsqueda binaria y luego que lo tenemos ya solo

resolvemos al elemento pedido

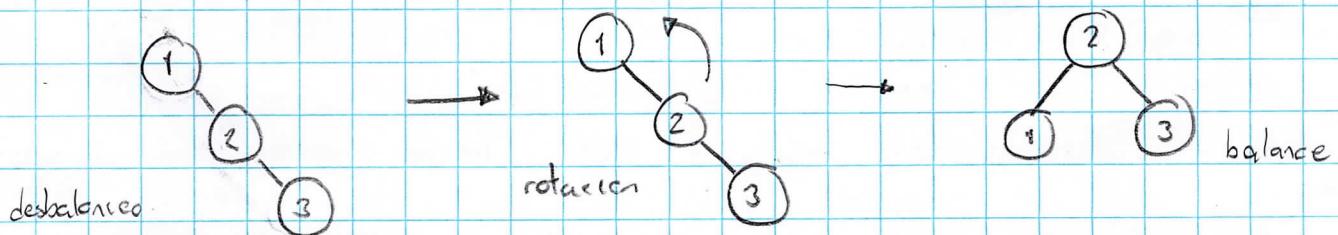
como usar varios árboles AVL en algunos ejercicios, veamos

como funcionan

primera veamos las rotaciones

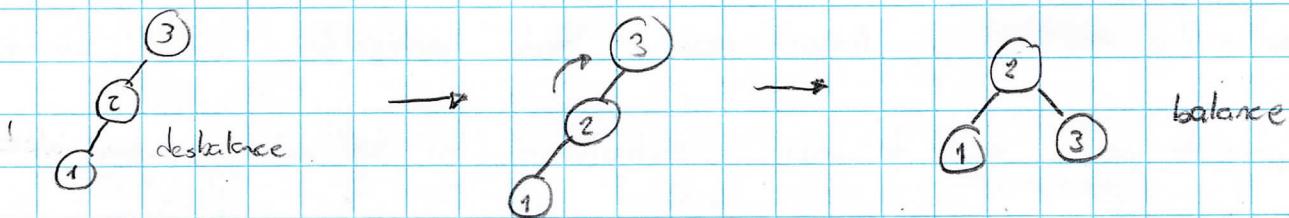
- rotación a la Izquierda, si agregamos un hijo derecho y se desbalancea

hacemos una rotación (solo es actualizar los hijos de los nodos y actualizar los nodos si guardan su tamaño)



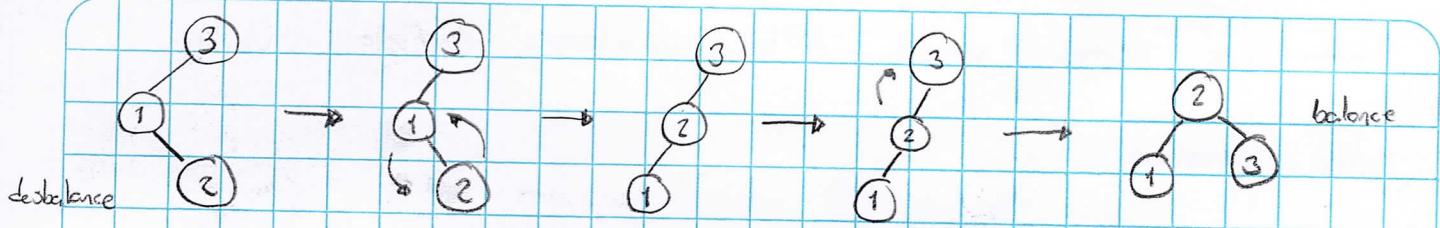
- rotación a la izquierda, es análoga a la rotación a la derecha, solo

con agregar a la izquierda y rotar a la derecha

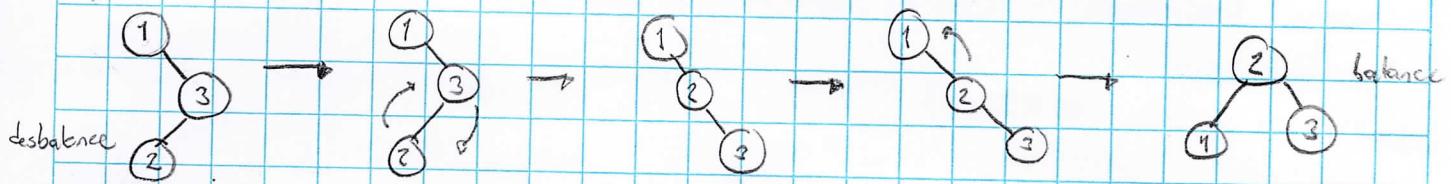


- rotación izq der, es una combinación de las rotaciones anteriores,

primero una izq y luego una der



- rotacion der 12g, es una combinacion de las dos primeras, primero der y luego 12g



