Ficha 15

SkipLists (Listas de Salto)

1.] Arboles de búsqueda equilibrados: Equilibrio AVL.

Hemos visto que las listas son muy eficientes y útiles cuando se tiene un conjunto de datos de tamaño desconocido y/o de crecimiento o decrecimiento impredecible. Las listas (tanto simplemente vinculadas como doblemente vinculadas) se adaptan a la perfección a ese contexto, aumentando o disminuyendo su tamaño de acuerdo al volumen de datos disponible y a sus variaciones.

Sin embargo, las *listas* presentan un clara desventaja con relación al uso de *arreglos*: en una lista no se cuenta con la propiedad del *acceso directo* a sus componentes. Salvo para el caso del primer nodo (a lo sumo también para el último si se cuenta con un puntero adicional a ese último nodo) el resto de los nodos sólo puede accederse recorriendo la lista en forma secuencial. Esto hace que muchos de los algoritmos asociados al manejo de listas tengan un *tiempo de ejecución que depende en forma directamente proporcional del número de nodos de la lista*. Por ejemplo: si se tiene una lista simple con *n* nodos, la inserción de un nuevo nodo al final de la lista requerirá recorrer la lista completa, lo cual implica *n* operaciones de avance del puntero de recorrido. De mismo modo, la operación de comprobar si un elemento está en la lista (o sea, una *búsqueda*), también llevará a un recorrido secuencial que en el *peor caso* insumirá *n* comparaciones.

Veremos que, en general, cuando el tiempo de ejecución de un algoritmo en el peor caso está en *proporción directa al tamaño n del conjunto de datos a procesar*, se dice que ese algoritmo tiene un tiempo de ejecución *del orden de n* para el peor caso, y se denota como *O(n)* mediante la ya conocida notación simbólica *O mayúscula* (o *Big O*).

En el campo de la programación de computadoras permanentemente se diseñan y aplican algoritmos nuevos para resolver problemas. Muchas veces se cuenta con varios algoritmos diferentes para resolver el mismo problema. En esos casos es obvio preguntarse cuál de esos algoritmos *es el mejor*. O bien, si solo se cuenta con un algoritmo para un problema dado, es también saludable preguntarse *qué tan bueno* es ese algoritmo y llegado el caso, también es razonable preguntar *si el rendimiento de ese algoritmo se puede mejorar*. El ***Análisis de Algoritmos*** es la rama de las Ciencias de la Computación cuyo objetivo es estudiar y responder ese tipo de preguntas.

Recordemos un caso relativamente simple del mundo del diseño, análisis e implementación de algoritmos: la ***inserción ordenada en una lista simple***. La idea es partir de una lista simple vacía, e insertar *n* elementos en ella pero de forma que en todo momento la lista se mantenga ordenada (digamos, de menor a mayor). Intuitivamente, una lista ordenada ofrece algunas ventajas de procesamiento: siempre es más práctico para un usuario ver una lista ordenada de datos (por ejemplo, para hacer una simple búsqueda visual). Además, en algunos casos saber que la lista está ordenada permite el procesamiento de sus datos en forma más eficiente (por ejemplo, al menos

en teoría, la búsqueda de un elemento podría suponerse más rápida si la lista está ordenada… ¡aunque eso es justamente lo que queremos dicutir aquí!).

En los modelos de programación que acompañan a las Fichas de Estudio sobre listas simples, hemos mostrado la forma de implementar la idea de inserción ordenada mediante el método *addInOrder()* de la clase *SimpleList*. Un análisis rápido del algoritmo *addInOrder()*, muestra que el mismo debe *recorrer secuencialmente* la lista de *n* nodos para encontrar lo que designamos como el *punto de inserción* del nuevo elemento *x* en la lista. Y dado que el recorrido es secuencial, sabemos de inmediato que en el peor caso su tiempo de ejecución será *O(n)*.

