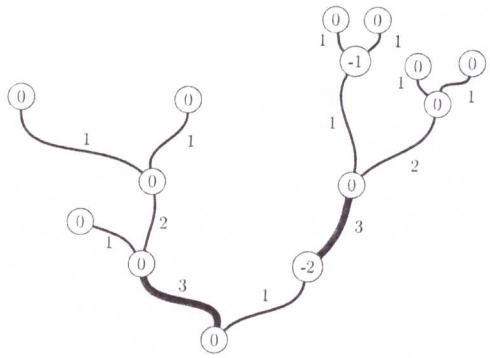
Estructuras de Datos y Algoritmos

Doble Grado, Ingeniería Informática, de Computadores y del Software

Examen Parcial, 23 de Junio de 2016.

1. **(2 puntos)** La compañía que gestiona la cuenca hidrográfica del río *Aguas Limpias* nos ha pedido que calculemos los tramos navegables de dicho río. Un tramo del río es navegable si contiene un caudal mayor o igual a 3 m^3/s. Como todos los ríos, el *Aguas Limpias* está formado por una serie de afluentes. Los afluentes que nacen de los manantiales llevan un caudal de un 1 m^3/s. Se considera que siempre confluyen exactamente dos afluentes. Cada vez que dos afluentes confluyen se calcula el caudal del tramo resultante como la suma de los caudales de sus afluentes. Adicionalmente, a lo largo del río se han construido varios embalses que hacen decrecer en una determinada cantidad el caudal del tramo saliente. En un embalse pueden confluir dos afluentes o solamente un tramo del río. La figura que se muestra a continuación es un ejemplo de un río que contiene 2 tramos navegables.

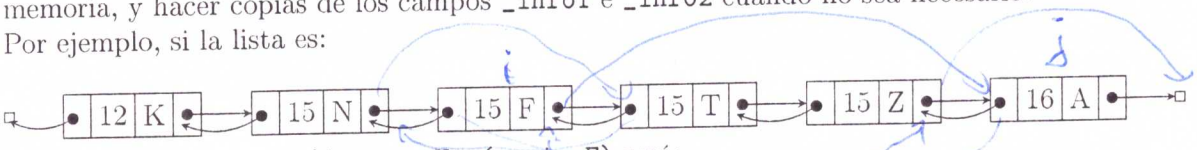


La estructura de afluentes se representa como un árbol binario cuyos nodos internos son o bien puntos de encuentro de dos afluentes, en cuyo caso el nodo contiene un 0, o bien un embalse, en cuyo caso el nodo contiene un número negativo que representa cuanto caudal puede absorber el embalse. Nótese que el caudal de un tramo no puede ser negativo pero sí 0. Implementa una función que dado un árbol que representa la estructura del río, calcule el número de tramos navegables.

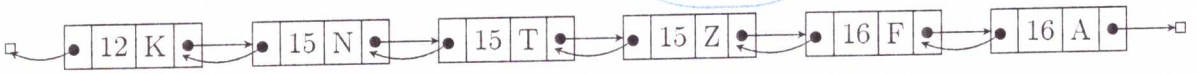
2. **(2,5 puntos)** Se tiene una lista doblemente enlazada con punteros al primer y último nodo, no circular, sin nodo cabecera, cuyos nodos tienen un campo `_info1` de tipo `int` y un campo `_info2` de tipo `T2`. La lista está parametrizada respecto al tipo `T2`:

```
template <class T2>
class ListaEnlazadaDoble{
private:
    class Nodo {
    public:
        Nodo() : _ant(NULL), _sig(NULL) {}
        Nodo(int n, const T2 &elem) : _info1(n), _info2(elem), _sig(NULL), _ant(NULL) {}
        Nodo(Nodo* ant, int n, const T2 &elem, Nodo *sig) : _info1(n), _info2(elem),
            _sig(sig), _ant(ant) {}
        int _info1;     T2 _info2;
        Nodo *_sig;    Nodo *_ant;
    };
    Nodo * _prim;   Nodo* _ult;
    ...
}
```

Los nodos de la lista se encuentran ordenados en orden creciente por el campo `_info1`, pudiendo existir muchos nodos con el mismo valor en dicho campo pero no nodos con el mismo valor en el campo `_info2`.
Se pide implementar una función `sumarUno` que reciba un puntero al comienzo de la lista enlazada y un valor de tipo `T2`, busque este valor en la lista y sume uno a su campo `_info1`. Si es necesario debe modificarse la lista para que siga estando ordenada por el campo `_info1`, situando el nodo a continuación de todos aquellos que tuviesen su mismo valor en el campo `_info1` antes de incrementar su valor. Si el valor dado no existe, la lista no se modifica.
La implementación debe ser lo mas eficiente posible. Para ello, se debe evitar liberar y reservar memoria, y hacer copias de los campos `_info1` e `_info2` cuando no sea necesario.
Por ejemplo, si la lista es:



el resultado de la operación `sumarUno(_prim,F)` será:



3. **(4,5 puntos)** Los ingenieros de la empresa Apel están actualmente diseñando el software para su nuevo reproductor de música *iPud*, el cual debe permitir añadir y eliminar canciones, añadir una canción existente a la lista de reproducción, avanzar a la siguiente canción, obtener el tiempo total de la lista de reproducción, y obtener una lista con las canciones escuchadas recientemente. En particular, el TAD `IPud` debe contar con las siguientes operaciones:

