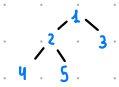
1. (1 punto) **(a)** Si insertamos un conjunto de enteros **ordenado** ascendentemente en un ABB, un APO-min y un AVL: ¿En cual de los 3 es más eficiente la operación de inserción? **Razonarlo.**

ABB

O(n), muy mala

APO-min



al no tener que hacer intercambios padre-hijo la eficiencia es O(1)

AVL En un AVL hay que hacer rotaciones para corregir los desequilibrios, por lo que es menos eficiente que O(1)

Respuesta: APO-min

(b) (b1) ¿Cuántos elementos, en promedio, hay en cada lista en una tabla hash abierta de tamaño B con N elementos ? **(b1)** (N/B) **(b2)** (N + B) **(b3)** (B/N) **(b4)** 1. **Razonarlo.**

La tabla consta de B listas y si hay N elementos, con b1. N/B distribuimos el mismo número de elementos por lista, la media.

(b2) ¿Es correcto en un esquema de hashing cerrado con un tamaño de la tabla **B** primo con resolución de colisiones usando hashing doble, el uso como función hash:

h(x)=[(x % C) % B] ? ¿Y como función hash secundaria **h_0(x)=[(x*C) % (B-2)] ?** B, C primos entre si. **Razonarlo**

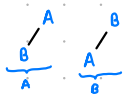
- En la función h(x) primero se hace módulo C entonces los valores quedan entre 0 y C-1, por lo que si C < B, luego al hacer módulo B no existiran valores que se ubiquen en las últimas posiciones de la tabla, por lo que no distribuye uniformemente los valores. No es válida.
- Luego, la función h_0(x) tiene como condición que todos los valores deben ser coprimos de B. Cómo se hace módulo de B-2 y todo número menor que B es coprimo con B, pues se cumple la condición y es válida.

(c) Dadas las siguientes 3 afirmaciones:

- Dados A y B dos árboles binarios (con más de un nodo) distintos con etiquetas diferentes, nunca puede ocurrir simultáneamente: Pre (A) = Post (B) y Post (A)=Pre (B)
- Un APO puede reconstruirse de forma unívoca dado su recorrido en postorden
- Solo hay un APO que tiene como preorden={4,9,24,33,21,74,63}

(c1) Todas son falsas **(c2)** Hay 2 ciertas y 1 falsa **(c3)** Hay 1 cierta y dos falsas **(c4)** Todas son ciertas. **Razonar la respuesta.**

- Falso, ejemplo:

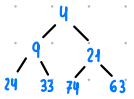


Pre(A) = AB = Post(B)

Post(A) = BA = Pre(B)

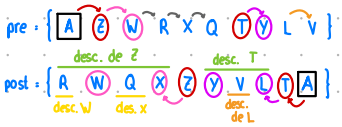
- Verdadero, ejemplo: postorden = {7 6 4 1} Tenemos 3 posibilidades que cumplen con el postorden, pero solo el primero cumple con la condicion de APO de que tiene que tener completo todos los niveles, menos el último (empujado a la izquierda)

- Verdadero, al ser un APO con 7 elementos, debe tener siempre 3 niveles y todos completos

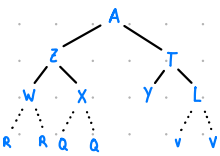


Respuesta: c2

(d) Dados los siguientes recorridos en **preorden** = (A,Z,W,R,X,Q,T,Y,L,V), y **postorden** = (R,W,Q,X,Z,Y,V,L,T,A) de un árbol binario **(d1):** No hay ningún árbol binario con esos recorridos asociados; **(d2):** Hay 1 solo árbol binario con esos recorridos asociados; **(d3):** Hay exactamente dos árboles binarios con esos recorridos asociados; **(d4):** Todo lo anterior es falso. **Razonar la respuesta.**



1.- Raiz
2.- hijos → Z y T
3.- descendientes de Z y T
4.- hijos de Z → X y W
5.- descendientes de X y W
6.- hijos de T → Y y L
7.- desc. de L



R: d4, hay más de dos árboles posibles

2. (1 punto) Se desea construir un **traductor** de un idioma origen a un idioma destino. Una palabra en el idioma origen puede tener más de una traducción en el idioma destino.