Un buen programador (en definitiva, un buen diseñador de algoritmos) debería tener por norma o actitud general el *principio de desconfiar de su primera solución, y preguntarse siempre si hay forma de hacerlo mejor*. Salvo en casos de programadores expertos, normalmente la primera solución que suele plantearse para un problema dado resulta ser la más intuitiva y obvia, pero normalmente esa solución obvia suele ser también ineficiente. Como sabemos, en el campo del Análisis de Algoritmos, la eficiencia de un algoritmo puede medirse en base a distintos criterios de medición,, y los dos criterios más utilizados para esas mediciones son el *tiempo de ejecución* y la *cantidad de memoria usada*, aunque también puede incidir la *complejidad del código fuente*. En el análisis que sigue, supondremos que el factor de eficiencia a medir será el *tiempo de ejecución*. Así, si nos preguntamos si *es posible mejorar la eficiencia del algoritmo de inserción ordenada en una lista simple*, entonces lo que queremos saber es *si hay alguna forma de plantear un algoritmo más rápido*…

Un algoritmo cuyo tiempo de ejecución es *O(n)* para el peor caso, es aquel que dado un conjunto de *n* elementos, recorre una sola vez el conjunto y aplica sobre todos y cada uno de sus elemento un cierto proceso individual. Nuestro algoritmo *addInOrder()* hace eso: en el peor caso, recorre una sola vez toda la lista, y aplica sobre cada nodo una única comparación. Entonces parece claro que si queremos mejorar ese tiempo, debemos pensar una estrategia que permita *ahorrar comparaciones* en forma sistemática. Y ese será nuestro objetivo…

2.] Antecedentes: búsqueda binaria en un arreglo ordenado y el comportamiento logarítmico.

Muchas veces ese tipo de estrategias ganadoras puede plantearse de forma directa sin tener que modificar la estructura interna del conjunto o estructura de datos analizada. Un ejemplo categórico en cuanto al ahorro de comparaciones puede verse en el conocido algoritmo de *Búsqueda Binaria* en un arreglo. Está claro que si tenemos un arreglo unidimensional *v* de *n* componentes, la *búsqueda secuencial* de un valor *x* en ese arreglo también tendrá un tiempo *O(n)* para el peor caso.

Ahora bien: si sabemos que el arreglo *v* está *ordenado* (supongamos de menor a mayor) y que permanecerá ordenado, entonces puede plantearse un algoritmo mucho más veloz conocido como *búsqueda binaria*. La idea es directa: se comienza comparando a *x* (el valor buscado) contra el elemento central *v[c]* del arreglo. Si x == v[c], entonces la búsqueda termina. Pero si *x != v[c]*, entonces se aprovecha el conocimento de que *v* está ordenado: si *x < v[c]*, entonces se desecha la mitad derecha de *v* (ya que esa mitad todos los valores son mayores a *v[c]* y por lo tanto también son mayores a *x*) y se continúa la búsqueda en la mitad izquierda, pero de tal forma que se aplica nuevamente en esa mitad la misma idea: el valor x se compara contra el valor central de la mitad izquierda, y otra vez se desecha la mitad. Obviamente, si en algún momento fuese *x > v[c]*, la

mitad desechada sería la de la izquierda y la búsqueda proseguiría de la misma forma en la mitad derecha.

Podemos ver sin mucho esfuerzo que la búsqueda binaria produce un ahorro importante de comparaciones con respecto a la búsqueda secuencial. La pregunta es: *¿qué tan bueno es el algoritmo de búsqueda binaria en comparación con el de búsqueda secuencial?* Para responder a esa pregunta, digamos que la búsqueda binaria aplica una variante simple de una conocida estrategia del diseño de algoritmos designada como *divide y vencerás*. La idea esencial de esa estrategia es tomar un conjunto de *n* elementos, dividirlo en dos subconjuntos aproximadamente del mismo tamaño, procesar cada subconjunto por separado, y luego unir y/o mezclar los conjuntos resultantes para llegar al resultado final. Más adelante en otras fichas de estudio, analizaremos con más detalle el rendimiento esperado de esta estrategia.

