Árboles rojo-negros con $n > 1$ nodos

a) Las 4 propiedades que definen a un **árbol rojo-negro** permiten, al menos en teoría, que el árbol tenga únicamente nodos negros. Si efectivamente tuviéramos un árbol rojo-negro tal que todos sus nodos fueran negros, ¿qué condición particular cumpliría este árbol en cuanto a su "forma" y a su cantidad n de nodos, y por qué?

Resp.

La propiedad, o regla, 4 (p.14 de las diapos. de la clase 11) implica que todo camino desde la raíz hasta una hoja (nula) tenga la misma cantidad de nodos negros (no nulos).

Así, si el árbol tiene un solo nodo, éste puede ser negro (de hecho, tiene que ser negro, porque al ser el único nodo es también la raíz, y la propiedad 2 exige que la raíz sea negra). [**No da puntaje, porque hay que considerar el caso $n > 1$.**]

El árbol **no puede** tener sólo dos nodos, ambos negros, ya que la cantidad de nodos negros desde la raíz hasta su hijo no nulo sería distinta (mayor) que la cantidad de nodos negros desde la raíz hasta su hijo nulo;

... pero sí podría tener sólo tres nodos, todos negros, en la forma de una raíz con dos hijos. [**1 pt.**]

La única forma en que este último árbol, de tres nodos negros, pueda tener más nodos, todos negros, es que cada uno de los dos hijos tenga a su vez dos hijos negros.

Siguiendo con esta argumentación, la "forma" de un árbol rojo-negro que sólo tuviera nodos negros sería la de un árbol completamente lleno en todos sus niveles [**1 pt.**]; si el árbol tiene k niveles, entonces tiene $n = 2^k - 1$ nodos [**1 pt.**].

b) Más allá de la posibilidad de a), ¿qué ocurre en la práctica con un **árbol rojo-negro**, con $n > 1$ nodos, construido sólo mediante inserciones que se hacen siguiendo exactamente el procedi-miento estudiado en clases? Argumenta de manera rigurosa que en este caso el árbol siempre va a tener al menos un nodo rojo.

Resp.

[**1 pt.**] Un nodo recién insertado se pinta siempre inicialmente de rojo; si el padre es negro, entonces el procedimiento de inserción termina ahí, y por lo tanto el árbol tiene al menos este nodo (recién insertado) rojo.

[**1 pt.**] En cambio, si el padre es rojo, miramos el color del tío. Si el tío es negro, entonces hacemos una o dos rotaciones (dependiendo de si el nodo fue insertado "por afuera" o "por adentro"), y final-mente cambiamos los colores de un padre rojo y de su hijo negro, dejándolos negro y rojo, res-pectivamente. De nuevo, hay al menos un nodo rojo.

[**1 pt.**] Si el tío es rojo, entonces cambiamos los colores del padre y tío (ambos inicialmente rojos) y del abuelo (inicialmente negro), pero mantenemos rojo el nodo recién insertado. Si bien en este caso los cambios de colores pueden replicarse hacia arriba, el nodo insertado se mantiene rojo.

Tablas de hash

a) h(a) = 5, h(b) = 2, h(c) = 2, h(d) = 8, h(e) = 1, h(f) = 1, h(g) = 5, h(i) = 2

b) Queremos saber cuáles son las palabras más frecuentes que aparecen en la novela *Fortunata y Jacinta*, de Benito Pérez Galdós. Explica cómo responderías esta pregunta eficientemente usando **tablas de hash**, teniendo presente las siguientes condiciones: *Fortunata y Jacinta* es una de las novelas más extensas de la literatura española; contar "a mano" no sirve. No sabemos cuántas palabras tiene la novela, ni tampoco cúantas palabras distintas tiene. Dispones de una función, *f*, que recibe un string de caracteres y devuelve un número entero en un rango muy grande. Según el método de resolución de colisiones que elijas usar, y para que el desempeño de la tabla sea eficiente, el factor de carga no puede ser mayor que 1/3 en un caso, o mayor que 3 en el otro caso (tú tienes que saber cuál es cuál). Específi-camente, explica:

a) ¿Cómo hay que ir ajustando el tamaño *m* de la tabla a medida que se van procesando más y más palabras de la novela? La idea es que se respete el límite superior para el factor de carga, pero también es importante no desperdiciar memoria como consecuencia de una tabla demasiado grande.

