Explique detalladamente el concepto de igualdad observacional. Relacionar el rol de la igualdad observacional con la axiomatizacion de DameUno : Conj [alfa] → alfa y explique alternativas y dificultades de axiomatizar alternativas.

Tomás Caballero & Paulo Ballan: El rol que cumple la igualdad observacional en la axiomatización de DameUno de Conj es la "restricción" que le pone a la función para que no rompamos congruencia. Por ejemplo si utilizaramos una alternativa a la axiomatización existente tal que: DU(Ag(a,c)) = a, esto nos daría un orden en nuestro conjunto que no es lo que se busca con un conjunto, ya que: DU(Ag(b,Ag(c,vacío))) != DU(Ag(c,Ag(b,vacío))) por lo tanto rompemos congruencia.

Se cuenta con n elementos que contienen una clave primaria y una secundaria. Hay m<<n claves secundarias distintas. Proponga dos formas eficientes y distintas que permitan ordenar los n elementos, en base a las dos claves (primero la primaria, luego la secundaria). Una de las dos formas tiene que utilizar solo arreglos.

Explicar por qué la función de abstracción no es sobreyectiva sobre el conjunto de términos. ¿Hay algún conjunto para el que sí lo sea?

Nicolás M. Obesio:
- No es necesariamente sobreyectiva sobre el cjto. de terminos de un TAD ya que por la forma en la que el ABS es construido NO es posible diferenciar entre instancias del TAD observacionalmente iguales y por lo tanto NO es posible garantizar que todo termino del TAD sea imagen del ABS para alguna estructura de representación.
- La función de abstracción si es sobreyectiva sobre las clases de equivalencia. (Su imagen contiene al menos un representante de cada clase) "

Enumerar y describir los tipos de recursión que se pueden eliminar con las técnicas vistas en la materia. Dar una esquematización (descripción general) de como aplicaría cada una.

Nicolás M. Obesio:

Tipos de Recursión

- Recursión lineal: 1 llamada recursiva
---- A la cola: Ultima operación es una llamada recursiva. (Ej: Busqueda Lineal)
---- NO a la cola: Ultima operación NO es una llamada recursiva. (Ej: Sumatoria)
- Recursión Multiple: 2 o más llamadas recursivas
---- Anidadas ('A la cola')
---- Multiples ('NO a la cola')

Procedimiento para eliminar recursión

- Recursión Lineal A la cola:
```
funcion F(params):
    while not caso_base:
        params = achicar(params)
    return algun valor
```
- Recursión Lineal NO a la cola:
  La transformo a la cola utilizando inmersión y aplico el primer procedimiento
- Recursión No-Lineal anidada:
  IDEA: Usar un pila como 'cola de procesamiento'
- Recursión No-Lineal multiple:
  Hacerla anidada y aplicar el procedimiento anterior."

El sistema de especificación TAD' es muy similar a los TADs que se utilizan en la materia, pero su única igualdad entre instancias se define así: dos instancias son iguales si y solo si aplicando axiomas se puede llegar de una a la otra.

1 Este sistema de especificación no tiene uno de los problemas formales más importantes que puede tener un TAD. ¿Cuál? ¿Por qué?
2 ¿Qué dificultad presenta la especificación del TAD' conjunto en este formalismo?

Nico Pazos: 1) El rompimiento de la congruencia. Porque la igualdad entre las instancias no dependería de la definición de la igualdad observacional, sino de la axiomatización de las funciones, que se basa en la forma sintáctica de cada instancia. El problema ocurre cuando dos instancias que son iguales se comportan distinto ante una función. Como en los TAD' las funciones definen la igualdad, es imposible que la igualdad contradiga a las funciones. DEAL WITH IT. 2) Dos conjuntos serían iguales solo si sus elementos fueron agregados en el mismo orden. Es decir, Ag(1,Ag(2,/0)) sería distinto a Ag(2,Ag(1,/0)). Para conservar la igualdad que se busca se podrían restringir los dominios de las operaciones (por ejemplo, que los conjuntos de géneros ordenables estén "ordenados" según su tira de generadores) o hacer algo trambólico que tire rayo láser, pero no es algo deseable.

Nicolás M. Obesio:
1- La igualdad es una congruencia.
2- Los modelos son mas feos. Mala legibilidad y menos intuitivo.

Escribir el algoritmo que convierte un arreglo a un heap, justificar su complejidad y por qué lo usaría de estructura soporte.

Nicolás M. Obesio:

```
heapify(arr, size):
    start = size / 2
    while start >= 0:
        sift_down(arr, start, size-1)
        start = start - 1
```

Este algoritmo a simple vista en O(n log n) pero si hacemos un analisis mas riguroso vemos que es O(n). El argumento es que el sift_down no siempre cuesta O(log n).

VER DEMO A LA DERECHA

https://stackoverflow growingwiththeweb 20why,when%20it's

Theoretically:

The Total steps `N` to build a heap of size `n`, can be written out mathematically by summing steps above:

$$N = \sum_{i=0}^{log(n)} \frac{n}{2^{i+1}} i = \frac{n}{2}\left(\sum_{i=0}^{log(n)} i\,(1/2)^i\right) \leq \frac{n}{2}\left(\sum_{i=0}^{\infty} i\,(1/2)^i\right)$$

The solution to the last summation can be found by taking the derivative of both sides of the well known geometric series equation:

$$\frac{\partial}{\partial x}\left(\sum_{i=0}^{\infty} x^i\right) = \frac{\partial}{\partial x}\left(\frac{1}{1-x}\right) \quad \Rightarrow \quad \sum_{i=1}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

Finally, plugging in `x = 1/2` into the above equation yields `2`. Plugging this into the first equation gives:

$$N \leq \frac{n}{2}(2) = n$$

Thus, the total number of steps is of size `O(n)`

Vincular la forma general de los algoritmos que utilizan la técnica de divide and conquer con la forma en la que se obtiene la ecuación de recurrencia que caracteriza la complejidad de un algoritmo recursivo y con los casos del teorema maestro.