En la búsqueda binaria, el conjunto de *n* elementos se divide en dos mitades del mismo tamaño aproximado *n/2*, se desecha una de las mitades y se procesa la otra de la misma forma. Por consiguiente, la mitad no desechada se volverá a partir en dos nuevas mitades, que ahora serán de tamaño *(n/2)/2 = n/4*, desechando nuevamente una de ellas. Entonces, en cada nueva iteración se hace una comparación de *x* contra el valor central de la mitad rescatada, se vuelve a dividir por 2 a esa mitad, se desecha una de las dos obtenidas y *se prosigue así hasta que ya no sea posible seguir dividiendo por 2* (es decir, hasta llegar a una situación en la que no se cuente con una nueva mitad que pueda volver a dividirse). Si llega ese momento sin haber encontrado a *x*, entonces puede deducirse que *x* no existe en *v*.

De lo anterior se deduce que en cada iteración se hace una comparación y luego se procede a dividir. Entonces la cantidad de comparaciones en el peor caso será igual a la cantidad de veces que se pueda hacer la división por 2 partiendo de *n*. En otras palabras: ¿cuántas veces se puede dividir a *n* por 2, tomar el cociente obtenido, volver a dividirlo por 2, y seguir así hasta llegar a un cociente de 1? Un ejemplo numérico ayudará a intuir la respuesta:

Sea n = 32	Iteración k = 1:	32 / 2 = 16
	Iteración k = 2:	16 / 2 = 8
	Iteración k = 3:	8 / 2 = 4
	Iteración k = 4:	4 / 2 = 2
	Iteración **k = 5**:	2 / 2 = **1**

Vemos que el proceso se detiene en la iteración *k = 5*. Esto es, la cantidad de divisiones que pueden hacerse para *n = 32* es igual a *k = 5*. Pero también podemos ver que $n = 32 = 2^5 = 2^k$ y por lo tanto resulta que la cantidad *k* de divisiones por 2 que pueden hacerse empezando en *n*, es el exponente al que debe elevarse la base 2 para obtener *n*… lo cual es lo mismo que el *logaritmo en base 2 de n*:

Número de divisiones: $k = log_2(n)$

Esto nos permite deducir que el algoritmo de búsqueda binaria en el peor caso hará una cantidad de comparaciones aproximadamente igual a $log_2(n)$ en un vector ordenado de *n* componentes… con lo cual el tiempo de ejecución para ese peor caso es $O(log_2(n))$. En general, se puede probar que en notación O mayúscula la base del logaritmo usado no es relevante, y por lo tanto, podemos generalizar y decir que el peor caso de la búsqueda binaria es $O(log(n))$.

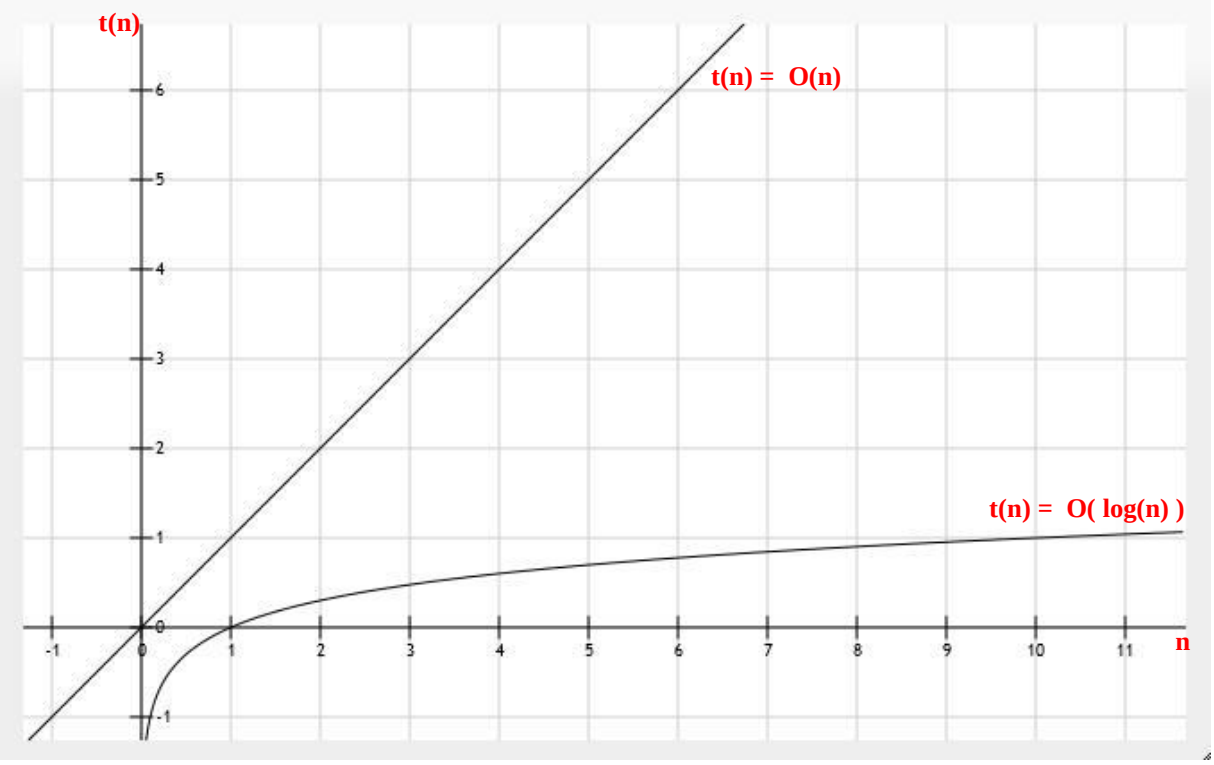
¿Qué significa el resultado obtenido? En otras palabras… ¿qué tan "mejor" es el $O(log(n))$ de la búsqueda binaria en comparación al $O(n)$ de la secuencial? La gráfica de ambas funciones (que se muestra más abajo) puede ayudar a comprender la respuesta. En ambos casos, si el número de

