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

   (a) En un esquema de **hashing doble puede ocurrir una sola vez** que para dos claves distintas k1 y k2, coincidan los valores de sus funciones hash primaria y secundaria asociadas, es decir $k1 \neq k2$, $h(k1)=h(k2)$ y $h_0(k1)=h_0(k2)$

   (b) Dado un **map <string, stack<int> >::iterator it,** no es posible hacer *it= k donde k es de tipo pair<string, stack<int> >

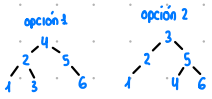
   (c) Un **AVL** no puede reconstruirse de de forma unívoca dado su recorrido en inorden

   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.

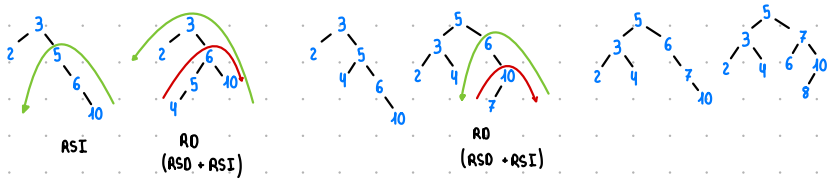
a) Falso, puede ocurrir más de una vez, esto significa que las funciones elegidas no son muy eficientes pero no hace que sean inválidas

b) Verdadero, map está diseñado para que solo se pueda modificar el valor asociado a la clave con it→second pero no se puede ni modificar la clave ni el par completo

c) Verdadero, ej: in = {1 2 3 4 5 6} → el inorden de un AVL siempre es creciente



(2) Si inserto las claves {3, 2, 5, 6, 10, 4, 7, 8} en un AVL de enteros, **2-a:** Hay que hacer dos rotaciones simples y una rotación doble **2-b:** Hay que hacer tres rotaciones dobles, **2-c:** Hay que hacer dos rotaciones dobles y una rotación simple **2-d:** Todo lo anterior es falso. **Mostrar el arbol final**



Respuesta: 2-c

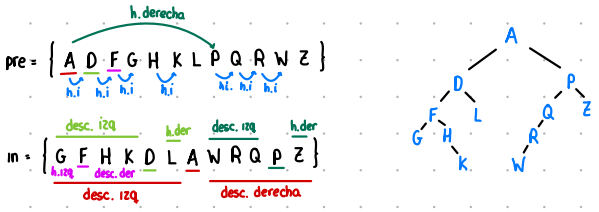
(3) El recorrido preorden de un cierto árbol binario es:

A D F G H K L P Q R W Z

y en inorden es:

G F H K D L A W R Q P Z

Construir el árbol binario.



(4) ¿Cuantos ABB diferentes pueden construirse (en cualquier orden) con los enteros {1,2,3,4}? ¿Y cuantos AVL? ¿Y cuantos APO?

• ABB

$$C_n = \frac{(2n)!}{(n+1)!\cdot n!} \qquad C_4 = \frac{8!}{5!\,4!} = 14 \text{ diferentes}$$

• AVL • 1 y 4 no pueden ser raices

4 diferentes

• APO • La raiz debe ser 1

• una hoja de estar en el último nivel a la izquierda.

3 diferentes

2. (1 punto) Tenemos un contenedor de pares de elementos, {string, stack<int>} definido como:

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

Implementar una clase **iterator** (constructor, *, ==, !=, ++) que itere sobre los string de longitud 4 y para los que la stack<int> tenga todos sus enteros impares. Además de los métodos **begin**() y **end**() de la clase contenedor.

```
class contenedor {
    private:
        map<string, stack<int>> datos;
    public:
        ⋮
        class iterator {
            private:
                map<string, stack<int>>::iterator it;
                bool condicion(){
                    if (it->first.size() != 4)
                        return false;
                    stack<int> aux = it->second;
                    while (!aux.empty()){
                        if (aux.top()%2 == 0)
                            return false;
                        aux.pop();
                    }
                    return true;
                }
            public:
                iterator(){}
                string & operator *(){
                    return it->first;
                }
                bool operator ==(const iterator &i) const{
                    return it == i.it;
                }
                bool operator !=(const iterator &i) const{
                    return !((*this) == i);
                }
                iterator & operator ++(){
                    ++it;
                    while (!condicion && it != datos.end())
                        ++it;
                    return *this;
                }
            friend class contenedor;
        };
        iterator begin(){
            iterator i;
            i.it = datos.begin();
            if(!i.condicion)
                ++it;
            return i;
        }
        iterator end(){
            iterator i;
            i.it = datos.end();
            return i;
        }
};
```

3. (1 punto) Dado un vector de listas VL y una lista L, implementar una función:
bool combina (vector<list<int> > VL, list<int> L);
que devuelva true si L es una combinación de las listas de VL no importando el orden en que aparezcan.
Notas:
- Los elementos de cada lista de VL deben aparecer solamente una vez en L.
- L no puede tener otros elementos adicionales aparte de los de la combinación de las listas de VL.
- El orden en que aparezcan las listas en L no importa, pero los elementos de cada lista si que deben aparecer en el mismo orden.

