Santiago Miquel García Santamaría 2º DGIIM

@smgarsan

Relación de ejercicios propuestos

Ejercicio 1

Implementar una función `int max_subtree (bintree<int> & T)` que devuelva la suma de etiquetas máxima de entre todos los posibles subárboles del árbol binario T.
Las etiquetas de los nodos pueden ser positivas o negativas (por tanto, el súbarbol vacío puede ser el de mayor suma, que se considera 0).

Para resolverlo, asignaremos a cada nodo una puntuación, que será la suma de las etiquetas del subárbol que cuelga de él. Así, la puntuación de cada nodo será el valor de su etiqueta, más la puntuación de su hijo izquierda y más la puntuación de su hijo derecha (hijo no existente tiene puntuación 0).

Recorreremos, entonces, el árbol en postorden (para visitar antes a los hijos), iremos asignando puntuaciones a los nodos y nos iremos quedando con la mayor puntuación.

```
int max_subtree(bintree<int> & T){
  int maximo = 0;
  bintree<int>::postorder_iterator it;

  for(it = T.begin_postorder(); it != T.end_postorder(); ++it){
    if(!it.getNodo().left().null())
      *it.getNodo() += *it.getNodo().left();

    if(!it.getNodo().right().null())
      *it.getNodo() += *it.getNodo().right();

    if(maximo < *it.getNodo())
      maximo = *it.getNodo();
  }
  return maximo;
}
```

Ha sido necesaria la implementación de la siguiente función:

```
nodo & bintree<int>::postorder_iterator::getNodo(){return elnodo};
```

Ejercicio 2

Implementar una función `void rotalista(list<int> & l, int n)` que traslada los n primeros elementos de la lista al final de la misma. Ejemplo: lista={1, 3, 5, 4, 2, 6}, con n=2 obtenemos lista={5, 4, 2, 6, 1, 3}.

Con precondición $0 \leq n \leq l.size()$, la siguiente función resuelve el problema.

```
void rotalista(list<int> & l, int n){
    for(int i=0; i<n; i++){
        l.push_back(*l.begin());
        l.pop_front();
    }
}
```

Ejercicio 3

Implementar una función `void juntalista(list<int> & l, int n)` que dada una lista agrupe los elementos de n en n dejando su suma. Ejemplo: lista={1, 3, 2, 4, 5, 2, 2, 3, 5, 7, 4, 3, 2, 2}, con n=3 queda lista={6, 11, 10, 14, 4}. No pueden usarse estructuras auxiliares y la función debe ser O(n).

Para resolver este ejercicio usaremos tres iteradores: escritura, que irá tomando el primer valor de cada subgrupo de n elementos, donde se acumulará la suma del subgrupo; lectura, que irá moviéndose entre los elementos del subgrupo; y borrado, que permitirá borrar todo el subgrupo menos el primer elemento tras haber acumulado la suma.

```
void juntalista(list<int> & l, int n){
    list<int>::const_iterator lectura, borrado;
    list<int>::iterator escritura;
    int contador;

    escritura = l.begin();
    while(escritura != l.end()){
        lectura = escritura;
        lectura++;
        borrado = lectura;

        contador = 1;
        while(contador<n && lectura != l.cend()){
            *escritura += *lectura;
            lectura++;
            contador++;
        }

        l.erase(borrado, lectura);
        escritura++;
    }
}
```

Otra implementación alternativa sería la siguiente:

```
void juntalista (list<int> & l, int n) {
    for (list<int>::iterator it = l.begin(); it != l.end(); it++) {
        list<int>::iterator aux = it;

        int suma = *it;
        for (int i = 1; i < n; i++) {
            it = aux;
            if (++it != l.end()) {
                suma += *it;
                l.erase(it);
```

```
            }
        }

        it = aux;
        *it = suma;
    }
}
```

En esta, nos paramos en una posción, y sumamos los n siguientes números uno por uno, de forma que al acumularlos, los eliminamos.