componentes aumenta, aumentará también el tiempo de ejecución. Pero si el tiempo es *O(n)* (es decir, la función *es lineal*) entonces a medida que *n* aumente, el *tiempo aumentará en la misma proporción*. Sin embargo, si el tiempo es *O(log(n))*, entonces a medida que *n* crezca el tiempo también aumentará, *pero lo hará de forma mucho más amortizada y suave*: grandes aumentos en el valor de *n*, provocarán pequeñas variaciones en el valor del tiempo medido.

Esto nos muestra que un *algoritmo de comportamiento logarítmico* es mucho más eficiente en cuanto al tiempo de ejecución que un algoritmo de comportamiento lineal, aunque debe comprenderse que cuando se analiza la eficiencia de un algoritmo en términos del aumento de *n*, lo que normalmente se busca es determinar ese comportamiento cuando *n* es *realmente grande* (es decir, lo que se conoce como el *comportamiento asintótico* de la función que mide el tiempo). Es en esa situación cuando la eficiencia compativa de un algoritmo toma plena vigencia: al fin y al cabo, si se trata de procesar pequeños lotes de datos (*n* igual a pocos cientos de elementos) entonces la mayor parte de los algoritmos serán veloces y la discusión carece de sentido…

Pero si el lote de datos es muy grande (*n* igual a varios miles, centenares de miles o millones de datos), entonces la acumulación de tiempo se vuelve progresivamente más pesada de manejar incluso para un computador… Allí es donde el buen programador será capaz de distinguirse del resto, seleccionando algoritmos adecuados para manejar el enorme volumen de datos que debe enfrentar, sin subestimar ese volumen. El *Desafío de Programación 01* planteado desde el aula virtual, apunta justamente en la dirección de mostrar al alumno en forma cruda el impacto de procesar lotes muy grandes con algoritmos poco eficientes… Vale la pena preguntarse: *¿se puede hacer mejor?*

Gráfica de las funciones O(n) y O(log(n)):



3.] SkipLists (Listas de Salto).

Volvemos a nuestra idea: mejorar el tiempo de ejecución de la inserción ordenada en una lista simple. El análisis de la búsqueda binaria nos mostró que podemos mejorar sustancialmente un algoritmo sin tener que modificar la estructura de datos subyacente (aunque eventualmente pueda requerirse un reordenamiento de sus componentes). Sin embargo, muchas veces la mejora sustancial requiere incorporar algún cambio en la *forma* de la estructura: un ejemplo simple es el caso del recorrido en ambos sentidos de una lista, que requiere que cada nodo incorpore un segundo puntero, y que la lista misma incluya un puntero al final de la misma. La estructura de datos sigue siendo esencialmente una lista, pero se incorporan a ella elementos de soporte que cambian de alguna forma su estructura interna, facilitando algunas tareas.