Nicolás M. Obesio: (A esta respuesta le falta desarrollo pero creo que va por este lado lo que esperan. El que quiera ayudar a expandirla esta bienvenido)

Divide and Conquer:
- **Dividir**: Dividir la instancia del problema en subinstancias mas pequeñas.
- **Conquistar**: Resolver los subproblemas de forma recursiva hasta llegar al caso base.
- **Combinar**: Combinar las soluciones en una unica para el problema original.

Teorema Maestro:

T(n) = | **a***T(n/**c**) + **f(n)**   n>1
          | O(1)   n=1

donde:
- **a** es la cantidad de subproblemas a subproblemas a resolver (DIVIDIR)
- **c** es la cantidad de particiones (Generalmente coincide con a)
- **f(n)** es el costo de "DIVIDIR" y "COMBINAR"

Explíquele a un matemático como hacer inducción sobre un TAD. Intentelo ahora con un artesano de plaza francia.

Teo Freund:
-Matemático: La inducción estructural se puede pensar como una generalización de la inducción en los naturales, donde uno tiene un orden bien fundado (<) y lineal sobre N y quiere demostrar una propiedad P que vale para todos los naturales. Uno plantea una demostración para n, suponiendo que vale para todo n'<n, el paso inductivo, de esta forma, si yo quiero ver que vale para un m en particular, lo demuestro utilizando el paso inductivo, luego voy a necesitar demostrar que vale para ciertos m1,m2,...mk, continúo recursivamente, y como siempre dependo de números menores, tengo muchas secuencias estrictamente decrecientes con respecto de <, que por ser bien fundado se que estas secuencias no son infinitas, por lo que en algún momento, todas las demostraciones dependen de un solo valor, el mínimo de esta secuencia, 0 (o 1 si 0 no está en N), y esta es la demostración del caso base.
La inducción estructural se hace sobre términos de un TAD, que se que siempre están formados por generados, estos los puedo separar en generadores básicos, que no toman valores de ese mismo TAD (nuestro 0) y recursivos, que dependen de instancias "menos complejas" (nuestros pasos inductivos). Luego, todo lo que tengo que hacer es conseguir un orden bien fundado sobre ellos donde mis generadores básicos sean elementos mínimos y mis recursivos tengan cierto orden que refleje su dependencia. Se podría plantear un orden parcial donde a<<b sii b es un subtérmino de a, es decir b aparece en a, pero para no complicarse la vida, uno puede usar el siguiente, a<b sii f(a)<f(b), donde f:Terminos -> N, y se la puede definir así:
*f(generador básico)=0
*f(generador recursivo que depende de x1,x2,....,xk)=1 + max(x1,x2,....,xk)
este orden es bien fundado ya que < es b.f. sobre los naturales y la imagen de f están incluidos en los naturales.
Luego, se opera igual que antes, se hace una demostración para todos los terminos t, tal que f(t)=n+1, suponiendo que la propiedad vale para todos los términos t', tal que f(t')<f(t) (t'<t), y como se que los únicos términos que tienen f(t)>0 son los recursivos, basta hacerlo para estos. Luego, si quiero demostrar para un termino s en particular, aplico el paso inductivo repetidas veces, y por ser el orden bien fundado se que esta secuencia de dependencias terminan en los terminos s' con f(s')=0, que son los generadores básicos, y este es nuestro caso base donde hay que probar que la propiedad vale para los generadores básicos.

-Artesano: ¿Qué es un collar?
>>> Un hilo es un collar
>>> Si yo meto una piedrita en un collar, obtengo otro collar
Ahora, imaginemos que me gustaría saber que todos los collares son hermosos, bueno, entonces basta con ver la forma en que se los construye, un hilo es hermoso, por el potencial que tiene, luego si yo tengo un collar, que es hermoso, y le inserto una piedrita, pués ahora será más hermoso, por lo que la propiedad vale.
Después, para cualquier collar que yo tenga, retrocedamos en su construcción, la última piedrita que yo le inserte, me dice que es hermoso, bueno, solo si ya era hermoso; la anteúltima piedrita que le inserte, me dice que es hermoso, solo si ya era hermoso;....; el hilo con el cuál empece, es hermoso? Si, entonces, mi collar es hermoso.
Esto es la inducción, que nos deja demostrar verdades en objetos que se construyen recursivamente. Ahora, imaginemos que noe estamos limitados a collares con un solo hilo, sino que pueden tener muchos y muy distintos; y que no solo tenemos piedritas, sino que tenemos piedritas, chapitas, vidrios, etc.., Buenos, si yo quiero demostrar algo sobre este tipo de collar, debo proceder de manera similar, pero haciendolo, primero, para cada tipo de hilo, y luego, para cada tipo de adorno. Esto es la inducción estructural.

Es muy bueno, me dieron ganas de hacer un collar.