Ejercicio 4

Implementar una función `void ordenag(list<int> & l, int m)` que dada una lista, ordene sus elementos a grupos de a m elementos. Por ejemplo si m=5, la función ordena los primeros 5 elementos entre sí, después los siguientes 5 y así sucesivamente. Si la longitud de la lista no es un múltiplo exacto de m elementos, entonces los últimos l.size()%m elementos se ordenan también. Ejemplo: lista={10, 1, 15, 7, 2, 12, 1, 9, 13, 3, 7, 6, 19, 15, 16, 11, 15}, con m=5 queda lista={1, 2, 7, 10, 15, 1, 3, 9, 12, 13, 6, 7, 15, 16, 19, 11, 15}.

```
void ordenag(list<int> & l, int m){
    list<int>::iterator it = l.begin();
    while(it != l.end()){
        multiset<int> aux;
        list<int>::iterator posicion = it;
        int contador = 0;

        while(contador < m && it != l.end()){
            aux.insert((*it));
            it++;
            contador++;
        }

        posicion = l.erase(posicion, it);
        l.insert(posicion, aux.begin(), aux.end());
    }
}
```

Ejercicio 5

Implementar una función `int dminmax(list<int> & l)` que dada una lista l, devuelva la distancia entre las posiciones del mínimo y del máximo de la lista. La distancia debe ser positiva si el mínimo está antes del máximo y negativa en caso contrario. Ejemplo: lista1={5, 1, 3, 2, 4, 7, 6} devolvería 4, mientras que lista2={5, 9, 3, 2, 4, 1, 6} devolvería -4.

Si el valor del mínimo aparece repetido se debe tomar la primera posición en que aparece, mientras que para el máximo se debe tomar la última.

```
int dminmax(list<int> & l){
    pair<int, int> min = {0, *(l.begin())};
    pair<int, int> max = min;
    list<int>::iterator it;
```

```
    int contador = 0;
    for(it = l.begin(); it != l.end(); it++){
        if((*it) < min.second)
            min = {contador,  *it};
        if((*it) >= max.second)
            max = {contador, *it};
        contador++;
    }

    return (max.first - min.first);
}
```

Ejercicio 6

Implementar una función `bool lexiord(list<int> & l1, list<int> & l2)` que devuelve true si l1 es mayor (en sentido lexicográfico) que l2 y false en caso contrario.

El orden lexicográfico se define comparando los elementos en las posiciones equivalentes de ambas listas hasta encontrar uno diferente. La lista que tenga el valor más grande es la mayor en orden lexicográfico. Si una lista está totalmente contenida en otra, es mayor la más larga. Si son iguales la función devolverá false.

Ejemplo: lista1={1, 3, 2, 4, 6}, lista2={1, 3, 2, 5}, llamando a lexiord(lista1, lista2) obtendríamos false.

```
bool lexiord(list<int> & l1, list<int> & l2){
    bool l1esmayor = true;

    if(l1.size()<l2.size())
        l1esmayor = false;

    list<int>::iterator it1 = l1.begin();
    list<int>::iterator it2 = l2.begin();
    while(l1esmayor && (it2 != l2.end())){
        if((*it1)<(*it2))
            l1esmayor = false;
        it1++;
        it2++;
    }

    return l1esmayor && (it1 != l1.end());
}
```

Ejercicio 7

Implementar una función `void rotacion(queue<int> & C)` que saque una cierta cantidad de enteros del frente de la cola C y los vuelve a insertar al final de la cola de forma que queda en el frente el primer número par que haya en la cola.

Ejemplo: Dada C={1, 3, 5, 2, 4}, tras la rotación obtenemos C={2, 4, 1, 3, 5}.

```
void rotacion(queue<int> & C){
    int contador = 0;
    bool par = false;
    while(!par && contador < C.size()){
        if(C.front()%2 != 0){
            C.push(C.front());
            C.pop();
            contador++;
        }
        else
            par = true;
    }
}
```

Ejercicio 8

Implementar una función `bool pesointerior(bintree<int> a)` que devuelve true si la etiqueta de los nodos interiores de a es la suma de las etiquetas de las hojas que cuelgan de ellos.

Para resolverlo, recorreremos una primera vez el árbol de modo que cuando detectemos que un nodo es hoja, restaremos el valor de su etiqueta a todos sus antecesores hasta llegar a la raíz.