   Usando como **representación** para el **TDA Traductor** un **map<string,set<string> >:**

   • Implementar una **clase iteradora** dentro de la clase Traductor para **dada una palabra** en el idioma de destino, iterar por todas las palabras del idioma de origen que tengan como traducción esa palabra. Han de implementarse (aparte de las de la clase iteradora) las funciones **begin** y **end**

```
class Traductor {
  private:
    map<string, set<string>> datos;
  public:
    ⋮
    class iterator {
      private:
        map<string, set<string>>::iterator it;
        string palabra;

        bool hay_destino(){
          set<string>::iterator aux;
          aux = it->second.find(palabra);
          if(aux == it->second.end())
            return false;
          else
            return true;
        }

      public:
        iterator(){}

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

    iterator begin(string p){
      iterator i;
      i.palabra = p;
      i.it = datos.begin();
      if(!i.hay_destino())
        ++i;
      return i;
    }

    iterator end(string p){
      iterator i;
      i.palabra = p;
      i.it = datos.end();
      return i;
    }
};
```

3. (1 punto) Implementar una función

void intercambia_sec (list<int>& L);

que dada una lista L, intercambie el grupo de los primeros elementos consecutivos impares por el siguiente grupo de elementos consecutivos pares de principio a final de la lista. **No pueden usarse estructuras de datos auxiliares.**

Por ejemplo si L={**1**,2,4,**5**,6,8,**7**,**9**,**13**,2,**9**} después de llamar a intercambia_sec (L) debe quedar L={**2**, **4**, 1, **6**, **8**, 5, **2**, 7, 9, 13,9}

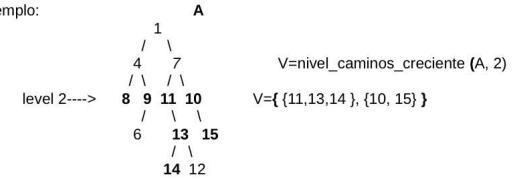
```
void intercambia_sec (list<int>& L) {
    if (L.empty()) return;

    auto end_pares = L.begin();

    bool seguir = true;
    while (seguir) {
        auto start_impares = end_pares;
        while (start_impares != L.end() && (*start_impares) % 2 == 0)
            ++start_impares;
        if (start_impares == L.end()) return;

        auto end_impares = start_impares;
        while (end_impares != L.end() && (*end_impares) % 2 != 0)
            ++end_impares;
        if (end_impares == L.end()) return;

        auto start_pares = end_impares;

        end_pares = start_pares;
        while (end_pares != L.end() && (*end_pares) % 2 == 0)
            ++end_impares;

        auto it1 = start_impares;
        auto it2 = start_pares;

        while (it1 != end_impares && it2 != end_pares) {
            int aux = *it1;
            *it1 = *it2;
            *it2 = aux;
            ++it1;
            ++it2;
        }

        while (it1 != end_impares) {
            L.insert(it2, *it1);
            it1 = L.erase(it1);   ---> con erase el iterador se invalida pero
        }                             devuelve un nuevo iterador al siguiente
                                      elemento (por eso no hay que hacer ++it1)
        while (it2 != end_pares) {
            L.insert(it1, *it2);
            it2 = L.erase(it2);
        }

        if (end_pares == L.end()) seguir = false;
    }
}
```

4. (1 punto) Dado un árbol binario de enteros A y un nivel en el árbol **level**, implementar una función:

   vector<list<int>> nivel_caminos_creciente (bintree<int> A, int level);

   que encuentre los caminos de longitud mayor o igual que uno **desde cualquier nodo** de ese nivel a una hoja en el árbol **que tengan una secuencia creciente de valores de etiquetas**. Devolver la solución en un vector de listas.