Resp. La idea básica es definir un *m* inicial, más o menos arbitrariamente, e ir ajustándolo —aumentándolo a aproximadamente el doble— mediante rehashing cada vez que sea necesario, es decir, cuando el factor de carga λ se aproxime mucho, o sobrepase levemente, el valor 3 (si usamos resolución de colisiones mediante encadenamiento).

b) Exactamente, ¿cómo se procesa una palabra? Es decir, ¿qué debe hacer tu programa cuando lee la próxima palabra? ¿Cómo se usan la función *f* y el tamaño *m* de la tabla?

Resp. Definimos una tabla de hash *T* de tamaño *m*, inicialmente vacía; un contador *n* de palabras distintas almacenadas en *T*, con valor inicial 0; y un factor de carga λ, también con valor inicial 0. Cuando se lee la próxima palabra *s* de la novela, se calcula su valor de hash, *h*, y se mira la tabla, en particular la casilla *T*[*h*]. Si *T*[*h*] está vacía, entonces se agrega *s* a *T*[*h*], con *s.count* = 0, se incrementa *n*, y se re-calcula λ. Si *T*[*h*] no está vacía y una de las palabras almacenadas allí es *s*, entonces se sólo incrementa *s.count*.

Los *m*'s deberían ser números primos (por eso el aumento en la letra a) no es exactamente al doble), en particular si la función de hash consiste en aplicar la operación módulo *m* al número entregado por la función *f* para la palabra —el string— que se está procesando.

c) ¿Cuándo, o bajo cuáles condiciones, se produce una colisión, y cómo se resuelve?

Resp. Si cuando se mira *T*[*h*], esta celda no está vacía, pero ninguna de las palabras almacenadas en la lista ligada que empieza ahí es *s*, entonces hay una colisión. Entonces, hay que agregar *s* a la lista ligada, inicializar *s.count* = 0, incrementar *n*, y re-calcular λ.

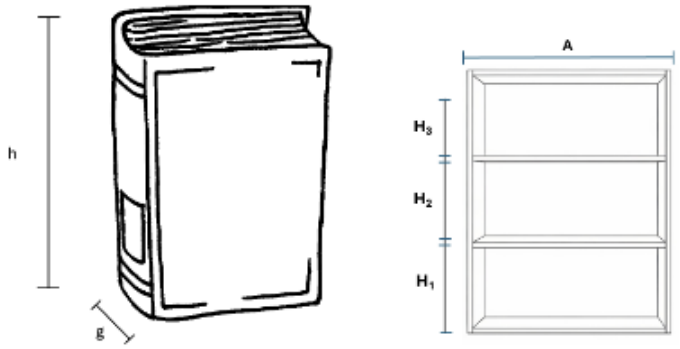
d) Una vez que tu programa termina de procesar todas las palabras de la novela, ¿cómo lo hace para responder la pregunta inicial: cuáles son las palabras más frecuentes?

Resp. Una posibilidad es declarar un array *R* de tamaño *n* (el número de palabras distintas), y recorrer la tabla *T* y almacenar en cada casilla de *R* una palabra *s* distinta junto a su *s.count*. Luego, hay que ordenar *R* de mayor a menor de acuerdo con los s.count; y finalmente hay que imprimir los contenidos de las primeras casillas de *R*.

Otra posibilidad es usar un heap —un array de tamaño igual al número de palabras más frecuentes que nos interesa— que se va actualizando cada vez que se procesa una nueva palabra.

