1. (0.75 puntos) **(1)** Supongamos que hacemos las 3 siguientes afirmaciones:

   (a) En un esquema de hashing doble puede usarse como función hash secundaria $h_0(k)$=[M-(k%M) -1]%M, con M primo.

   (b) Es correcto hacer la siguiente definición **set <stack <int>> :: iterator q;**

   (c) Un **APO** puede reconstruirse de de forma unívoca dado su recorrido en preorden

   1-a: Las tres son ciertas **1-b:** Dos son ciertas y una falsa **1-c:** Dos son falsas y una cierta; **1-d:** Las tres son falsas. **Razonar la respuesta.**

a) Falso, la función h_0 nunca puede ser 0, ya que el valor de $h_1 = h_2 = h_3 \dots$ y no se encontraría un nuevo lugar para insertar.
En este caso si $k = M-1 \rightarrow (M-[M-1]-1)\,\%\,M = (1-1)\,\%\,M = 0\,\%\,M = 0$
Luego, el resultado debe ser un coprimo de M, cosa que tampoco asegura esta función.

b) verdadero, ya que los elementos tipos stack<int> son comparables y se pueden mantener ordenados en el set.
Luego se define un iterator para recorrer el set.

c) Verdadero, ya que el APO tiene como condición tener todos los niveles completos menos el último que tiene los nodos empujados a la izquierda.
ejemplo: pre = { 1 4 8 5 6 }

Respuesta: 1-b

(2) Supongamos que hacemos las 2 siguientes afirmaciones **si** busco el elemento máximo en un APO-min:

(a) El elemento debe estar necesariamente en una hoja.
(b) Encontrarlo lleva un tiempo O(n)

(2.a). Ambas afirmaciones son ciertas **(2.b).** Una afirmación es cierta y la otra falsa (2.c) Ambas afirmaciones son falsas. **Razonar la respuesta.**

a) Verdadero, porque todos los nodos deben ser menor a sus hijos

b) Si, ya que hay que recorrer el abol completo (tanto subarbol derecho como izquierdo)

Respuesta: 2a

2. (1.25 puntos)Supongamos que disponemos de un **map<string, list<pair<int,int>> >** que contiene un conjunto de palabras de un libro y 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.

   (a) Implementar una función que construya un vector **vector<list<string> > v,** dónde **v[i]** contenga todas las palabras del capítulo **i+1** ordenadas alfabéticamente y sin repeticiones.
   P.ej. si tenemos el map:
   <informática, {<**1**,10>,<**2**,10>,<**3**,41>}>
   <telemática, {<**1**,2>,<**1**,21>,<**2**,50>,<**3**,31> }>
   <robótica, {<**3**,30>,<**4**,5>}>
   el vector contendría:
   v[0]={informática,telemática,telemática} ->**v[0]={informática,telemática}**
   v[1]={informática, telemática}
   v[2]={informática,telemática, robótica}
   v[3]={robótica}

```
void palabras_por_capitulo (map<string, list<pair<int,int>>> &M, vector<list<string>> v){

    map<int, set<string>> aux;

    for (auto it = M.begin(); it != M.end(); ++it)
        for (auto its = it->second.begin(); its != it->second.end(); ++its)
            aux[its->first].insert(it->first);

    list<string> laux;
    for (auto it = aux.begin(); it != aux.end(); ++it)
        for (auto its = it->second.begin(); its != it->second.end(); ++its)
            laux.push_back(*its);

    v.push_back(laux);
    laux.clear();

}
```

(b) Implementar un **iterador** que itere sobre las palabras del libro que aparezcan en capítulos impares y en posiciones pares. Han de implementarse (aparte de las de la clase iteradora) las funciones **begin()** y **end()**.

```
class libro {
  private:
    map<string, list<pair<int,int>>> palabras;
  public:
    :
    class iterator {
      private:
        map<string, list<pair<int,int>>>::iterator it;
        bool condicion() {
            bool encontrado = false;
            for(auto its = it->second.begin(); its != it->second.end() && !encontrado; ++its)
                if( its->first %2 !=0 && its->second %2 ==0)
                    encontrado = true;
            return encontrado;
        }
      public:
        iterator(){}
        bool operator == ( const iterator &i ) const {
            return it == i.it;
        }
        bool operator != ( const iterator &i ) const {
            return !((*this) == i);
        }
        string & operator * (){
            return it->first;
        }
        iterator & operator ++(){
          if( it != palabras.end())
            ++it;
          while( !condicion && it != palabras.end())
            ++it;
          return *this;
        }
        friend class libro;
};

iterator begin(){
    iterator i;
    i.it = palabras.begin();
    if( !i.condicion())
        ++i;
    return i;
}
iterator end(){
    iterator i;
    i.it = datos.end();
    return i;
}
};
```

3. (1 punto) Implementar una función:

void resumecola(list<int> &L,queue<int> &Q);

que va tomando un elemento entero n de la cola Q y, si es estrictamente positivo, saca n elementos de L (si ya no quedan n elementos, saca todos los que queden) y los reemplaza por su producto. Si el elemento de la cola es negativo o cero, no hace nada. Esto ocurre con todos los elementos de Q hasta que se acaben, o bien se acaben los elementos de L. **No pueden usarse estructuras auxiliares**.