   - `create`: Crea un iPud vacío.
   - `addSong(IP,S,A,D)`: Añade la canción `S` (`string`) del artista `A` (`string`) con duración `D` (`int`) al iPud `IP`. Si ya existe una canción con el mismo nombre la operación dará error.
   - `addToPlaylist(IP,S)`: Añade la canción `S` al final de la lista de reproducción. Si la canción ya se encontraba en la lista entonces no se añade (es decir, la lista no tiene canciones repetidas). Si la canción no está en el iPud se devuelve error.
   - `deleteSong(IP,S)`: Elimina todo rastro de la canción `S` del iPud `IP`. Si la canción no existe la operación no tiene efecto.
   - `play(IP)`: La primera canción de la lista de reproducción abandona la lista de reproducción y se registra como reproducida. Si la lista es vacía la acción no tiene efecto.
   - `current(IP)`: Devuelve la primera canción de la lista de reproducción. Si la lista de reproducción es vacía se devuelve error.
   - `totalTime(IP)`: Obtiene la suma de las duraciones de las canciones que integran la lista de reproducción actual. Si es vacía se devuelve 0.
   - `recent(IP,N)`: Obtiene la lista con las `N` últimas canciones que se han reproducido (mediante la operación `play`), de la más reciente a la más antigua. Si el número de canciones reproducidas es menor que `N` se devolverán todas. La lista no tiene repeticiones, de manera que si una canción se ha reproducido más de una vez solo figurará la reproducción más reciente.

   Se puede asumir que las canciones quedan unívocamente determinadas por su nombre (`string`).

Se ha elegido la siguiente representación para el TAD `IPud` usando los TADs vistos en clase:

```
typedef struct{
  string artist;
  int duration;
  bool inPlaylist; //indica si esta en la lista
  bool played; //indica si esta en la lista de reproducidas
} SongInfo;
class iPud{
  public:
    ... las operaciones pedidas...
  private:
    HashMap<string, SongInfo> songs;
    List<string> playlist; //la lista de reproduccion, sin repeticiones
    List<string> played; //las ya reproducidas, sin repeticiones
    int duration;
}
```

Se pide:

1. **(1 punto)** Indicar el coste de cada una de las operaciones con esta representación e implementar la operación `play`.
2. **(1 punto)** Modificar la representación anterior para que todas las operaciones tengan coste constante, excepto `recent` que no debe superar el coste lineal en N.
3. **(2,5 puntos)** Supongamos que se desea extender la funcionalidad del IPud para permitir manejar múltiples listas de reproducción, identificadas mediante un nombre unívoco. En particular se dispondría de las operaciones:
   - `saveCurrentList`, que guarda la lista de reproducción actual con el nombre proporcionado;
   - `generateArtistList`, que genera una nueva lista de reproducción con las canciones de un artista (si es que hay alguna), y la almacena con el propio nombre del artista;
   - `setPlaylist` que pone como lista actual la lista proporcionada.
   - `allList` que devuelve una lista con los nombres de las listas de reproducción almacenadas.

   Rediseña la representación del TAD de manera que se minimice el coste de las nuevas operaciones. Indica y justifica el coste esperado de las nuevas operaciones y de las antiguas que se hayan visto modificadas.

   Implementa con dicha representación la operación `generateArtistList`.

4. **(1 punto)** Responde a las siguientes cuestiones (0.2 puntos cada una):

4.1) Dada una lista enlazada simple con un puntero al primer nodo y un puntero al último, como la utilizada en la implementación de las colas dada en clase, indicar y justificar cuál de las siguientes operaciones no puede realizarse en tiempo constante.

a) Añadir un elemento al comienzo de la lista

b) Borrar el elemento del comienzo de la lista

c) Añadir un elemento al final de la lista

d) Borrar el elemento del final de la lista

4.2) Sea un algoritmo de ordenación consistente en, dado un vector, insertar todos los elementos en un árbol binario de búsqueda, y posteriormente obtener en el vector original el recorrido en inorden del árbol. Se pide:

a) Indicar el coste en tiempo y en espacio en el caso peor de dicho algoritmo.

b) Lo mismo pero suponiendo que se usa un árbol con lógica de re-equilibrado.

4.3) ¿Qué recorrido de un árbol binario de búsqueda de enteros se debe guardar para que pueda construirse posteriormente un árbol idéntico al inicial? Se debe tener en cuenta que sólo se guardarán los enteros contenidos en los nodos sin ninguna información adicional.

a) El preorden

b) El inorden

c) El recorrido por niveles

d) El preorden o el recorrido por niveles

e) No es posible reconstruirlo igual que estaba con un solo recorrido

4.4) El coste de la operación insertar en una tabla hash es

1. en el caso peor lineal en el número de elementos de la tabla
2. en el caso peor logarítmico en el número de elementos de la tabla
3. en el caso promedio constante
4. en el caso promedio logarítmico en el número de nodos de la tabla

Son correctas:

a) 1 y 3

b) 1 y 4

c) 2 y 3

d) 2 y 4

4.5) Supongamos un `HashMap` con claves `int` y valores `char`, con tamaño inicial 5, cuya función hash se define como $hash(k) = 2 * k$, y que incluye equipamiento para duplicar su tamaño cuando se supera el 80% de tasa de ocupación (como la implementación vista en clase). Inicialmente está vacío y se insertan los pares <7,a>, <10,b>, <2,c>, <5,d>, <4,e> en ese orden.

a) Dibuja el estado final de la tabla. ¿Cuál es la tasa de ocupación?

b) ¿Qué problema tiene esta función hash sobre esta tabla?