Nicolás M. Obesio:

Nicolás Hertzulis:

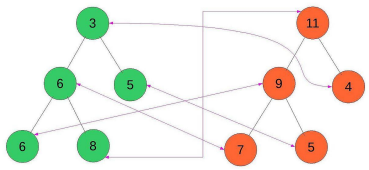
Artesano:
Alan es un artesano de Plaza Francia que se dedica a hacer collares. Para explicarle inducción estructural, vamos a hacer una analogía con sus collares.
¿Qué es un collar?
>>> Un hilo es un collar.
>>> Si metemos una piedrita en un collar, obtenemos otro collar.
El local de Alan es especial porque presta los materiales y deja que el cliente arme su propio collar. Tenemos a disposición un solo tipo de hilo y podemos usar la cantidad de piedritas que queramos pero son todas iguales.
Ahora imaginemos que queremos demostrar matemáticamente que sin importar la habilidad del cliente, cualquier collar que pueda armar será hermoso.
Si el hilo no es hermoso ya empezamos mal, porque encontramos un collar que no es hermoso. A esto le decimos caso base.
Como las piedritas que tenemos nos gustan, vemos que si tenemos un collar hermoso cualquiera y le agregamos una piedrita más, sigue siendo hermoso. A esto le decimos paso inductivo.
Si el hilo es hermoso (se cumple el caso base) y agregar una piedrita a cualquier collar hermoso hace que siga siendo hermoso (vale el paso inductivo) entonces es imposible armar un collar feo, por lo tanto acabamos de demostrar que cualquier collar que el cliente pueda armar será hermoso. A esto le llamamos una demostración por inducción. Con la inducción podemos demostrar características (lindas, feas, etc) sobre las artesanías de cualquier local, siempre que sus artesanías se construyan agregando materiales iguales a un material base.
Mientras le explicábamos a Alan se acercó Betty, que tiene un local de collares con la misma modalidad que Alan pero el cliente dispone de dos colores de hilo (negro y blanco) y tres tipos de adorno (piedritas, chapitas y vidrios). Un collar puede tener varias tiras de hilo de distinto color y la cantidad de adornos que queramos de cualquier tipo. Como en el local de Alan, acá también queremos demostrar que todos los collares son hermosos.
En este caso tenemos dos casos base: uno por cada color de hilo. Si los dos colores son hermosos, los casos base valen.
El paso inductivo es: si todos los adornos y los dos colores de hilo nos gustan, entonces para cualquier collar hermoso que tengamos, agregarle un adorno o una tira de hilo hace que siga siendo hermoso. Esto es inducción estructural sobre el local de Betty. La inducción estructural permite demostrar características de las artesanías de cualquier local, sin importar la variedad de materiales que utilice.
Para seguir con nuestro vocabulario matemático, de ahora en más en lugar de decir local de Alan o local de Betty diremos TAD de Alan y TAD de Betty y a cada collar lo llamaremos instancia del TAD.
Notemos que hay infinitos TADs posibles, solo agregando un adorno o un hilo distinto ya tenemos otro TAD. Y siempre podemos proceder de la misma manera para demostrar cosas. Si escribimos el procedimiento de manera genérica, sin estar atados a ningún TAD, diremos que escribimos la generalización de la inducción estructural.

Explique detalladamente el rol que juega el invariante de representación en el diseño jerarquico de TADs.

Carlos Giudice: El invariante de representacion es una funcion que toma como parametro cualquier elemento de la imagen abstracta del genero de representacion del TAD modelado y devuelve como resultado un valor booleano que es true si, y solo si, la estructura de entrada es una instancia valida del tipo. Hay varios objetivos que son cumplidos al crear un buen invariante de representacion, el primero es una coherencia interna donde no hay contradicciones entre informacion redundante, por ejemplo una estructura conjunto que tiene un nat t para guardar el tamaño, no deberia tener mas o menos que t elementos almacenados. Tambien es util el invariante de representacion porque nos garantiza informacion que puede ser crucial a la hora de demostrar propiedades del tipo en cuestion, por ejemplo la propiedad de balanceo de un AVL es la propiedad que usamos para conocer su altura log(n) y consecuentemente saber los costos de sus operaciones de busqueda insercion y borrado. Finalmente tambien puede decirse que al mantener la coherencia interna y hacer valer las propiedades que impone el contexto de uso, el invariante como precondicion de una estructura de representacion nos protege de inicializaciones inadecuadas de la estructura y limita el dominio de la funcion de abstraccion.

Tomás Caballero & Paulo Ballan: El rol que juega el invariante de representación a la hora de realizar el diseño de un TAD radica en el "sentido" y la validez que tiene la estructura a diseñar. También nos puede asegurar propiedades en la estructura para mantener complejidades determinadas. Es un predicado que garantiza que las distintas funciones aplicadas a la estructura terminen correctamente. Todas las operaciones del módulo deberán preservar el invariante de representación.
Por ejemplo si diseñamos a partir de un TAD de una lista ordenada, uno de los puntos que debería cubrir el inv rep es que cada elemento sea menor o igual a su consecuente.