Bastará entonces con un segundo recorrido en el que deberá verificarse que el valor de las etiquetas de todos los nodos interiores es 0. En caso contrario, no se verifica la condición buscada.

Ha sido necesaria la implementación de la siguientes funciones:

```
nodo & bintree<int>::postorder_iterator::getNodo(){return elnodo};
```

```
bool esHoja(const bintree<int> & A, const bintree<int>::node &v)
{
  return ( v.left().null()  && v.right().null() );
}
```

La función buscada sería la siguiente:

```
bool pesointerior(bintree<int> a){
    bintree<int>::postorder_iterator it;
    for(it = a.begin_postorder(); it != a.end_postorder(); ++it){
        if(esHoja(a, it.getNodo())){
            bintree<int>::node n = it.getNodo();
            while(!n.parent().null()){
                n = n.parent();
                *n -= *it;
            }
        }
    }

    bool valido = true;
    for(it = a.begin_postorder(); it != a.end_postorder() && valido; ++it){
        if(!esHoja(a, it.getNodo()) && (*it)!=0)
            valido = false;
```

```
    }
    return valido;
}
```

Ejercicio 9

Dada una lista de enteros *l* y 2 listas *seq* y *reemp*, implementar una función `void reemplaza(list <int> & l, list <int> & seq, list <int> & reemp)` que busque todas las secuencias *seq* en *l* y las reemplace por *reemp*.

Ejemplo: Dada lista={1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5}, seq={4, 5, 1} y reemp={9, 7, 3} obtendríamos lista={1, 2, 3, 9, 7, 3, 2, 3, 9, 7, 3, 2, 3, 4, 5}.

La solución que presentamos a continuación se basa en recorrer la lista *l* con un iterador *lectura*. Cada vez que encontremos al primer elemento de *seq*, guardamos esa posición con el iterador *fijador* y vamos comparando la lista *l* con *seq*. Si *seq* está contenida totalmente, la borramos de la lista *l* e insertamos en su lugar la secuencia *reemp*. En caso de que no esté contenida, volvemos a la siguiente posición a la que se quedó fijador.

```
void reemplaza(list<int> & l, list<int> seq, list<int> reemp){
    list<int>::iterator lectura, fijador, secuencia;

    lectura = l.begin();
    while(lectura != l.end()){
        secuencia = seq.begin();

        if(*lectura == *secuencia){
            fijador = lectura;
            bool coincidencia = true;
            while(lectura != l.end() && secuencia != seq.end() && coincidencia){
                if(*lectura != *secuencia){
                    coincidencia = false;
                    lectura = fijador;
                }
                else{
                    lectura++;
                    secuencia++;
                }
            }
            if(coincidencia && (secuencia == seq.end())){ //secuencia contenida
                l.erase(fijador, lectura);
                l.insert(lectura, reemp.begin(), reemp.end());

                //Ajustamos iterador de lectura al final de secuencia
introducida.
                int contador = 1;
                while(contador < reemp.size()){
                    lectura++;
                    contador++;
                }
            }
        }

        lectura++;
    }
```

```
}
```

Implementación alternativa:

```
void reemplaza(list <int> & l, list <int> & seq, list <int> & reemp) {
    bool encontrado;
    list<int>::iterator posicion_guardada;

    for (auto it_l = l.begin(); it_l != l.end(); it_l++) {
        encontrado = true;

        if ( *it_l == *seq.begin() ) {
            posicion_guardada = it_l;

            // Comprobamos que todos los siguientes son idénticos
            for (auto it_seq = seq.begin(); it_seq != seq.end() && encontrado;
it_seq++) {
                if (*it_seq != *it_l)
                    encontrado = false;

                it_l++;
            }

            it_l = posicion_guardada;

            if (encontrado) {
                for (auto it_reemp = reemp.begin(); it_reemp != reemp.end();
it_reemp++) {
                    *it_l = *it_reemp;
                    it_l++;
                }
            }
        }
    }
}
```

Ejercicio 10

Implementar una función `int sumaparante(bintree <int > a, node n)` que devuelva la suma de las etiquetas de los valores tales que su etiqueta y la etiqueta de todos los antecesores es par. Si la etiqueta de n es impar devuelve 0.