todas las rotaciones tienen complejidad $O(1)$, siempre sin las mismas operaciones

ahora veamos la insercion, los pasos para esto es, empezando

en la raiz vamos buscando el lugar del nuevo elemento, viendo si es

menor o mayor que la raiz, luego recursivamente hasta encontrar un hijo vacio

(su hijo) e insertar, luego desde el nodo agregado vamos subiendo

por los padres hasta la raiz, actualizando si los nodos guardan

algun valor extra (como tamaño) y checando si el arbol esta

balanceado, si tenemos un desbalanceo en a, con b el hijo de dcha

mete el nuevo elemento y es el nieto de a en el camino del nuevo a q,

entonces entonces dependiendo de que hijo sean b y c, hacen la rotacion necesaria para rebalancear, casos:

caso 1: b es el hijo izq de a y c es el hijo izq de b, entonces

notamos a la derecha a, a, y actualizamos los datos adicionales que

guardan (si tienen) a, b y c

caso 2: b es el hijo izq de a y c es el hijo der de b, entonces

hacemos una rotacion izq der (rotar izq a b y luego rotar der a a),

actualizamos datos en a, b y c

caso 3: b es el hijo der de a y c es el hijo izq de b, entonces

hacemos una rotacion der izq (rotar der a b y rotar izq a a), actualizamos datos

caso 4: b es el hijo der de a y c es el hijo der de b, entonces

hacemos rotacion der a a, actualizamos datos

luego de rotar seguimos subiendo hasta la raiz, balanceando (si estan desbalanceados) y actualizando datos extras de los nodos

, por lo general

para ver si esta balanceado se checa la altura de los arboles

notemos de insertar nos toma $O(\log n)$, primero buscamos el nodo correcto, esto es ir bajando por niveles del árbol y nos toma $O(\log n)$, luego toca volver a subir e ir actualizando o rebalanceando los nodos $O(\log n)$ también

Ahora veamos eliminar un elemento, primero buscamos al elemento por eliminar y tenemos caso de quien es:

Caso 1: es una hoja, entonces solo la borramos (cortar la conexión de hijo con su padre)

Caso 2: el elemento a eliminar, tiene un hijo, entonces copiamos el valor del hijo y lo ponemos en el nodo del por eliminar, mandamos a borrar al hijo

Caso 3: el elemento por eliminar, tiene dos hijos, entonces buscamos el sucesor del elemento a eliminar (y esto se hace buscando al mínimo valor del hijo der) entonces cambiamos el valor del sucesor y lo mandamos a eliminar (como en el caso 2)

hijo de eliminar, al nodo eliminado sera a, entonces desde a

luego subiendo por los padres hasta la raíz, actualizando datos

extra de los nodos y checando si el arbol está balanceado (como en la inserción)

si el arbol del nodo b está desbalanceado entonces checamos estos nodos,

c es el hijo de altura mayor de b ; y d es el hijo de altura mayor de c , entonces

dependiendo de quienes sean b,c y d , con la rotación para rebalancear , casos:

caso 1: c es el hijo izq de b y d el hijo izq de C , entonces rotacion

der a b , actualizan datos extra de b, c y d

caso 2: c es el hijo izq de b y d el hijo der de c , entonces hacemos

rotacion izq der (rotacion izq a c y rotacion der a b) ; actualizar
datos

caso 3: c es el hijo der de b y d el hijo izq de c , entonces hacemos

rotacion der izq (rotacion der a c y rotacion izq a b) , actualizar

datos

caso 4: c es el hijo der de b y d el hijo der de c , entonces hacemos

rotacion der a b , actualizan datos

luego te vuelve segun subiendo hasta la raíz , actualizando datos y

chequeando si está balanceado (en este caso balanceando con rotaciones)

la complejidad de eliminar es $O(\log n)$, busco el elemento y eliminamos toma $O(\log n)$ ya que buscamos bajando por niveles del árbol, y al eliminar si nos toca el caso 2 o 3, entonces solo bajamos más en el árbol por lo tanto esto es $O(\log n)$ y luego vamos subiendo y actualizando hasta la raíz, aquí también hacemos operaciones constantes en cada nivel entonces es $O(\log n)$

Entonces así funcionan insertar y eliminar en AVL (los que uso para la tarea), y lo importante es que como se autobalancea la altura siempre es $\log n$ y por lo tanto las operaciones anteriores son $O(\log n)$