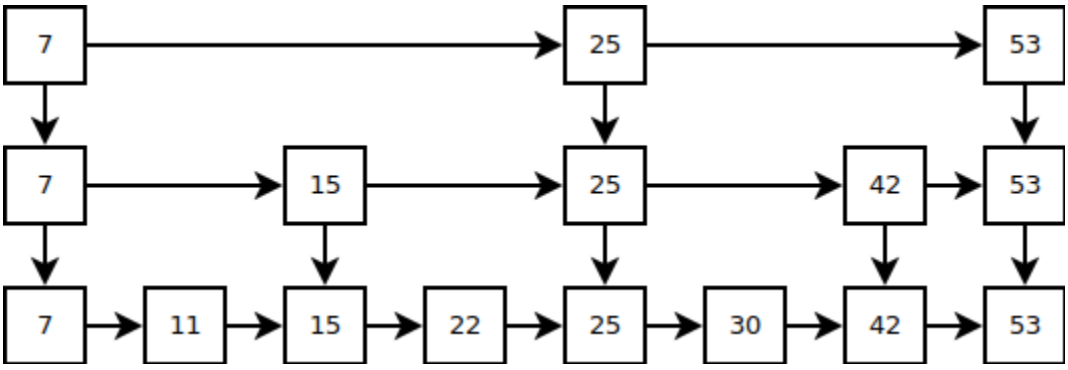
Skip List.

Esta es una estructura de datos diferente a la que hemos visto anteriormente debido a que tiene el carácter de ser *randomizada*. Para meternos en el tema, primero es importante recordar la estructura de Lista Ligada para entender en su totalidad el funcionamiento de esta nueva estructura.

En esta estructura tenemos una lista "**base**" u original, la cual contiene *n* elementos. La pregunta obligada es: ¿cuánto nos tardamos en encontrar a un elemento en la estructura? La respuesta es sencilla: *O(n)*. ¿Por qué? Porque solo podemos acceder a los elementos de la lista de uno en uno. ¿Cómo podríamos hacerla el doble de rápido? Podemos crear una lista **auxiliar** que recorra los elementos saltándose uno ["uno sí y uno no"] y que esté ligada con la lista base. Esto hace que en la lista auxiliar se pueda acceder a los elementos el doble de rápido. La idea ahora es que para buscar un elemento, recorremos la lista auxiliar en la búsqueda del mismo y si lo encontramos, podemos ser felices. ¿Qué sucede si no lo encontramos? Si el elemento en el que nos encontramos es más grande, simplemente bajamos a la lista base para ahora "retroceder" en ella y buscar al elemento; nos vamos a detener cuando el elemento en el que nos encontremos sea menor al que buscamos y eso significa que el elemento no se encuentra en la estructura. También significa que no encontramos el elemento si llegamos al centinela de cola.

La idea entonces es tener una lista base con *n* elementos y tener *m* listas que cada vez tengan menos elementos en su interior para recorrer de manera más rápida "saltándose" elementos y que estén ligadas entre ellas. Un ejemplo de cómo se vería visualmente la estructura es como la siguiente:



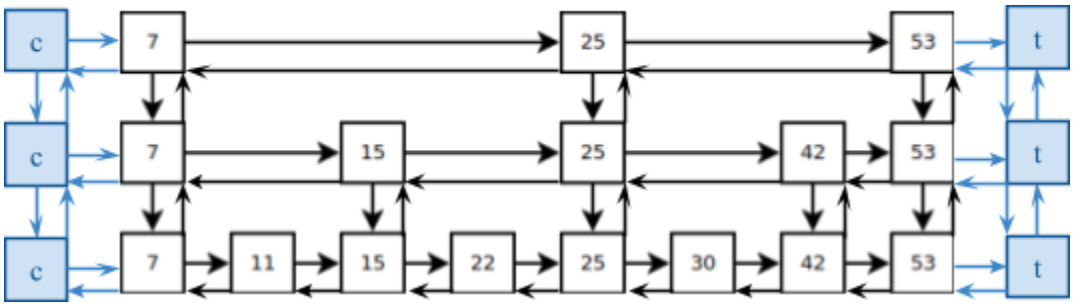
Cabe mencionar que en esta imagen faltan los **centinelas** de la cabeza y la cola para cada una de las listas, además de que cada nodo tiene que estar **doblemente ligado** para cada uno de sus vecinos: el de su izquierda, el de su derecha, el de arriba y el de abajo.

La pregunta del millón: ¿cuántas listas máximo podemos tener en una estructura con *n* datos? La respuesta es igual a muchas de las respondidas en este documento: *log(n)*. ¿Por qué? Porque cada lista auxiliar que agregamos, estamos eliminando la mitad de sus elementos, entonces estamos trabajando con mitades. Aquí hay una similitud con los árboles binarios de búsqueda.

¿Cuáles son sus **operaciones**? En un Skip List se puede buscar, insertar o borrar un elemento.