Por ejemplo:
Si VL=[**(1,2,3)**,(4,5,6),**(7,8)**], y L=(**7,8**,4,5,6,**1,2,3**) entonces:
combina(VL, L) devuelve **true.**
Si L=(**3,2,1**,7,8,4,5,6), entonces: combina(VL, L) devuelve **false.**

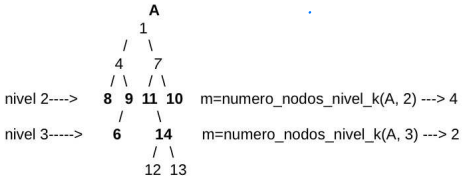
```
bool combina ( vector < list < int >> VL , list < int > L ) {
    for ( auto it = VL.begin() ; it != VL.end() ; ++it )
        if ( ! encontrar ( (*it) , L ) )
            return false;
    if ( ! L.empty() )
        return false;
    return true ;
}
```

```
bool encontrar ( list < int > L1, list < int > &L2 ) {
    auto it2 = L2.begin();
    while ( it2 != L2.end() ) {
        auto it2_copia = it2;
        auto it1 = L1.begin();
        while ( it2_copia != L2.end() && it1 != L1.end()
                && (*it2_copia) == (*it1) ) {
            ++it2_copia ;
            ++it1 ;
        }
        if ( it1 == L1.end() ) {
            L2.erase ( it2 , it2_copia );
            return true;
        }
        ++it2 ;
    }
    return false;
}
```

4. (1 punto) Implementar una función que determine el número de nodos que se encuentran en el nivel k de un árbol binario A

int numero_nodos_nivel_k(const bintree<int> &A, int k);

Ejemplo:




```
int numero_nodos_nivel_k(const bintree<int> &A, int k){
    bintree<int>::node n = A.root();
    return contar_nodos(n, 0, k);
}

int contar_nodos(bintree<int>::node n, int nivel, int k){
    if(n.null()) return 0;
    if(nivel == k) return 1;
    return contar_nodos(n.left(), nivel+1, int k) + contar_nodos(n.right(), nivel+1, int k);
}
```

5. (1 punto) Implementar una función

void corta_map(map<int, list<int> > &M, int p, int q);

que **elimine** todas las claves y todos los elementos de las listas asociadas que **no están** en el rango **[p,q)**. Aunque una clave no se elimine por estar en el rango, **también se deben eliminar los elementos de la lista asociada que no están en el rango**, con la particularidad de que si la lista quedara vacía entonces la clave también debe ser eliminada. **No se pueden usar contenedores auxiliares.**

Por ejemplo:
Si M={1->(2,3,4),
5->(**6,7,8**),
8->(4,5),
3→(1,3,7)},
entonces **corta_map(M,1,6)** debe dejar
M={1->(2,3,4),
3->(1,3)}.
Nótese que la clave 5 ha sido eliminada aunque está dentro del rango porque su lista quedaría vacía.

```
void corta_map(map<int, list<int> > &M, int p, int q){
    for(auto it = M.begin(); it != M.end(); ){
        if(it->first < p || it->first >= q)
            it = M.erase(it);
        else{
            for(auto itl = it->second.begin(); itl != it->second.end(); )
                if((*itl) < p || (*itl) >= q)
                    itl = it->second.erase(itl);
                else
                    ++itl;
            if(it->second.empty())
                it = M.erase(it);
            else
                ++it;
        }
    }
}
```

6. (1 punto) (a) Insertar (detallando los pasos) las siguientes claves (en el orden indicado):

{55, 39, 57, 74 ,58 ,60, 90, 46, 15, 71, 61}

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

$M = 13$

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

- $h_0(k) = 1 + (k \% 11)$
- $h_i(k) = (h_{i-1}(k) + h_0(k)) \% 13$

$h_1(59) = 7$
$h_1(39) = 0$
$h_1(57) = 5$
$h_1(74) = 9$
$h_1(58) = 6$
$h_1(60) = 8$
$h_1(90) = 12$
$h_1(46) = 7$
- $h_0(46) = 2$
- $h_2(46) = 9$
- $h_3(46) = 11$

$h_1(15) = 2$
$h_1(71) = 6$
- $h_0(71) = 6$
- $h_1(71) = 11$
- $h_3(71) = 3$

$h_1(61) = 9$
- $h_0(61) = 7$
- $h_2(61) = 3$
- $h_3(61) = 10$

	K	estado	
0	39	x	
1			
2	15	x	
3	71	x	
4			
5	57	x	
6	58	x	
7	59	x	
8	60	x	
9	74	x	
10	61	x	
11	46	x	
12	90	x	

(b) Construir un **APOmin** a partir de las claves {10, 12, 1, 14, 5, 8, 15, 3, 9, 7, 4, 2} insertadas en ese orden.