Con la llamada `sumaparante(arbol, arbol.root())` obtendremos la suma buscada. Se calculará de manera recursiva.

```
int sumaparante(bintree<int> a, bintree<int>::node n){
    int suma;
    if(!n.null()){
        if((*n)%2==0)
            suma = *n + sumaparante(a, n.left()) + sumaparante(a, n.right());
        else
            suma = 0;
    }
    else
        suma = 0;

    return suma;
}
```

Ejercicio 11

Implementar una función `node cont_hijos(list <int> l, bintree <int> a)` que devuelva un nodo m en el árbol *a* cuyas etiquetas de hijos (de izquierda a derecha) coinciden con los elementos de la lista *l*. Si no hay ningún nodo así devuelve nodo nulo.

```
bintree<int>::node cont_hijos(list<int> l, bintree<int> a){
    bintree<int>::node solucion;
    bool encontrado = false;
    bintree<int>::preorder_iterator it;
    for(it = a.begin_preorder(); it != a.end_preorder() && !encontrado; ++it){
        if(!it.getNodo().left().null() && !it.getNodo().right().null()){
            if((*it.getNodo().left() == *l.begin())
                && (*it.getNodo().right() == *(++l.begin()))){

                    solucion = it.getNodo();
                    encontrado = true;
                }
        }
    }
    return solucion;
}
```

Ha sido necesario implementar:

```
nodo & bintree<int>::preorder_iterator::getNodo(){return elnodo};
```

Ejercicio 12

Dadas dos listas de enteros `l1` y `l2`, implementar una función `bool check_sum(list <int> l1, list <int> l2)` que devuelva true si los elementos de `l1` pueden agruparse sumando de forma que se puedan obtener los elementos de l2 sin alterar el orden de los elementos.

Para resolverlo, iremos restando a los elementos de *l2* los elementos de *l1* hasta que vayan quedando cero. Se verificará la condición del enunciado si se agotan todos los elementos de *l1* y al llegar al final, la lista *l2* tiene todos sus elementos hechos cero.

```
bool check_suma(list<int> l1, list<int> l2){
    list<int>::iterator it1, it2;
    it1 = l1.begin();
    it2 = l2.begin();
    list<int> comparar(l2.size(), 0);

    while(it1 != l1.end() && it2 != l2.end()){
        if(*it2 != 0){
            *it2 -= *it1;
            it1++;
        }
        else
            it2++;
    }

    return (it1 == l1.end()) && (comparar == l2);
}
```

Ejercicio 13

Dado un árbol binario A y un nodo n, implementar una función `bool camino_ordenado (bintree<int> a, node n)` que devuelva `true` si existe algún camino desde n hasta una hoja con los elementos ordenados
Por ejemplo, en el siguiente árbol:

```
     3
   /   \
  6     7
 / \   / \
1   3 9   5
```

3, 7 y 9 es un camino ordenado.

```
bool camino_ordenado (bintree<int> arbol, bintree<int>::node nodo) {
    bool camino_izquierda = false, camino_derecha = false;

    if (nodo.right().null() && nodo.left().null())
        return true;
    else {
        if ( !nodo.right().null() && *nodo.right() > *nodo )
            camino_derecha = camino_ordenado(arbol, nodo.right());

        if ( !nodo.left().null() && *nodo.left() > *nodo )
            camino_izquierda = camino_ordenado(arbol, nodo.left());

        return camino_derecha || camino_izquierda;
    }
}
```

Ejercicio 14

Implementar una función `bool anagrama ( list< int >& l1, list < int >& l2 )` que devuelve si `l1` y `l2` tienen la misma cantidad de elementos y los elementos de `l1` son una permutación de los de `l2`

Ejemplos:

```
l1 = {1, 23, 21, 4, 2, 3, 0} | l1 es una permutación
l2 = {21, 1, 3, 2, 4, 23, 0} |       de l2

l1 = {1, 3, 5} | false
l2 = {4, 5, 4} |
```