Por ejemplo: Si L=(**1,3,2**,1,4,**5,3**,2,4,1) y Q=(**3**,2,-1,0,**2**,5,2,-3) entonces L debe quedar así: L=(**6**,4,**15**,8), y Q=(2,-3) (es decir, sobran elementos de Q).

Otro ejemplo: Si L=(**1,3,2**,1,4,5,3,**2,4,1,3,2**,1,4,7) y Q=(**3**,2,-1,0,2,**5**) entonces L debe quedar así L=(**6**,4,15,**48**,1,4,7), y Q=() (es decir, sobran elementos de L que quedan como estaban en la lista)

```
void resumecola ( list <int> &L, queue <int> &Q ) {
    auto it = L.begin();
    while ( !Q.empty() && it != L.end() ) {
        if ( Q.front() > 0 ) {
            int producto = 1;
            for (int i=0; i< Q.front() && it != L.end(); i++) {
                producto *= (*it);
                it = erase(it);
            }
            insert (it, producto);
        }
        Q.pop();
    }
}
```

4. (1 punto) Dado un árbol binario A y un nodo n en el mismo, implementar una función:

bool camino_impares (bintree<int> A, bintree<int> :: node n);

que devuelva true si existe algún camino desde n a una hoja con todos los elementos impares.

Ejemplo:

true

false

```
bool caminos_impares ( bintree <int> A, bintree<int>:: node n) {
    if (n.null())
        return true;
    return ((*n) % 2 != 0) && ( caminos_impares (A, n.left()) || caminos_impares (A, n.right()) )
}
```

5. (1 punto) Dado un **vector de conjuntos** vector<set<int> > V, implementar una función

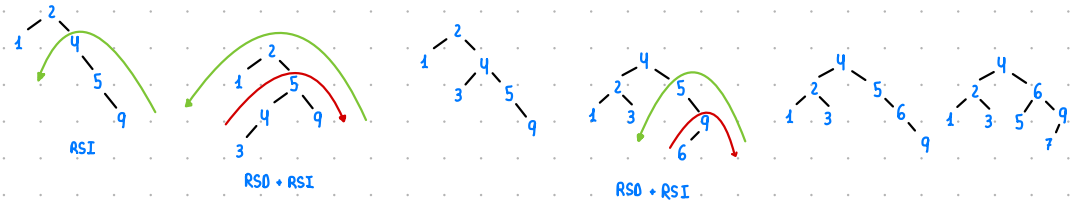
void estaenconjunto(vector<set<int> > &V, map<int,int> &recuento);

que devuelva a través del map **recuento** el índice del conjunto, junto con el número de enteros impares que contiene.

Por ejemplo, si V=[{**1**,2,**3**},{2,**3**,4},{**3**,4,**5,9**}, {2,4,6,8}, {**9**}, {**1,3,5**}] entonces debe devolver recuento={<**1**,2>, <**2**,1>, <**3**,3>, <**4**,0>, <**5**,1>, <**6**, 3>}.

```
void estaenconjunto ( vector < set < int >> & V , map < int , int > & recuento ) {
    for (int i = 0; i < V.size(); i++) {
        int num_impar = 0;
        for ( auto it = V[i].begin() ; it != V[i].end() ; ++it)
            if ( (*it) % 2 != 0)
                num_impar ++;
        recuento[i+1] = num_impares;
    }
}
```

6. (1 punto) (a) Insertar (detallando los pasos) las siguientes claves (insertadas en ese orden) en un **AVL**: {2, 1, 4, 5, 9, 3, 6, 7} especificando los pasos seguidos e indicando cuando sea necesario el **tipo de rotación** que se usa para equilibrar y mantener la estructura de AVL.



(b) Insertar (detallando los pasos) las siguientes claves (en el orden indicado):
{51, 35, 53, 70 ,54 ,56, 86, 42, 11, 67, 57}

en una **tabla hash cerrada** de tamaño 13 con resolución de colisiones usando hashing doble.

$M = 13$

$h_1(k) = h(k) = k \% 13$

$h_0(k) = 1 + (k \% 11)$

$h_i(k) = (h_{i-1}(k) + h_0(k)) \% 13 \quad i = 2,3,4...$

$h_1(51) = 12$
$h_1(35) = 9$
$h_1(53) = 1$
$h_1(70) = 5$
$h_1(54) = 2$
$h_1(56) = 4$
$h_1(86) = 8$
$h_1(42) = 3$

$h_1(11) = 11$
$h_1(67) = 2$
- $h_0(67) = 2$
- $h_2(67) = 4$
- $h_3(67) = 6$

$h_1(57) = 5$
- $h_0(57) = 3$
- $h_2(57) = 8$
- $h_3(57) = 11$
- $h_4(57) = 1$
- $h_5(57) = 4$
- $h_6(57) = 7$

	K	estado	
0			
1	53	x	
2	54	x	
3	42	x	
4	56	x	
5	70	x	
6	67	x	
7	57	x	
8	86	x	
9	35	x	
10			
11	11	x	
12	51	x	