Nicolás M. Obesio:
- Inv. Rep es un predicado que nos indica si una instancia del **tipo de representación** es valida para representar una **instancia del tipo**.
- Debe preservarse luego de aplicar cualquier funcion exportada.
- Fuerza a la estructura a mantenerse **coherente** y en muchos casos a que respete las **restricciones del contexto de uso**. (Ej: Que mantenga balanceo en altura para un AVL)



Min Heap

Max Heap

Decir si es V o F (justificar o dar contraejemplo)

-Sea S arreglo de claves representado por max-heap. Sean S[i] y S[j] claves del heap / i < j y S[i] < S[j] → el arreglo obtenido intercambiar S[i] y S[j] sigue siendo max-heap.
-Sea S arreglo de claves representado por max-heap. Sean S[i] y S[j] claves del heap / i < j y S[i] > S[j] → el arreglo obtenido intercambiar S[i] y S[j] sigue siendo max-heap.

Teo Freund:
-Falso) S=[8,5,7,4,3,6], indexado desde 1, i=4, j=6, i<j. S[i]=4<6=S[j] => S'=[8,5,7,6,3,4], pero 5<6, y 5 es el padre de 6, no se cumple la propiedad max-heap.
-Falso) S=[8,7,5,6,3,4], indexado desde 1, i=4, j=6, i<j. S[i]=6>4=S[j] => S'=[8,7,5,4,3,6], pero 5<6, y 5 es el padre de 6, no se cumple la propiedad max-heap.

Explique la relación entre invariante de representación y complejidad algorítmica, y entre función de abstracción y la demostración de que el diseño es correcto con respecto a la especificación.

Nicolás M. Obesio:
- Relación entre Inv. Rep y Complejidad Algoritmica
En muchos casos, si no se cumple el Inv. Rep es imposible garantizar complejidades. (EJ: Balanceo en altura en AVL que nos permite garantizar busqueda logaritmica)
(También podriamos mencionar que nos permite garantizar complejidades espaciales al no permitir cosas como elementos repetidos pero no se si excede el scope)

- Relación entre Abs y Correctitud de la implementación
La función de abstracción es el nexo entre la implementación y la especificación. Si ella no podriamos trazar un paralelismo entre ambos mundos y por ende no podriamos probar correctitud.

Se desea implementar un TAD que permita representar un conjunto de números racionales, donde a las tradicionales operaciones de Agregar(S,x) y Borrar(S,x) se agrega la siguiente: CLOSERTOAVG(S), que toma como input un conjunto S y devuelve el valor contenido en S más cercano al promedio de los valores contenidos en S. Discutir la implementación de este TAD utilizando variantes de al menos 4 estrucuras de datos vistas en clase para representar conjuntos/diccionarios standard.

1) Tupla <arreglo, nat>. En el nat tengo el promedio y en el arreglo los elementos.
2) Búsqueda binaria con variante para ver diferencias entre el promedio y el número.
3) Encolo todo en un heap, cuando encolo saco el promedio, y cuando los saco voy fijandome cual me da la menor diferencia.

df

Responder Verdadero o Falso. Justificar.

a) Lo que hace que una operación sea un observador básico es que deba escribirse en base a los generadores.
b) Si una operación rompe la congruencia debe ser transformada en observador básico.
c) Dos instancias del mismo TAD pueden ser observacionalmente iguales y aun así ser distinguibles por una operación.
d) Si un enunciado dice "Siempre que vale A sucede inmediatamente B y B no puede suceder de ninguna otra manera" y la correspondiente axiomatización incluye las operaciones A y B entonces el TAD está mal escrito.

Carlos Giudice:
a) falso, los observadores son funciones de un TAD que toman como parametro a cualquier elemento pertenceciente al conjunto de instancias del TAD y lo mandan a una clase de quivalencia.
b) falso, si tus operaciones rompen la congruencia, escribilas bien. Bueno la pregunta es medio rara, lo que si podriamos tener en cuenta es que si para dos instancias onservacionalmente iguales del TAD hay una funcion que tiene resultados diferentes para cada instancia puede ser que esa funcion este dandonos informacion que realmente es un aspecto necesario al momento de modelar las caracteristicas relevantes de las instancias. En ese caso estaria bueno hacer que la funcion sea un observador.
c) **POLEMICO**, si son observacionalmente iguales, no deberia poder distinguirlos ninguna operacion. Sin embargo tenia entendido que se puede escribir "otras operaciones" con un hack al estilo pattern matching mediante el cual es posible distinguir instancias en base a su sintaxis y no su semantica, en ese caso tendriamos resultados diferentes aun para istancias que cumplen igualdad observacional.
d) falso, el TAD no estaria mal escrito, simplemente las funciones A y B serian equivalentes.

Ivan Gk: Carlos, me parece que en la C estas distinguiendo entre semantica observacional y, lo segundo que decis, semantica inicial. Yo supongo que la pregunta esta orientada a semantica observacional, por eso me parece correcta la respuesta de Nico abajo. En la pregunta D también estoy de acuerdo con Nico: fijate que dice que si sucede A sucede B, y no puede suceder B de otra forma, y luego te dice que tenes una operación para A y otra para B, lo cual infringe lo primero, que es comportamiento automatico. Es decir, B pasa sólo, una vez que pase A, pero no puedo forzar a que pase B, si no paso A.

Esto siempre se cumple: a =obs b => OO(a) =obs OO (b), por lo q C es Falso

Nicolás M. Obesio:
a- Falso. Es legal axiomatizar otras operaciones en base a los generadores. (Si bien es una practica que se desalienta)
b- 'Verdadero'. Si una propiedad distingue TADs observacionalmente iguales quiere decir que esta observando alguna propiedad que no observa los observadores basicos por lo que se la podria agregar. (Tambien esta el argumento de Carlos que es cierto. Esta pregunta necesitaria algo mas de contexto.)
c- Falso. Se rompería la congruencia si eso pasara.
d- Verdadero. B es comportamiento automatico.

