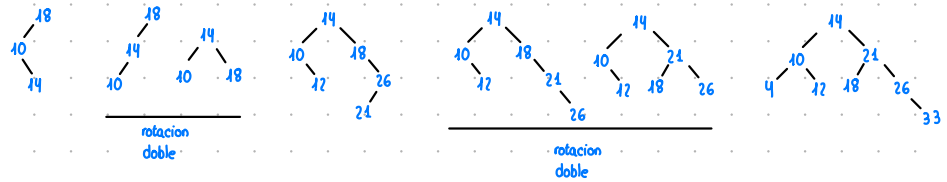
1. (1 punto) **(a)** Si inserto las claves {18, 10, 14, 26, 12, 21, 33, 4} en un AVL de enteros, (a1): Hay que hacer dos rotaciones simples y una rotación doble (a2): Hay que hacer una rotación simple y una rotación doble, (a3): Hay que hacer dos rotaciones dobles y una rotación simple (a4): Todo lo anterior es falso. **Mostrar el árbol final**

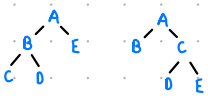


Respuesta: a4 - Dos rotaciones dobles

(b) ¿Puede reconstruirse forma unívoca un árbol binario en el que todos los nodos tienen 0 ó 2 hijos, conociendo su preorden? ¿y un árbol binario completo conociendo su postorden? Razonarlo

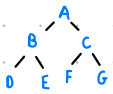
- No, existiría más de una posibilidad

ejemplo: {A, B, C, D, E}



- Si, esto se cumple por la condición de que es un arbol completo, por lo que tiene todos los niveles completos, excepto el último que tiene las hojas empujados a la izquierda.

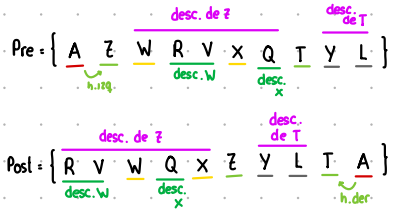
ejemplo: {D, E, B, F, G, C, A}



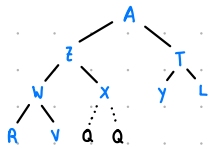
(c) Dados los siguientes recorridos Preorden y Postorden:

Pre = {A,Z,W,R,V,X,Q,T,Y,L} **Post = {R,V,W,Q,X,Z,Y,L,T,A}**

(c1) Hay exactamente 2 árboles binarios con esos recorridos. (c2) No hay ningún árbol binario con esos recorridos.(c3) Hay exactamente 1 árbol binario con esos recorridos. (c4) Hay más de 2 árboles binarios con esos recorridos



1. Raiz → A
2. Z y T
3. descend. de Z y T
4. W y X
5. descend. de W y X
6. Y y L



Respuesta: c1

(d) Dado el siguiente fragmento de código:

{**map <int,int> M**; M[0]=1; map <int,int> ::iterator p; p=M.find(9);}

¿Cual de las siguientes afirmaciones es verdadera?

d-1: M no se modifica y p->first=9 **d-3:** M se modifica y p ->first=9

d-2: Da un error **d-4:** M se modifica y p=M.end()

Respuesta: d-4

M se modifica en la instruccion M[0]=1.

Y al no haber un 9 en M, la función find devuelve end()

2. (1 punto) Tenemos un contenedor de pares de elementos, **{clave, bintree<int>}** definida como:

```
template <typename T>
class contenedor {
    private:
      unordered_map<T, bintree<int> > datos;
    ..............
};
```

Implementar un **iterador** que itere sobre las claves que cumplan la propiedad de que el bintree asociado tenga como suma de sus etiquetas un número par. Se deben implementar (aparte de las de la clase iterator) las funciones begin() y end() de la clase contenedor.

```
template < typename T >
class contenedor {
    private :
        unordered_map < T , bintree < int > > datos ;
    public :
        ⋮
        class iterator {
            private:
                unordered_map < T , bintree < int > > :: iterator it ;

                int suma ( bintree < int > :: node n ) {
                    if ( n.null() )
                        return 0 ;
                    else
                        return (*n) + suma(n.left()) + suma(n.right()) ;
                }
                bool condicion () {
                    if ( suma(datos->second.root()) % 2 == 0)
                        return true;
                    else
                        return false ;
                }
            public :
                iterator() { }

                bool operator == (const iterator & i) const {
                    return it == i.it ;
                }
                bool operator != (const iterator & i) const {
                    return ! (*this == i) ;
                }
                T & operator * () {
                    return * it->first ;
                }
                iterator & operator ++ () {
                    if ( it != datos.end() )
                        ++ it;
                    while ( ! condicion () && it != datos.end() )
                        ++it;
                    return * this ;
                }
                friend class contenedor ;
        } ;

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

3. (1 punto) Implementar una función

bool permutalista (list<int> & L1, list<int> & L2)

que devuelva true si L1 y L2 tienen la misma cantidad de elementos y los elementos de L1 son una permutación de los de L2

P.ej: si L1={1,23,21,4,2,3,0} y L2={21,1,3,2,4,23,0} devolvería TRUE pero si L1={1,3,5} y L2={1,5,4} devolvería FALSE

Si hay elementos repetidos tienen que estar el mismo número de veces en las 2 listas para poder ser TRUE. **No pueden usarse estructuras auxiliares**, el algoritmo puede ser destructivo y no conservar las listas iniciales y **no puede usarse ningún algoritmo de ordenación**.

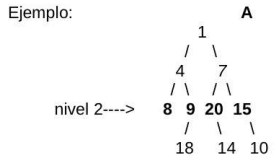
```
bool permutalista ( list<int> & L1, list<int> & L2 ) {
    if ( L1.size() != L2.size() ) permutacion = false ;
    for ( auto it = L1.begin() ; it != L1.end() && permutacion ; ++it ) {
        bool encontrado = false;
        for ( auto it2 = L2.begin() ; it2 != L2.end() && !encontrado ; ++it2 ){
            if ( (*it) == (*it2) ) {
                encontrado = true;
                L2.erase (it2);
            }
        }
        if ( !encontrado ) permutacion = false ;
    }
    return permutacion;
}
```

4. (1 punto) Dado un bintree<int>, implementar una función

void prom_nivel(bintree<int> &T, list<float> &P);

que genere una lista de reales P, donde el primer elemento de la lista sea el promedio de los nodos del árbol de nivel 0, el segundo sea el promedio de los de nivel 1, el tercero el promedio de los de nivel 2, y así sucesivamente. Es decir, que si el árbol tiene profundidad N, la lista tendrá N+1 elementos de tipo float.

Ejemplo:



prom_nivel(A, P)

P={1.0, 5.5, **13.0**, 14.0}