3. Técnicas Algorítmicas

Bob acaba de comprar una repisa de libros de 3 pisos de diferentes alturas H_1, H_2, H_3 cms y de A cms de ancho y con una capacidad máxima de W kg. Los libros se deben acomodar en forma vertical, es decir, con su grosor ocupando espacio en el ancho de la repisa.
Además, cuenta con un conjunto de libros a ser acomodados en la repisa. Cada libro L_i se caracteriza por su altura h_i cms, su grosor g_i cms y su peso w_i kg.
Suponga que todos los libros caben de alto en alguna repisa y que cada piso de la repisa puede acomodar a varios libros.



En los siguientes puntos, seleccione el uso de una estrategia algorítmica distinta (es decir, no puede usar la misma técnica en dos puntos distintos) entre Backtracking, Codiciosa o Programación Dinámica que mejor se ajuste para resolver cada uno de los siguientes problemas. En cada caso debe plantear las restricciones, la función objetivo, la ecuación de recurrencia o la estrategia codiciosa según corresponda, describir la solución y justificar la elección realizada.
Solución: Las restricciones generales son:
NO PUEDEN REPETIRSE TÉCNICAS

- Restriccion de largo por piso: $\sum_{i=1}^{n} g_i <= A$ para cada **piso** y $g_(i+1) > L$
- Restriccion de peso en repisa completa: $\sum_{i=1}^{n} w_i <= W$ y $\sum_{i=1}^{n} w_{i+1} > W$ para la repisa completa
- Resricción de altura por piso $h_i <= H_1, H_2, H_3$

a) (2 pts.) Asignar libros a pisos de la repisa de modo que se cumpla con todas las condiciones.
Solución:1 Backtracking, solo se necesita una solución cualquiera que cumpla que la suma de los pesos de los libros acomodados: Se eligen los L_i y se acomodan de modo que se cumplan las 3 restricciones.
Solución:2 Programación dinámica

b) (2 pts.) Distribuir libros para maximizar el número total de libros ubicados. Sin exceder ancho ni peso soportado por la repisa.
Solución:1 Greedy
estrategia codiciosa puede ser escoger por densidad peso/grueso o escoger por los libros más livianos primero hasta alcanzar el máximo en cada piso
Solución:2 Backtracking con heurística

c) (2 pts.) Ordenar libros para minimizar el espacio sobrante en cada piso sin exceder el peso ni ancho.
Solución: Programación dinámica
Es similar al problema de la mochila
El espacio disponible después del libro L_i es d[i][j] por cada piso [j] La ecuación de recurrencia es tiene

4 opciones: colocar el libro en Piso1, Piso2, Piso3 o no colocar el libro. d[i]= A-max($A - \sum_{i=1}^{n} g_i$,A) con j=1,2,3

4. Grafos

Bob colecciona libros raros (los guarda en la repisa de la Pregunta 3) y descubrió recientemente un diccionario escrito en un idioma desconocido que utiliza los mismos caracteres que el idioma castellano. Sin embargo, el orden de las palabras en el diccionario es diferente de lo que uno esperaría si los caracteres estuvieran ordenados de la misma manera que en castellano. Bob intentó determinar el orden de los caracteres del extraño alfabeto usando la lista ordenada de palabras del diccionario. Frustrado por la tediosidad de la tarea, se dio por vencido y decidió pedirle ayuda a su amiga Alice, que modeló el problema usando grafos y fue capaz de resolverlo de inmediato. En esta pregunta, usted deberá replicar el proceso con el que Alice resolvió el problema, diseñando un algoritmo tal que dada una lista de palabras ordenada alfabéticamente (usando el orden desconocido de los caracteres), permita obtener una lista ordenada con los caracteres del alfabeto.
Ejemplo: Para la siguiente lista ordenada de palabras

```
XWY
ZX
ZXY
ZXW
YWWZ
```

el orden de los caracteres que la produce es `X` $<$ `Z` $<$ `Y` $<$ `W`.
Asumiendo que la lista ordenada de palabras contiene suficiente información para obtener el orden buscado, responda los siguientes puntos.

a) (2 pts.) Modele el problema usando un grafo, de tal manera que a partir del mismo pueda obtener el resultado pedido (es decir, el orden de los caracteres). Indique el tipo de grafos que va a usar (dirigido, no dirigido, cíclico, acíclico) y a qué corresponde cada una de sus componentes.