> Restricciones: Si hay elementos repetidos, tiene que estar el mismo número de veces en ambas listas. No se pueden usar estructuras auxiliares. El algoritmo debe ser como máximo O(n^6 2). Puede ser destructivo. No puede usarse ningún tipo de ordenación

```
bool anagrama ( list< int >& l1, list < int >& l2 ) {
    for ( auto num: l1 ) {
        if ( find( l2.begin(), l2.end(), num) == l2.end() )
            return false;

        if (   count( l1.begin(), l1.end(), num )
            != count( l2.begin(), l2.end(), num ) )

            return false;
    }

    return true;
}
```

Ejercicio 15

Implementar una función `void flota_pares( stack<int> & p )` que recorre los elementos de una pila de forma que los elementos pares quedan arriba de los impares. Los elementos pares deben quedar en el mismo orden entre sí y lo mismo para los impares.

> Restricciones: Pueden usarse pilas auxiliares (y solo pilas). La función debe ser O(n)

Ejemplo:

```
p = {4, 17, 9, 7, 4, 2, 0, 9, 2, 17}    =>    p = {4, 4, 2, 0, 2, 17, 9, 7, 9,
17}
```

```
void flota_pares( stack<int> & p ) {
    stack<int> pares, impares;
    int elemento;

    while ( !p.empty() ) {
        elemento = p.top();

        if ( elemento % 2 == 0 )
            pares.push(elemento);
        else
            impares.push(elemento);
```

```
        p.pop();
    }

    while ( !pares.empty() ) {
        p.push( pares.top() );
        pares.pop();
    }

    while ( !impares.empty() ) {
        p.push( impares.top() );
        impares.pop();
    }
}
```

Ejercicio 16

Implementar una función `void creciente (queue<int> & q)` que elimine elementos de q de forma que los elementos que queden estén ordenados de forma creciente.

> Restricciones: Pueden usarse colas auxiliares (y solo colas). La función debe ser O(n)

Ejemplo:

```
q = {5, 5, 9, 13, 19, 17, 16, 20, 19, 21}    =>    q = {5, 5, 9, 13, 19, 20, 21}
```

```
void creciente (queue<int> & q){
  queue<int> q2;
  int n;

  n = q.top();
  q2.push(n);
  q.pop();

  while (!q.empty ()){
    n = q.top();

    if (n >= q2.top()){
      q2.push(n);
      q.pop();
    }
  }

  while (!q2.empty()){
    n = q2.top();
    q.push(n);
  }
}
```

Ejercicio 17

Sea L una lista conteniendo los elementos (4, 7, 6, 5, 3). Se aplica sobre ella:

```
list<int>:: iterator p, q;
p = L.begin();
q = ++p;
p = ++q;
p = L.erase(p);
```

¿Cuál es verdadera?
a. *p = *q = 7

b. *p = 5, *q = 7

c. *p = 5, q inválido

d. p, q inválidos

Respuesta: c

Ejercicio 18

Implantar una función

```
bool depth_if(const ArbolBinario<int> &a)
```

que devuelva `true` si el nodo a mayor profundidad en `a` tiene etiqueta par y `false` en caso contrario.

Una posible estrategia es crear una función que retorne la hoja en mayor profundidad (si hay varios, retorna el primero de ellos), que utilizaremos fuertemente en `depth_if`. Para dicha fución, crearemos una auxiliar recursiva que va recorriendo todos los nodos del árbol, y cuando llega a una hoja realiza las comprobaciones necesarias.

```
void hoja_mayor_profundidad_aux(const ArbolBinario<int> &a, const
ArbolBinario<int>::nodo &nod, int profundidad, int &max_prof,
ArbolBinario<int>::nodo &nodo_max_prof) {
    if ( !nod.nulo() )
        if ( nod.hi().nulo() && nod.hd().nulo() ) {
            // estamos ante un nodo hoja, comprobamos
            if ( profundidad > max_prof ) {
                max_prof = profundidad;
                nodo_max_prof = nod;
            }
        } else {
            if ( !nod.hi().nulo() )
                hoja_mayor_profundidad_aux(a, nod.hi(), ++profundidad, max_prof,
nodo_max_prof);
            if ( !nod.hd().nulo() )
                hoja_mayor_profundidad_aux(a, nod.hd(), ++profundidad, max_prof,
nodo_max_prof);
        }
}

ArbolBinario<int>::nodo hoja_mayor_profundidad(const ArbolBinario<int> &a) {
    ArbolBinario<int>::nodo nod;
    int max_prof = 0;
    hoja_mayor_profundidad_aux(a, a.getRaiz(), 0, max_prof, nod);
    return nod;
```

```
}

bool depth_if(const ArbolBinario<int> &a) {
    ArbolBinario<int>::nodo hijo_mas_prof = hoja_mayor_profundidad(a);
    if ( *hijo_mas_prof % 2 == 0 )
        return true;
    else
        return false;
}
```