Nico Pazos:
d) Para mi el enunciado es medio ambiguo. No se entiende si es una operación que devuelve una instancia del tipo o es, por ejemplo, un booleano (por ejemplo una operacion que sea EsPar y otra EsMultiploDeDos). Segun la mirada ambas respuestas que dieron me cierran.

Es verdadero, B es comportamiento automatico, confirmado por ayudantes

Tomás Caballero y Paulo Ballan:

a) Hacemos heapsort con una pequeña variacion. Primero hacemos heapify que nos costaria O(n) con floyd. Cuando terminamos heapify vamos desencolando y sumando los primeros K que desencolamos para ver si cumple o no la condicion. Esto nos da una complejidad de O(n) con el heapify + O(k*log n) en el desencolamiento. Con lo que nos queda O(n*log n). salu2.

b) Hacemos mergesort acotando la parte que sin agregados nuevos, cuando terminamos de hacer mergesort con esta misma, hacemos el merge con el "pedazo" nuevo. Así hasta el infinito.

En cada uno de los siguientes escenarios, indique qué método de ordenamiento de los estudiados en clase utilizaría y porque.

a) Se tiene un arreglo de naturales A[1..n] y se desea ordenarlos (de mayor a menor) solamente si la suma de los k elementos más grandes es menor que X. Se desea realizar ésta tarea de forma eficiente, dandola por terminada lo antes posible si la condición no se cumple. (la verificación forma parte de la tarea, no se debe verificar la condición antes de empezar a ordenar)
b) Se tiene un arreglo redimensionable de naturales A[1..n] y se desea ordenarlos. Sin embargo, durante el proceso de ordenamiento es posible que se agreguen nuevos elementos al final del arreglo.

Carlos Giudice:
a) Hacemos selection sort, en las primeras k iteraciones, sumamos el valor encontrado a la variable acum (que inicializamos en 0 al principio del programa) si acum > X, tiramos fatal error.
b) este me parecio medio raro porque es como que tendria que haber un interrupt o algo asi pero bueno no importa, la cuestion seria ir haciendo insertion sort y que inserten los nuevos elementos en el vector de no ordenados, entonces no importa que elemento hayan metido, nosotros seguimos metiendo cada cosa en su lugar. Fe de erratas: antes habia escrito insertion sort en el primero pero es selection sort, como dijo el cumpañero Naxio.

E. Gambaccini:

a) Max heapify del arreglo, sumo los k primeros, y en base a eso decido si ordenar

Nicolás M. Obesio:
a- Selection Sort ya que el criterio es terminar la ejecución lo antes posible si la condición no se cumple.
b- Ordenamos los nuevos elementos con algun algoritmo rapido y hacemos un merge con los que ya estaban ordenados.

Ivan Gk: Nico, nose si esta bien tu respuesta b, ya que te dice que te dan los nuevos elementos mientras estas ordenando los de A. Por esto, yo usaria insertion sort, que es online, como dijo Charly arriba.

a) Dar la propiedad que hace que los ABB sean AVLs. O sea, el invariante. (en castellano)
b) Dar un algoritmo lineal que verifique si un ABB cumple con el invariante del AVL. Justificar la complejidad obtenida.
c) Mostrar un ejemplo de AVL donde el borrando genera más de una rotación.

Carlos Giudice: a) para que un ABB sea AVL tiene que cumplirse que para todo nodo del ABB tiene que cumplirse que: |altura(subArbolIzq) - altura(subArbolDer)| <2

Sebastian Bocaccio:b) Para que sea Avl tengo que chequear que sea arbol binario de busqueda y que la rama mas larga izquierda y derecha difiera en un factor a lo sumo + - 1 (eso para cada nodo)

```
// Chequeo que sea arbol binario.
esAvl1(raiz r){

    // Primero chequeo que sea arbol binario.
    if(tieneHijoIzquierdo(r) && Izq(r) > r ){return false}
    else if(tieneHijoDerecho(r) && Der(r) < r ){return false}

    //Chequeo que no haya problema con los hijos
    return (esAvl1(Izq(r)) && esAvl1(Der(r)))

} O(n)
Esta funcion es O(1) para cada nodo y se llama para cada nodo. O(n)

// Me fijo que este balanceado
<int altura, bool valor> esAvl2(raiz r){

    if(soyHoja(r)){return <0,true>}
```

Franco Lancioni: c) Como el balaneceo RR o LL no disminuyen la altura total del subárbol balanceado, había pensado algo así como poner un subárbol que fuerce un LR (si baja la altura al eliminar) con altura h en un subárbol y otro subárbol de altura h+1 unidos por una raíz. Al terminar el balanceo LR, un subárbol tiene h-1 de longitud y el otro h+1 -> hace falta otro balanceo

RESUMIDO: avl?(in: nodo a, out: nat altura) -> bool { nat altIzq, altDer; bool i = avl?(a.izq, altIzq); bool d = avl?(a.der, altDer); altura = 1+max(altDer, altIzq); res = i && d && |altIzq-altDer|<2}

a) ¿Cuántos caracteres DISTINTOS tiene que tener un texto como mínimo para que alguno de ellos reciba un código de longitud k en una codificación de Huffman (con k>1)? Jusifícar.
b) ¿Cuántos caracteres en total tiene que tener un texto como mínimo para que algún caracter reciba un código de longitud k en una codificación de Huffman (con k>1)? Justificar
c) Responder a) y b) para códogs de longitud fija. Justificar.

