4. (1 punto) Implementa una función para determinar si un árbol binario tiene **más de un camino** desde una hoja a la raíz cuya suma de etiquetas sea igual a k.

0,25

bool suma_k(const bintree<int> &arb, int k)

5. (1 punto) Tenemos un contenedor de pares de elementos, {clave, bintree<int>} definido como:

0,75

```
class contenedor {
    private:
      map<string, bintree<int> > datos;
    ...
}
```

Implementa un iterador para iterar sobre los string de longitud 4 y cuyo bintree<int> tenga una estructura de árbol binario de búsqueda.

6. (1 punto) Un **APO sesgado** es un árbol binario equilibrado que cumple que para cualquier nodo Z la clave almacenada en Z es **menor** que la del hijo izquierda de Z y esta a su vez **menor** que la del hijo derecha (si lo tiene), y además sus hojas está empujadas a la izquierda. Implementa una función para insertar un nuevo elemento en el árbol y aplícalo a la construcción de un APO sesgado con las claves {29, 24, 11, 15, 9, 14, 4, 17, 22, 31, 3, 16}.

0,25

7. (2 puntos) Supongamos que dos personas (usuarios) entran en un **laberinto** con habitaciones con las siguientes características:
   - Cada habitación del laberinto tiene una puerta por donde se entra y dos puertas posibles por las que se sale.
   - Hay habitaciones que conducen a la calle y por lo tanto finaliza el recorrido por el laberinto.
   - Para pasar por una puerta se tira un dado y si sale un número par se sale por la puerta izquierda y si sale impar se sale por la derecha.
   - A cada habitación solamente se accede por una única habitación.

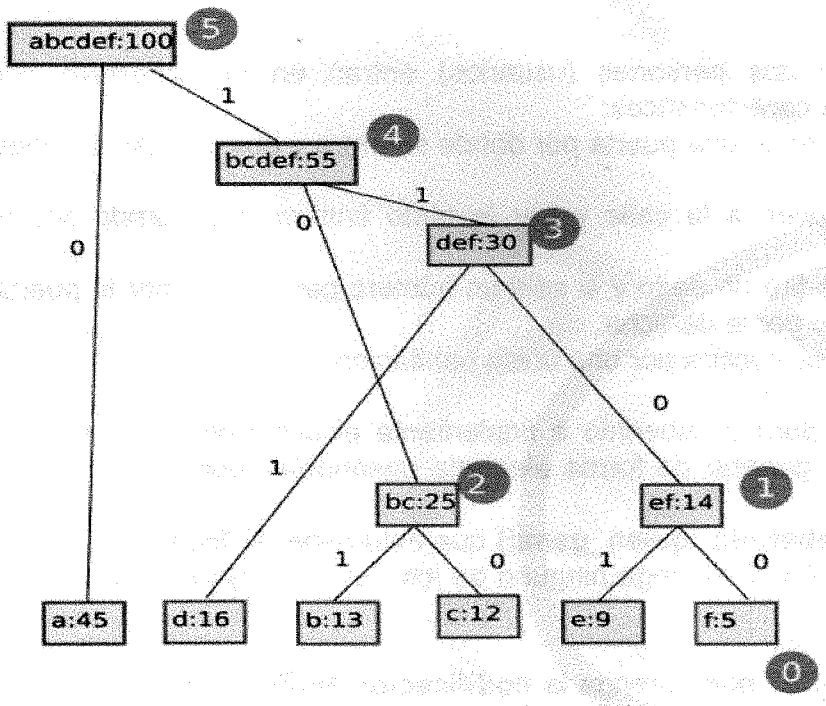
   Realiza las siguientes tareas:
   - Establece la representación para el laberinto e implementa el constructor de la clase laberinto. El laberinto se debe generar de forma aleatoria pasándole como parámetro el número de habitaciones.
   - Implementa una función **int laberinto::quien_gana()** que determine si llega primero a la salida el usuario 1, el usuario 2 o si no llega ninguno de los dos (en cuyo caso devuelve 0).

8. (2 puntos) Implementa una función que obtenga la **codificación Huffman** de un conjunto de caracteres. Este algoritmo codifica cada carácter en función del número de veces que aparece, asignando códigos más cortos a caracteres que aparecen más veces. En la imagen de más abajo se puede ver un ejemplo de su funcionamiento. Los pasos para obtener el código Huffman de un conjunto de caracteres, con las frecuencias de aparición de cada carácter son:

   - Se definen las parejas: {frecuencia, carácter asociado}. Dichas parejas frecuencia-carácter se insertan en una cola con prioridad. La más prioritaria debe ser la pareja con menor frecuencia.
   - Se sacan los dos elementos con menor frecuencia, se suman sus frecuencias y se añaden a su codificación los valores '0' y '1'. A continuación, se inserta un nuevo elemento en la cola con prioridad con frecuencia la suma de las frecuencias de los elementos, y con caracteres asociados la concatenación de los que se han sacado.
   - Si la cola sólo tiene un elemento, terminar, en otro caso volver a sacar las dos elementos con mayor prioridad y repetir los pasos anteriores.