NOTA: *en esta resolución, es posible que aunque haya una hoja que esté a igual profundidad que otra y que sea par, pero que la hoja que se retorne sea impar. Podría modificarse el código de la función* `hoja_mayor_profundidad_aux` *para que de prioridad a las hojas pares, véase:*

```
void hoja_mayor_profundidad_aux(const ArbolBinario<int> &a, const
ArbolBinario<int>::nodo &nod, int profundidad, int &max_prof,
ArbolBinario<int>::nodo &nodo_max_prof) {
    if ( !nod.nulo() )
        if ( nod.hi().nulo() && nod.hd().nulo() ) {
            // cambiamos las comprobaciones para reemplazar en caso de que la
hoja tenga la misma profundidad que
            // la mejor, pero sea par
            if ( (profundidad > max_prof) || ( (profundidad == max_prof) &&
(*nod % 2 == 0) )) {
                max_prof = profundidad;
                nodo_max_prof = nod;
            }
        } else {
            if ( !nod.hi().nulo() )
                hoja_mayor_profundidad_aux(a, nod.hi(), ++profundidad, max_prof,
nodo_max_prof);
            if ( !nod.hd().nulo() )
                hoja_mayor_profundidad_aux(a, nod.hd(), ++profundidad, max_prof,
nodo_max_prof);
        }
}
```

Ejercicio 19

Implementar una función

```
bool en_todos(const vector<set<int> > &v)
```

que devuelva `true` si existe al menos un elemento que pertenece a todos los conjuntos `v[j]`.

```
bool en_todos(const vector<set<int> > &v) {
    bool esta = false;

    if ( v.size() > 1 ) {
        for ( it = v[0].begin(); it != v[0].end() && !esta; ++it ) {
            bool continuar = true;
            for ( int i = 1; i < v.size() && continuar; ++i ) {
                if ( !v[i].find(*it) )
```

```
                    continuar = false;
            }
            if ( continuar )
                esta = true;
        }
    } else
        esta = true;  // sólo hay un set, o ninguno

    return esta;
}
```

Ejercicio 20

Implementar una función

```
bool es_neg(const set<int> &A, const set<int> &B)
```

que devuelve `true` si el conjunto `B` contiene los de `A` pero cambiados de signo.

Podemos usar un `set` que contenga los elementos opuestos de `B`. Bastará comprobar que el set `A` y el nuevo set son iguales.

```
bool es_neg(const set<int> &A, const set<int> &B) {
    set<int> opB;
    set<int>::iterator it, it2;
    bool es = true;

    // comprobamos que tengan el mismo tamaño
    if ( A.size() != B.size() )
        return false;

    // creamos el set de opuestos de B
    for ( it = B.begin(); it != B.end(); ++it )
        opB.insert(-1*(*it));

    it = A.begin(); it2 = opB.begin();

    while ( it != A.end() && es ) {     // cuando lleguemos al fin de uno,
habremos llegado al fin del otro
        if ( *it != *it2 )
            es = false;
        ++it; ++it2;
    }

    return es;
}
```

Ejercicio 21

Implementar una función

```
void apply_map(const list<int> &L, const map<int,int> &M, list<int> &ML)
```

que dada una lista `L` y un map `M` devuelva en `ML` el resultado de aplicar a `M` los elementos de `L`. Si algún elemento de `L` no está en el dominio de `M`, no se incluye en `ML` el elemento correspondiente.