Solución: A partir de los strings de entrada (y su orden) se pueden derivar relaciones de orden entre los caracteres. Se define un grafo $G = (V, E)$ tal que para todo carácter x del alfabeto, hay un nodo v_x en V (**0.8 puntos**). Luego, agregaremos aristas al grafo que muestren las dependencias entre caracteres que se pueden leer de los strings. Es decir, la arista dirigida (v_x, v_y) pertenece al conjunto de aristas E del grafo si y sólo si se puede deducir $x < y$ desde los strings de entrada (**0.8 puntos**). Dado que se asume que hay suficiente información en los strings, el grafo tendrá suficientes aristas que permitan obtener el orden alfabético pedido. El grafo es acíclico y dirigido (**0.4 puntos**), y por lo tanto admite un orden topológico, que es lo que nos dará el orden pedido (las aristas nos dan el orden entre pares de caracteres, un orden topológico nos da el orden completo. Es importante argumentar esto último en alguna parte del ejercicio, puede ser aquí o en otro de los puntos posteriores.

b) (2 pts.) Diseñe un algoritmo tal que a partir de la lista ordenada de palabras genere el grafo del punto anterior, y luego a partir del grafo permita obtener la lista ordenada de caracteres. **Solución:** Asumimos que el algoritmo recibe la lista ordenada de palabras P. El algoritmo es el siguiente

Algorithm 1: `SortAlfabeto`$(P[1..n])$

```
1  begin
2      V ← ∅
3      E ← ∅
4      s ← P[1]
5      for i = 2 to n :
6          s' ← P[i]
7          j ← 0
8          while i < s.len() and i < s'.len() :
9              if s[i] ≠ s'[i] :
10                 E ← E ∪ {(s[i], s'[i])}
11                 V ← V ∪ {s[i], s'[i]}
12                 break
13         s ← s'
14     Construir el grafo G = (V, E)
15     L ← TopSort(G)
16     return L
```

El **break** de la línea 12 funciona como en lenguaje C, rompiendo la ejecución del loop **While** de la línea 8. Si el grafo construido en la parte a) no es el correcto, tener en cuenta eso en este punto. Si el algoritmo es coherente con el grafo planteado en el punto a), asignar puntos parciales.

c) (2 pts.) Muestre el grafo correspondiente a la siguiente lista de palabras, y el resultado de aplicar su algoritmo sobre el mismo:

```
CEDA
CFA
CFD
CBE
AFE
AD
ABC
EAC
EE
EFF
EFD
```

Además de la lista ordenada de caracteres, muestre (gráficamente) el estado final de la ejecución de su algoritmo que le permite determinar el resultado obtenido.

Solución: el grafo para el ejemplo dado (**0.8 puntos**) tiene el conjunto de nodos $V = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\}$, mientras que el conjunto de aristas que se puede deducir de las palabras es $E = \{(\texttt{C}, \texttt{A}), (\texttt{A}, \texttt{E}), (\texttt{A}, \texttt{D}), (\texttt{E}, \texttt{F}), (\texttt{F}, \texttt{B}), (\texttt{F}, \texttt{D}), (\texttt{D}, \texttt{B})\}$. Luego de ejecutar el sort topológico, los vértices quedan marcados con los intervalos (**0.7 puntos**):

- `A` tiene el intervalo $[2, 11]$.
- `B` tiene el intervalo $[4, 5]$.
- `C` tiene el intervalo $[1, 12]$.
- `D` tiene el intervalo $[3, 6]$.
- `E` tiene el intervalo $[7, 10]$.
- `F` tiene el intervalo $[8, 9]$.

El sort topológico por lo tanto produce el orden: $\texttt{C}, \texttt{A}, \texttt{E}, \texttt{F}, \texttt{D}, \texttt{B}$. (**0.5 puntos**)