```
void prom_nivel ( bintree <int> &T, list <float> &P ) {
    vector <vector<int>> aux;
    bintree <int>:: node n = T.root();
    if ( n.null() ) return;
    new_nivel ( n, aux, 0 );
    for (int i=0; i< aux.size() ; i++) {
        float suma = 0.0;
        for ( int j=0; j< aux[i].size() ; j++)
            suma += aux [i][j];
        P. push_back ( suma / aux[i].size());
    }
}
```

```
void new_nivel (bintree <int>:: node n , vector <vector<int>> aux, int nivel ) {
    if ( aux.size() < nivel )
        aux. push_back ( vector<int> () );
    aux[nivel]. push_back ( (*n));
    if ( !n.left().null() )
        new_nivel ( n.left(), aux, nivel +1);
    if ( !n.right().null)
        new_nivel ( n.right(), aux, nivel +1);
}
```

5. (1 punto) Implementar una función:

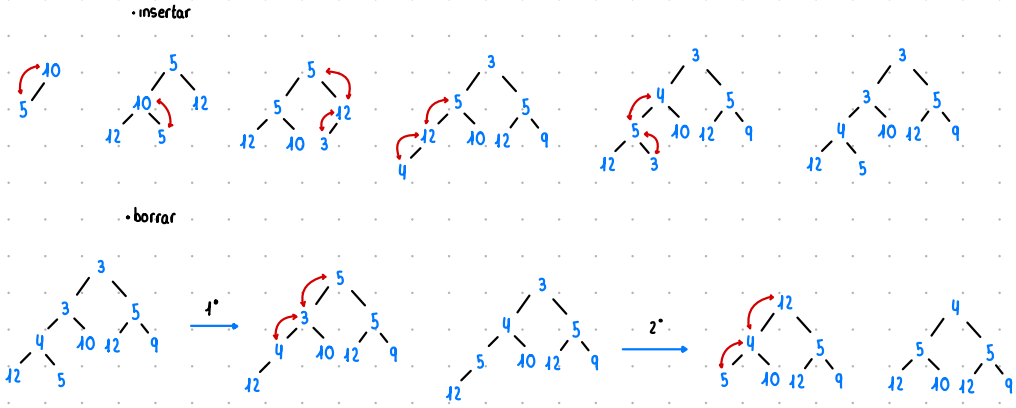
bool tiene_suma_constante (set<int> s, int M);

que dado un conjunto de enteros **s** y un entero **M**, devuelva true si existe un subconjunto de elementos enteros cuyos valores sumen exactamente M.

P.ej si. s={**1,2,3,4,5**,6}; **tiene_suma_constante**(s,**15**) devolvería **true**
y si hacemos **tiene_suma_constante**(s,**22**) devolvería **false**

```
bool tiene_suma_constante (set<int> s, int M){
    for (auto it = s.begin(); it != s.end(); ++it){
        int suma = 0;
        for (auto it2 = it; it2 != s.end() && suma < M; ++it2){
            suma += (*it2);
        }
        if (suma == M) return true;
    }
    return false;
}
```

6. (1 punto) (a) Insertar en el orden indicado (detallando los pasos) las siguientes claves en un **APO**: {10, 5, 12, 12, 5, 3, 9, 4, 3}. Borrar dos elementos y mostrar el APO resultante



(b) Insertar (detallando los pasos) las siguientes claves (en el orden indicado):

{47, 31, 49, 66 ,50 ,52, 82, 38, 7, 63, 53}

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

$M = 13$

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

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

	47	31	49	66	50	52	82	38	7	63	53
$h_1(k)$	8	5	10	1	11	0	4	12	7	11	1
$h_0(k)$	4	10	6	1	7	9	6	6	8	9	10
$h_2(k)$										7	11
$h_3(k)$										3	8
$h_4(k)$											5
$h_5(k)$											2

	k	dir	estado
0	52		×
1	66		×
2	53		×
3	63		×
4	82		×
5	31		×
6			
7	7		×
8	47		×
9			
10	49		×
11	50		×
12	38		×