> **Restricciones:** No pueden usarse estructuras auxiliares y la función ha de ser *O(n)*.

> **Ejemplo:**
>
> - L = {1,2,3,4,5,6,7,1,2,3}
> - M = {(1,2), (2,3), (4,5), (7,8)}
> - ML = {2,3,5,8,2,3}

```
void apply_map(const list<int> &L, const map<int,int> &M, list<int> &ML) {
    list<int>::iterator itl;

    for ( itl = L.begin(); itl != L.end(); ++itl )
        if ( M.find(*itl) != M.end() )
            ML.insert(M[*itl]);
            // equivalentemente, podría haberse hecho
            // ML.insert(M.find(*itl)->second);
}
```

Ejercicio 22

Implementar una función

```
void longest_path(const bintree<int> &A, list<int> &L)
```

que dado un árbol binario `A` devuelva en `L` la lista de valores en el camino más largo en el árbol. Si hay más de uno se devuelve cualquiera de ellas.

```
void longest_path_aux(const ArbolBinario<int> &A, list<int> &L, const
ArbolBinario<int>::nodo &n, list<int> camino) {
    if ( !n.nulo() ) {
        if ( n.hi().nulo() && n.hd().nulo() ) {
            // estamos en una hoja: vemos si el camino es mayor
            if ( camino.size() > L.size() )
                L = camino;
        } else {
            if ( !n.hi().nulo() ) {
                list<int> new_camino = L;
                new_camino.push_back(*n.hi());
                longest_path_aux(A, L, n.hi(), new_camino);
            }
            if ( !n.hd().nulo() ) {
                list<int> new_camino = L;
                new_camino.push_back(*n.hd());
                longest_path_aux(A, L, n.hd(), new_camino);
            }
        }
    }
}

void longest_path(const ArbolBinario<int> &A, list<int> &L) {
    list<int> camino;
```

```
    ArbolBinario<int>::nodo n = A.getRaiz();
    camino.push_back(n);
    longest_path_aux(A, L, n, camino);
}
```

Ejercicio 23

Implementar una función

```
void path_of_largest(const bintree<int> &A, list<int> &L)
```

que dado un árbol binario `A`, devuelva en `L` el camino que se obtiene recrriendo el árbol desdela raíz a las hojas siempre con el hijo con mayor valor de etiqueta. Si para un nodo el valor más grande está repetido, entonces el camino puede seguir por cualquiera de ellos.

```
void path_of_largest_aux(const ArbolBinario<int> &A, list<int> &L, const
ArbolBinario<int>::nodo &nod) {
    if ( !n.nulo() ) {
        if ( !n.hi().nulo() && !n.hd().nulo() ) {
            // tiene ambos hijos: seguimos por el mayor de ellos
            if ( *(n.hi()) > *(n.hd()) ) {
                L.push_back(*(n.hi()));
                path_of_largest_aux(A, L, n.hi());
            }
            else if ( *(n.hi()) < *(n.hd()) ) {
                L.push_back(*(n.hd()));
                path_of_largest_aux(A, L, n.hd());
            }
        } else if ( !n.hi().nulo() ) {
            // tiene solo hijo izquierdo: seguimos por él
            L.push_back(*(n.hi()));
            path_of_largest_aux(A, L, n.hi());
        } else if ( !n.hd().nulo() ) {
            // tiene solo hijo derecho: seguimos por él
            L.push_back(*(n.hd()));
            path_of_largest_aux(A, L, n.hd());
        }
    }
}

void path_of_largest(const ArbolBinario<int> &A, list<int> &L) {
    path_of_largest_aux(A, L, A.getRaiz());
}
```