Esto muestra claramente la relación profunda que existe entre los algoritmos diseñados para resolver un problema y las estructuras de datos pensadas para soportar los datos de esos problemas. En el caso de la inserción ordenada en una lista (y consecuentemente la búsqueda veloz en esa lista) es fácil ver *que solo podremos lograr una mejora sustancial si encontramos la forma de ahorrar comparaciones en forma sistemática*. Idealmente, nos gustaría plantear para una lista una estrategia tan buena como la búsqueda binaria, con tiempo de ejecucíon *O(log(n))* tanto para la inserción, como para el borrado y la búsqueda. El problema de esta propuesta se ha comentado ya al inicio de esta Ficha de Estudio: las listas no proveen acceso directo a sus componentes, y ese pequeño detalle simplemente elimina la posibilidad de programar en *forma directa* la búsqueda binaria.

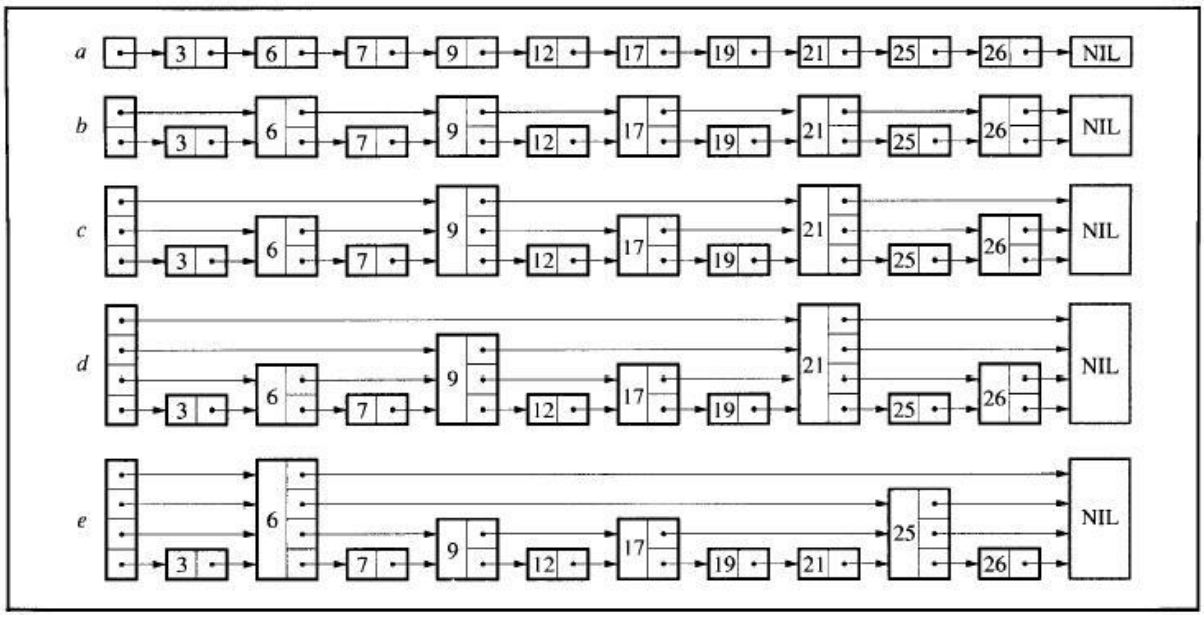
Una solución muy elegante para permitir resolver este problema con eficiencia es la *Lista de Saltos* o *SkipList*, que fue planteada en 1990 por *William Pugh* en su artículo *SkipLists: a probabilistic alternative to balanced trees.*

La idea detrás de las *SkipLists* puede asimilarse en forma progresiva: Si contamos con una lista simple con *n* nodos (supongamos ordenada de menor a mayor) y queremos buscar un valor (o insertar uno nuevo en forma ordenada), en el peor caso tendremos que recorrer la lista completa con tiempo *O(n)*. Si cada 2 nodos agregamos un puntero adicional que a su vez apunte 2 nodos más adelante (ver figura *b* en la gráfica siguiente), entonces una búsqueda requerirá *en el peor caso*, no más de *n/2 + 1* comparaciones (ver gráfico de la página siguiente).