Nicolás M. Obesio:
a- Balanceo en Altura.
b- Escribir el algoritmo naive y hacerlo lineal usando inmersión.
c -

Carlos Giudice:
a) el algoritmo de huffman toma un conjunto de caracteres y toma los dos que menor frecuencia tienen, creando dos nodos hijos de un tercer nodo que se considera un nuevo caracter cuya frecuencia es la suma de las frecuencias de sus dos hijos, este nuevo "caracter" compuesto reemplaza a sus hijos en el conjunto se repite esta tarea hasta que el conjunto de caracteres es vacio. si ocurre que el caracter compuesto siempre pertenece a los dos caracteres de menor frecuencia armamos un arbol binario totalmente degenerado, en este caso la altura es maxima para la cantidad de caracteres iniciales. Entonces seria necesario empezar con al menos k+1 caracteres para tener un arbol de huffman de altura k, lo que implicaria que los dos primeros caracteres que tomamos reciben codigos de longitud k.
b) sumatoria(fibonacci(k)) justo la funcion fibonacci nos sirve para calcular las frecuencias ascendentes que necesitamos para crear un arbol de huffman degenerado :D
c) en ambos casos vamos a tener arboles de huffman perfectamente balanceados, osea que su altura h va a ser log2(#nodos)+1 y #nodos=caracteres+nodosInternos y por propiedades de abb perfectamente balanceado #nodos=2*caracteres-1. hacemos matematica y vemos h = log2(2*caracteres-1)+1. Nosotros queremos que h sea k+1, entonces planteamos k+1 = log2(2*caracteres-1)+1 <===> k = log2(2*caracteres-1) <===> 2^k=2*caracteres-1 <===>(2^k+1)/2=caracteres distintos. podemos decir que en el caso del texto podrian haber x ocurrencias de caracteres, por lo tanto tiene que tener un multiplo de esa cantidad.

Considerar una tabla hash con direccionamiento abierto, en la cual se marca de forma diferenciada las posiciones que contienen un elemento, aquellas que nunca contuvieron un elemento y aquellas que en algún momento tuvieron un elemento pero fue borrado. Describir los algoritmos de inserción, búsqueda y borrado y analizar sus complejidades asintóticas.
Comparar con la versión de hashing en la cual no se puede distinguir los borrados de "vacio".

Nicolás M. Obesio:
Hash Table con Direccionamiento Abierto: Todos los elementos se guardan dentro de la tabla.

Definimos esta ocupado = No esta vacio y no fue borrado

Insercion(clave, significado, tabla)
i=0;
mientras (tabla[hash(clave, i)] este ocupado && i<|tabla|)
  i++;
si (i < |tabla|):hacer
  tabla[hash(clave, i)] = (clave, significado)
  marcar como ocupado tabla[hash(clave, i)]
sino:
  <OVERFLOW>

Busqueda(clave, tabla)

mientras (i < |tabla| && [hash(clave, i)] este ocupado o haya sido borrado && tabla[hash(clave, i)].clave != clave):
  i++;
Si (i < |tabla| && tabla[hash(clave, i)] esta ocupado:
  devolver tabla[hash(clave,i)]
else:
  <NO ESTA>

Borrado

mientras (i < |tabla| && [hash(clave, i)] este ocupado o haya sido borrado && tabla[hash(clave, i)].clave != clave):
  i++;
Si (i < |tabla| && tabla[hash(clave, i)] esta ocupado:
  marcar como borrado tabla[hash(clave, i)]
else:
  <NO ESTA>

Comparación
En la version que no se puede distinguir el borrado del vacio hay que correr todos los elementos para que no queden huecos y no romper la busqueda.

Explicar por que las colas de prioridad son ineficientes para realizar búsquedas pese a ser un árbol balanceado y por qué no se pueden modificar para que lo sean sin perder una de sus propiedades fundamentales.

Nicolás M. Obesio:
La estructura preferida para implementar Colas de Prioridad son los Heaps.
a- Los Heaps, en particular, no son Arboles Binarios de Busqueda, por lo tanto, no se puede hacer "busqueda binaria" (O(log n)) y debemos hacer una busqueda lineal(O(n)) .
b- No se pueden modificar por que la propiedad de Heap resulta incompatible con la de los ABB.

Nico Pazos:
b) (continuando lo de mi tocayo)... pues en los ABB los nodos por debajo de un nodo particular son mayores o menores dependiendo de si están en su subárbol derecho o izquierdo. En un heap, todos los que están por debajo son menores (si es max-heap), sin importar el subárbol.

Detallar el criterio de balanceo de los árboles B y mostrar cómo se mantiene a través de la inserción y el borrado.

Nicolás M. Obesio:
Un Arbol-B, de orden M, tiene las siguientes propiedades:
- La raiz es una hoja o sino tiene entre 2 y M hijos.
- Los nodos que no son hojas (Excepto la raiz) tienen entre M/2 y M hijos.
- Todas las hojas estan a la misma altura.
Con estas propiedades nos aseguramos que el arbol este balanceado.

Insertar:
- **Si el nodo esta lleno:** Insertamos la clave en su posición y puesto que no caben en un único nodo dividimos en dos nuevos nodos conteniendo cada uno de ellos la mitad de las claves y tomando una de éstas para insertarla en el padre(se usará la mediana).Si el padre está también completo,habrá que repetir el proceso hasta llegar a la raíz. En caso de que la raíz esté completa,la altura del árbol aumenta en uno creando un nuevo nodo raíz con una única clave.
- **Si el nodo NO esta lleno:** Agregarlo en el nodo.