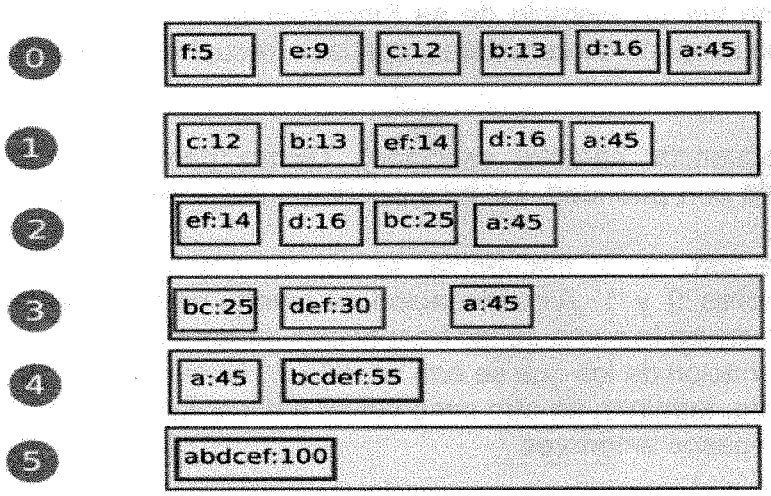
   Construye el árbol (o la estructura que se decida) asociado a la codificación y realiza las siguientes tareas:
   - Implementa una función que dada una cadena de caracteres devuelva el texto codificado.
   - Implementa una función que decodifique una cadena (formada por 0s y 1s)

   En la siguiente imagen se puede ver un ejemplo de codificación Huffman para los caracteres a, b, c, d, e, f con frecuencias 45,13,12,16,9,5 respectivamente. En el primer paso se introducen los caracteres en la cola con prioridad con sus frecuencias y con una codificación: "". A continuación, se realiza el algoritmo descrito anteriormente.



Paso

Cola de Prioridad



Códigos

a=""	c=""
d=""	e=""
b=""	f=""
a=""	c=""
d=""	e="1"
b=""	f="0"
a=""	c="0"
d=""	e="1"
b="1"	f="0"
a=""	c="0"
d="1"	e="01"
b="1"	f="00"
a=""	c="00"
d="11"	e="101"
b="01"	f="100"
a="0"	c="100"
d="111"	e="1101"
b="101"	f="1100"

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Estructuras de datos. Curso 2017-2018
Convocatoria extraordinaria de Febrero
Grado en Ingeniería Informática.
Doble Grado en Ingeniería Informática y Matemáticas

0,25

1. (0.5 puntos) Razonar la verdad o falsedad de las siguientes afirmaciones:

   (a) El TDA **cola con prioridad** se puede implementar de forma óptima usando un heap.

   (b) La declaración **map<list<int>, string> m;** es una declaración válida.

   (c) Un **AVL** puede reconstruirse de forma unívoca dado su recorrido en **inorden**

   (d) Es imposible que un árbol binario (con más de dos nodos) sea **AVL y APO** a la vez.

   (e) En un esquema de **hashing doble** nunca puede ocurrir que para dos claves k1 y k2 distintas coincidan simultáneamente sus valores h (función hash primaria) y h_0 (función hash secundaria). Es decir, con $k1 \neq k2$, h(k1) = h(k2) y h_0(k1) = h_0(k2)

1,25

2. (1.5 puntos) Dada una clase **libro** que almacena las palabras que contiene y la posición (1..n) en que está cada palabra en el libro:

```
class libro{
 private:
   struct palabra {
     string pal; // palabra
     unsigned int posición; //posición en la que está la palabra
   };
   list<palabra> datos;
...
};
```

   - Implementa un método que dada una palabra obtenga todas las posiciones en las que aparece. **list<int> libro::posiciones(const string &pal)**
   - Implementa una clase iterator dentro de la clase libro que permita recorrer las palabras que comiencen por **z** y estén en una posición par. Implementa los métodos begin_z_par() y end_z_par().

0,5

3. (1 punto) Dado un **map<string, list<pair<int,int> > >** que contiene un conjunto de palabras de un libro y que tiene asociada a cada palabra una lista **list<pair<int,int> >** dónde cada par contiene un número de capítulo y una posición dentro del mismo dónde aparece dicha palabra.
   Construye un vector **vector<list<string> >** dónde **v[i-1]** contenga todas las palabras del capítulo **i** ordenadas alfabéticamente y sin repeticiones.
   Por ejemplo, si tenemos el map:
   {<casa, {<1,10>,<2,10>,<3,40>}>, <ventana, {<1,2>,<1,20>,<3,30>}> }
   el vector contendría:
   v[0]={casa,ventana}
   v[1]={casa}
   v[2]={casa,ventana}