Ejemplo:




```
vector < list < int >>  nivel_caminos_creciente ( bintree < int > A , int level ) {
    vector < bintree < int > :: node >  nodos ;
    vector < list < int >>  caminos ;
    list < int > camino ;
    encontrar_nodos ( n.root(), nodos , level, 0 ) ;
    for ( int i=0 ; i < nodos.size() ; i++ ){
        encontrar_caminos ( nodos[i], caminos , camino ) ;
    return caminos ;
}

void encontrar_caminos ( bintree < int > :: node &n , vector < list < int >> &caminos, list < int > &camino ) {
    if ( !camino.empty() && (*n) < camino.back () ) return;
    camino. push_back (*n) ;
    if ( n.left().null() && n.right().null() )
        if ( camino.size() > 1 ) caminos. push_back (camino) ;
    else {
      if ( !n.left().null() ) encontrar_caminos (n.left(), caminos , camino ) ;
      if ( !n. right(). null() ) encontrar_caminos (n. right , caminos , camino ) ;
    }
    camino. pop_back() ;
}

void  encontrar_nodos ( bintree < int > :: node &n , vector < bintree < int > :: node > &nodos , int level , int contador ) {
      if ( level == contador ) {
        nodos. push (n) ;
      } else {
          if ( !n.left(). null() ) encontrar_nodos ( n.left(), nodos, level , contador +1 ) ;
          if ( !n.right(). null() ) encontrar_nodos ( n.right(), nodos, level , contador +1 ) ;
      }
}
```

5. (1 punto) Implementar una función:

void solo_en_2(vector<set<int> > &VS,set<int> &S1);

que dado un vector de conjuntos enteros VS, encuentre el conjunto S1 de todos aquellos elementos que están exactamente en dos de ellos.

Por ejemplo,si VS=[{**0**,1,**2**,3}, {1,3,**4**,5}, {1,3,6,7}, {**2**,**4**,7,**9**}, {**0**,7,8,**9**}], entonces S1={**0**,**2**,**4**,**9**}

```
void solo_en_2(vector<set<int>> &VS, set<int> &S1){
    map<int, int> aux;
    for(auto cit = VS.cbegin(); cit != VS.cend(); ++it){
        for(auto cit2 = cit->cbegin(); cit2 != cit->cend(); ++cit2){
            if(aux.find(*cit2) == aux.end())
                aux[*cit2] = 1;
            else
                aux[*cit2] += 1;
        }
    }
    for(auto it = aux.begin(); it != aux.end(); ++it)
        if(it->second == 2)
            S1.insert(it->first);   // it->first  /  (*it).first
}
```

6. (1 punto) (a) Insertar (en ese orden) las claves {3, 11, 15, 37, 8, 6} en una **Tabla Hash cerrada** de tamaño 13 usando como función hash **h(k) = k % 13**. A continuación borrar el 15 y finalmente insertar el valor 21. Resolver las colisiones usando **hashing doble.**

$M = 13$

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

- $h_i(k) = (h_{i-1}(k) + h_0(k)) \% 13 \quad i = 2, 3, 4 \ldots$
- $h_0(k) = 1 + (k \% 11)$

	3	11	15	37	8	6	21
$h_1(k)$	3	11	2	11	8	6	8
$h_0(k)$	4	1	5	5	9	7	11
$h_2(k)$				3	4		6
$h_3(k)$				8			4
$h_4(k)$							2
$h_5(k)$							0

	K	dir	estado
0	21		X
1			
2	15		B
3	3		X
4	8		X
5			
6	6		X
7			
8	37		X
9			
10			
11	11		X
12			

(b) Construir un **AVL** con las claves {7, 6, 9, 10, 14, 8, 11}, especificando los pasos seguidos e indicando cuando sea necesario el **tipo de rotación** que se usa para equilibrar y mantener la estructura de AVL.