Eliminar
- Si al borrar la clave, el nodo se queda con un numero menor que el minimo y uno de los hermanos adyacentes tiene al menos uno más que ese mínimo: **REDISTRIBUYO**
- Si no puedo redistribuir: **Unión**. Unir los nodos junto con la clave que los separa y se encuentra en el padre.

Emiliano G.: Related: Arboles 2-3-4: https://es.wikipedia.org/wiki/%C3%81rbol_2-3#.C3.81rbol_2-3-4

El invariante de representación suele escribirse de manera formal y eso permite utilizarlo para una serie de cosas. Si se escribiese en castellano, ¿cuáles de esas cosas se podrían seguir haciendo y cuáles no? Justifique.
Si el invariante de un tipo resultase programable, ¿lo haría? ¿para qué lo usaría? Justifique.

Ivan Gk: Se podria seguir haciendo todo lo que se hace, salvo que sería mucho mas engorroso, ya que el lenguaje natural es muy ambigüo, y por esto habría que explicar mucho mas que en lógica de primer orden. Sería util programarlo: nos permitiría comprobar que una instancia del tipo es correcto antes de utilizarlo en un algoritmo. Sin el rep programado, corremos el riesgo de que pasen instancias invalidas del tipo a los metodos que programamos.

Nicolás M. Obesio: (Creo que esperan esto)

Final de 13/12/16
1)Se pide construir una doble cola de prioridad que satisfaga:
a) min y max en O(1)
b) desencolar min y max en log(n)
c) borrar min, borrar max en log(n)
d) ingresar m elementos en O(min{ n + m, m log(n) })

Juan Gonzalez: Creo que simplemente esperan tener dos Heaps en donde ambos tienen los mismos elementos sólo que uno cumple el invariante de max heap y otro el de min heap. El borrado en esos casos sería 2log(n) (porque debo borrarlos de ambos heaps). Por otro lado, cuando se ingresan m elementos habria que hacer una cuenta entre que conviene más, si ingresar los m en los dos heaps, lo cual costaría O(2*m*log n) o ingresar simplemento los elementos en el arreglo (O (m + n)) y reamrar ambos heaps con el algoritmo de Floyd. Esto ultimo costaría O (m + n) Mati Sandacz: Creo que la interpretación que estas dando está mal. A medida que vas insertando los m elementos, el tamaño del heap se va ir pareciendo cada vez mas a O(m+n), por ende insertar m elementos en el peor caso va a costar O(m log(n+m)) y no O(m log n) como se detalla en la solución.

Pedro Boitier:
Armo 2 heaps, uno max y otro min, sobre arreglo<tupla>. Donde estas tuplas tienen <elemento, indiceOtroHeap>. El indiceOtroHeap representa la posición en el otro heap para poder mantener actualizados ambos heaps. Todo esto cierra con los primeros 3 requisitos. El 4to requisito lo fuerzo poniendo un if adelante que vea si rearmando los dos heaps (O(2(n + m))) es menos costoso que agregarlos al existenten (O(m*log n)).

Final de 13/12/16
2) Explicar la interfaz y estructura de representación de un modulo que implemente el TAD conjunto ordenado. Proveer las funciones necesarias para garantizar todas las cosas necesarias en tiempo eficiente. Insertar, buscar y borrar en O(log(n)). Recorrer de forma iterativa todo en tiempo lineal. No olvidar conceptos de aliasing y de argumentos.

Final de 13/12/16
3) Analizar los siguientes peores casos. Si se puede utilizar el teorema maestro, utilizarlo. Justi

a) T(N)= 4 T(N/2) + 3 N^2
b) T(N)= 16 T(N/4) + (N^2)/log(N)
c) T(N)= 2^N T(N/8) + 1
d) T(N)= 3 T(N/2) + N^2 log(n)
e) T(N) = 16 T(N/2) + F(N) log(n)
donde F(N)= N^3 (si n par) N^2 (sino)

f) T(N) = 3 T(N/2) + F(N)
donde F(N)= N^3 (si n par) N^2 (sino)

Juan Gonzalez: c) No aplica porque a no es una constante. s

a) Caso 2 del teorema maestro, N^2 log(n)
b) Teorema maestro no aplica, la diferencia entre N^log4(16) y N^2/log(N) no es polinomial.
c) a no es constante, el teorema maestro no aplica.
d) Caso 3 del teorema maestro, N^2 log(n) pertene a omega(N^2) y c = 3/4.
e) N^logb(a) = N^4, luego, acotando F(N) por O(N^3), F(N) pertenece a O(N^3.5) => T(N) = theta(N^4)
f) Entraria en el 3er caso del teorema maestro, el problema es no hay theta(F(N)), ya que F(N) es O(N^3) y omega(N^2)

Final de 13/12/16
4) Se brinda un TAD Conjunto especificado con los siguientes cambios significativos:

Obs:
	secu()

Igualdad Observacional:

 c =obs c' <=> secu(c) =obs secu(c')

Otras Operaciones:

oneOff(c) = prim(secu(c))
A) Explicar si está o no bien especificado, y si es correcto respecto a la especificación de la cátedra.
B) suponer correcto. ¿Qué aspectos te parecen mejores o peores?
C) Justificar si puede demostrar esto:
Ag(n, Ag(n_1, ... (Ag(n_k, vacio())...)) =obs Ag(m, Ag(m_1, ... (Ag(m_k, vacio())...))
<=>
Add(n, Ag(n_1, ... (Add(n_k, vacio())...)) =obs Add(m, Add(m_1, ... (Add(m_k, vacio())...))
Donde Add es el generador que reemplaza el comportamiento de Ag en la nueva especificación.