APUNTE NO. 19 Lunes 12 de abril

Algunas implementaciones en código importantes.



1. La **entrada** a la estructura va a ser por medio del centinela de la **cabeza** de la lista más "superficial" de la estructura.
2. Se va a diseñar un método de **búsqueda** que regresa como resultado el nodo si el elemento está en la estructura o regresa el nodo anterior a donde debería de estar si es que no se encuentra.

Buscar un elemento en un SkipList.

Nos adentramos en la estructura por medio de la lista auxiliar más "superficial". Se va a ir recorriendo esta lista hasta que suceda cualquiera de estos tres casos: si encontramos al elemento, bajamos hasta la lista base y bailamos; si nos encontramos con un elemento más grande que el que estamos buscando, nos movemos al nodo anterior y bajamos uno para seguir recorriendo para "adelante" pero ahora con la lista inferior. Así seguimos recorriendo para adelante, uno para atrás y uno para abajo para encontrar al elemento. Si en algún momento llegamos al centinela de cola, significa que no se encuentra el elemento en la estructura. Este método [tanto en código como en palabras] tiene que llegar a la lista base.

Borrar un elemento en SkipList.

Para empezar, buscamos al elemento en la estructura. Recordemos que este método nos devuelve [de nuevo, ya sea en código o en palabras] al elemento en la lista base si es que lo encontró. Supongamos que sí lo encuentra ¿Qué se hace ahora? Es sencillo: ligamos doblemente al elemento que se encuentra a su derecha con el que se encuentra a su izquierda y subimos repitiendo este doble ligamiento con los vecinos. Nos detenemos hasta que ya no podamos subir más listas [y esto puede ser porque ya se acabaron o porque el nodo ya no se encuentra en la lista de arriba].

Dado que si hay *n* elementos solamente puede haber *log(n)* listas, es importante considerar que puede ser necesario borrar alguna de las listas auxiliares si la cantidad de ellas es mayor a *log(n)*. En este caso, colapsar significa borrar la lista de hasta arriba.

Si nos damos cuenta, la estructura poco a poco se va a desacomodar conforme se vayan eliminando más elementos, entonces es necesario considerar en algún momento un método que puedan reestructurar las listas auxiliares y que se intente mantener la propiedad de tener *log(n)* listas máximas. Este momento de reestructura lo podemos considerar cuando en algún método se tengan que recorrer los *n* elementos en la lista base, como por ejemplo en el toString.

APUNTE NO. 20 Miércoles 14 de abril

Método de **inserción** en SkipList.

Primero buscamos el lugar en donde debe de estar el elemento en la lista base. Supongamos que ya encontramos ese lugar, ¿qué hacemos? Simplemente ligamos doblemente el que debería de estar a su izquierda y el que debería de estar a su derecha. Ahora bien, ¿consideramos si lo podemos subir a la lista auxiliar o no? Sí, porque si no lo hacemos, poco a poco tendremos una lista base con muchos elementos y las listas auxiliares ya no nos ayudarían mucho. Para hacerlo divertido, lo podemos hacer de manera aleatoria con un volado: **si sale águila**, lo subimos, **si sale sol**, no lo subimos. Si sale águila, simplemente creamos un nuevo nodo para ligar doblemente a los vecinos correspondientes en la lista auxiliar que se encuentra arriba y volvemos a tirar un volado. Nos detenemos en este curioso procedimiento en el momento en que nos salga un sol. Por lo tanto, una Skip List no es una estructura determinista. También debemos de considerar el caso de expandir una lista más a la estructura. ¿Cuándo debemos de hacer eso? Podemos decir que cuando es necesario tener una lista más por la cantidad de datos existentes en la estructura, es decir, cuando el número ideal de listas es mayor al número existe de ellas. Recordemos que **más o menos queremos mantener la cantidad de listas en *log(n)***.

La pregunta es: ¿a lo más cuántos volados vamos a poder tirar? En el caso en que tengamos la suerte de que siempre salga águila, el número máximo de volados que podemos tirar es de *log(n)*.

Algo importante a tomar en cuenta.

Aquí podemos ver con más detenimiento que en el mejor de los casos, al ir insertando una cantidad *n* de elementos, la estructura va a ser de *O(log(n))* [más o menos], pero que en el peor de los casos esta estructura cumple con un terrible *O(n)*. No podemos asegurar siempre estar en el mejor caso, pero si en algunas cosas se tarda más, en general no está mal. Para la mayoría de las aplicaciones funciona, pero recordemos que no es buena idea utilizar una estructura así para programar el piloto automático de un avión. Se gana mucho al no tener que utilizar métodos de balanceo, esto hace que sea algo más "ligero" que las otras estructuras, pero recordemos siempre la desventaja [que al mismo tiempo es ventaja] de que no es determinista. La esperanza es que va a funcionar muy bien la mayoría de las veces.