Ejercicios adicionales

Ancestro común más cercano (AMC)

sea *A* un árbol binario con *n* nodos. Se define el **ancestro común más cercano (AMC)** entre dos nodos `r` y `w` como el ancestro de mayor profundidad que tiene tanto a `r` como a `w` como descendientes (se entiende como caso extremo que un nodo es descendiente de sí mismo). Diseñar una función para hacer esto.

```
template<typename T>
int getProfundidad(const ArbolBinario &arb, ArbolBinario<T>::nodo n) {
    int profundidad = 0;

    while ( n != arb.getRaiz() ) {
        n = n.padre();
        profundidad++;
    }

    return profundidad;
}

template<typename T>
ArbolBinario<T>::nodo AMC(const ArbolBinario<T> &arb, ArbolBinario<T>::nodo r,
ArbolBinario<T>::nodo w) {
    ArbolBinario<T>::nodo ancestro;

    // primero hacemos que ambos estén en el mismo nivel, entonces
    // encontrar el ancestro común más cercano será ir subiendo en los
    // padres de cada uno hasta que coincidan
    int prof_r = getProfundidad(arb, r), prof_w = getProfundidad(arb, w);
    if ( prof_r < prof_w )
        for ( int i = 0; i < prof_w-prof_r; ++i )
            w = w.padre();
    else if ( prof_r > prof_w )
        for ( int i = 0; i < prof_r-prof_w; ++i )
            r = r.padre();

    // caso límite
    if ( r == w )
        return r;

    bool continuar = true;
    while ( continuar ) {
        r = r.padre();
        w = w.padre();

        if ( r == w ) {
            ancestro = r;
            continuar = false;
        }
    }

    return ancestro;
}
```

TDA Documento

TDA Documento tiene en su representación una tabla hash en la que cada palabra del documento tiene asociada una lista ordenada con las posiciones en las que aparece la palabra en el mismo.

Implementar función

```
int Documento::min_distancia(string pal1, string pal2);
```

que devuelva la distancia mínima en la que aparecen las palabras `pal1` y `pal2` en el documento. Para la representación de la tabla hash se usa hashing abierto.

```
class Documento {
    map<string, list<int> > hash;
private:
    int min_distancia(string pal1, string pal2) {
        list<int> pal1_h = hash[pal1];
        list<int> pal2_h = hash[pal2];

        list<int>::iterator it1, it2;
        int minimo = numeric_limits<int>::max();

        for ( it1 = pal1_h.begin(); it1 != pal1_h.end(); ++it1 )
            for ( it2 = pal2_h.begin(); it2 != pal2_h.end(); ++it2 ) {
                int dif = abs(*it1-*it2);
                if ( dif < minimo )
                    minimo = dif;
            }

        return minimo;
    }
};
```

Árbol binario hilvanado

Un árbol binario hilvanado es una ED para la representación de un ABB que facilita ciertos recorridos. En el registro que define un nodo se añade un puntero `hilvan` que apunta al siguiente nodo en el recorrido en inorden del ABB. Diseñar un procedimiento para insertar un nodo en este árbol.

> **Bonus:** aquí aparece la función de inserción en un ABB cualquiera

```
class ABBhilvan {
    ArbolBinario<T> arb;
    // suponemos implementado el nodo con hilvan
public:
    void insertar(T elem) {
        // es una inserción similar, pero modificamos hilvan
        ArbolBinario<T>::nodo n = arb.getRaiz(), padre;
        while ( !n.nulo() ) {
            if ( elem == *n )
                raise_error();  // no puede existir el mismo nodo

            padre = n;
            if ( elem < *n )
                n = padre.hi();
            else if ( elem > *n )
                n = padre.hd();
        }

        n = ArbolBinario<T>::nodo(elem);
        if ( padre.nulo() )            // el árbol estaba vacío
            pass;
        else if ( elem < *padre ) {    // insertamos a izquierda de padre
```

```
            n.hilvan = padre;
        } else if ( elem > *padre ) {  // insertamos a derecha de padre
            n.hilvan = padre.hilvan;
            padre.hilvan = n;
        }
    }

    // algoritmo para insertar en ABB (sin hilvan)
    void insertar(T elem) {
        insertar_nodo(arb.getRaiz(), elem);
    }
private:
    void insertar_nodo(ArbolBinario<T>::nodo n, T elem) {
        if ( n.nulo() ) {
            n = ArbolBinario<T>(elem);
        } else {
            if ( *n < elem )
                insertar_nodo(n.hi(), elem);
            else
                insertar_nodo(n.hd(), elem);
        }
    }
};
```