D) DameUno(Ag(n, Ag(n_1, ... (Ag(n_k, vacio())...))) =obs oneOff(Add(n, Ag(n_1, ... (Add(n_k, vacio())...))) Ubeda
Decir que rol juega la función de abstracción en la correctitud de un diseño con respecto al TAD.

Sebastian Bocaccio:
int INFINITO = 999999;

```
bool printearEnOrden( std::vector<int> arreglo){
	int minActual = INFINITO;
	int minAnterior;
	bool primerCambio = true;
	int cantMinimoActual;
	for ( int j = 0 ; j < arreglo.size(); j++){
		cantMinimoActual = 0;
		for( int i = 0 ; i < arreglo.size(); i++){
			if(primerCambio){
				if (arreglo[i] < minActual){
					cantMinimoActual = 1;
					minActual = arreglo[i];
				}
				else if (arreglo[i] == minActual){
					cantMinimoActual++;
				}
			}
			else{
				if (arreglo[i] <= minActual &&  arreglo[i] > minAnterior){
					if( arreglo[i] == minActual){
						cantMinimoActual++;
					}
					else{
						cantMinimoActual = 1;
						minActual = arreglo[i];
					}

				}

			}

		}
		for( int i = 0 ; i < cantMinimoActual; i++){
			cout<< minActual;
		}
		minAnterior = minActual;
		minActual = INFINITO;
		primerCambio = false;
	}
```

Para explicar porque A tarda más que B podemos pensar que si bien los datos vienen dados de forma uniforme, existe una probabilidad alta de que en algún lugar del árbol no haya quedado balanceado pero que aún así el tiempo de búsqueda y agregado sea logarítmico, aunque un poco distorsionado en algunos sectores.

Por otro lado, lo que tienen las tablas de hash es que su tiempo de búsqueda depende mucho sobre la capacidad de crear uniformidad de su función de hash (que en este caso suponemos funciona bastante por cómo es la distribución) y cuanto espacio empleamos para la tabla. Considero que en este caso la tabla tiene un tamaño tal que permitía realizar el insertado y búsqueda en la mitad del tiempo del que requería en el ABB.

Por último, creo que la amplia diferencia que se encuentra con la forma C radica en la utilización de quicksort para ordenar, el cual se sabe que funciona muy bien en muchos casos si bien en peor caso es O(n^2). Además, el emplear una estrategia tipo mergeSort aporta mucho ya que la complejidad de esto es la de recorrer los dos vectores en peor caso O(|S| + |T|). Esto funciona más eficientemente que para cada elemento de T buscarlo en el árbol S' ya que la complejidad de encontrar la intersección es O(|T|* log(|S|)).

Con respecto a la diferencia con la tabla de hash, evidentemente al tardar la mitad del tiempo en decir si un elemento está o no (como esta dicho arriba) es normal que este método tarde la mitad ya que |S| es significativamente más chico que |T| por lo que la búsqueda de los elementos de T requirió más tiempo de cómputo que el ordenar S.

Si |S| = 10^3, entonces la tabla de hash tendría muchas menos colisiones (suponiendo que mantiene el tamaño) por lo que se remarcaría mucho más la diferencia con el árbol ABB aunque la cantidad de niveles del árbol también haya decrecido. Con respecto al método C, si bien se acelera el ordenado de S, en el peor caso buscar las colisiones sigue manteniendo la misma complejidad O(|S| + |T|) por lo que siento que la diferencia del tiempo con respecto a los otros métodos se achicaría proporcionalmente

Si |S| = 10^5, entonces la tabla de hash tiene muchas más colisiones (suponiendo que mantiene el tamaño) por lo que podría arriesgar a decir que el método de ABB podría hasta ser mejor o igual en cuestiones de tiempo. Creo que en el método C el quicksort mostraría una amplia mejora antes los otros dos métodos, lo que le otorga a C un buen comienzo. Pero también, como la complejidad de la unión no se ve afectada en peor caso porque |S| = |T| pero si se ve afectada para los métodos A y B creo que para este tamaño de S la diferencia sería aún más remarcable.

Final 20/7/18

1) Se tiene una secuencia const (su contenido no puede ser alterado). Poponer un algoritmo que la muestre en orden (asc o desc). El algoritmo debe tener complejidad espacial constante (O(1))

Tenes dos vectores S y T con numeros aleatoriamente elegidos entre 0 y 10^6. |S| = 10^4 y |T| = 10^5, T viene ordenado. Queremos hacer la interseccion de ambos. Probamos 3 formas:

a) Metes S en un conjunto S' sobre ABB balanceado y despues por cada uno de T, ves si esta en S'.

b) idem a pero con una tabla de hash con encadenamiento en vez de ABB balanceado.

c) Ordenas S (ponele que con quicksort, algo n*lg(n)) y ves la interseccion entre S y T con un algoritmo estilo merge.

Luego de las pruebas, sabemos que el a tardo el doble que el b y el b tardo 4 veces que el c. Explicar por que, y conjeturar que pasaria si |S| = 10^3 y si |S| = 10^5.

Porque d