Si continuamos con esta idea, podemos pensar en agregar otro puntero cada 4 nodos, que a su vez apunte 4 nodos más adelante (ver figura *c* en la gráfica anterior). Esto permitiría que una búsqueda, en el peor caso, insuma no más de *n/4 + 2* comparaciones. Compruebe el alumno visualmente esta afirmación, intentando determinar cuántas comparaciones hará para encontrar el valor 25 en la estructura de la figura *c*…

La idea general, entonces, es que si cada 2^i nodos agregamos un puntero extra que lleve a 2^i posiciones más adelante, entonces en el peor caso una búsqueda insumiría $n/2^i + i$ comparaciones. Intuitivamente, si duplicamos el número de punteros, lo que estamos haciendo divide sucesivamente por 2 a cada sublista obtenida de la partición anterior, por lo cual el tiempo de búsqueda resultante para el peor caso es $O(log_2(n))$ que ya vimos es equivalente simplemente a *O(log(n))*. De esta forma, la estructura de datos obtenida será efectivamente muy veloz para hacer búsquedas… pero la inserción y el borrado serán muy costosos, debido al elevado número de reajustes de punteros que deberán hacerse para mantenerla equilibrada cada vez que se inserte o elimine un valor…

Asimilación progresiva del concepto de SkipList[1]:



Para hacer que tanto las inserciones como las eliminaciones se mantengan en orden logarítmico, hay que formular algunos ajustes. Empecemos haciendo una definición: un nodo que dispone de *k* punteros hacia adelante, se designa como un *nodo de nivel k*. En general, un nodo de nivel *k* tendrá a uno de sus punteros apuntando al nodo que se encuentre 2^{k-1} lugares más adelante. Puede verse claramente que si cada 2^i*-ésimo* nodo tiene 2^i punteros hacia adelante, entonces los niveles de los nodos siguen un patrón de distribución sencillo: el 50% de los nodos son de nivel 1, el 25% son de nivel 2, el 12.5% son de nivel 3, y así sucesivamente (verifique intuitivamente esta afirmación con la figura *d* de la gráfica anterior). Hemos dicho que sería demasiado costoso en términos de tiempo insumido mantener y/o recuperar esta estructura de equilibrio, que es la que garantiza el orden logarítmico, cada vez que se inserta o elimina un nodo.

¿Qué hacer entonces? Se propone que al insertar un nuevo nodo, en lugar de forzarlo a entrar justo en el lugar y en el nivel que sería óptimo (muy costoso…) *se seleccione en forma aleatoria el nivel en el que se debe insertar, pero de forma tal que se mantengan las proporciones de reparto de nodos por nivel* que se mostraron en el párrafo anterior (vea la figura *e* en la gráfica anterior, y compárela con la figura *d*). Ahora la idea es que cada nodo de nivel *k*, en lugar de apuntar a un nodo ubicado 2^{k-1} lugares más adelante, *apunte al siguiente nodo del nivel k o mayor*. De esta forma, el nivel de un nodo elegido aleatoriamente no necesita cambiar de allí en adelante, y la inserción del nuevo nodo solo requeriría operaciones de actualización de punteros a nivel local (es decir, en el propio nodo y en sus adyacentes inmediatos) pero ya no en todos los nodos de la lista, lo cual reduce drásticamente el costo de la inserción.

El problema ahora es que de esta forma la proporción de nodos por nivel se mantiene, pero el equilibrio general de la estructura de la *SkipList* puede degradarse. En la idea original, cada nodo de nivel *k* está ubicado en un lugar que permite tener aproximadamente $n/2^k + k$ nodos a sus lados y entre otros nodos de nivel *k*, lo cual daba el orden logarítmico. Pero si ahora los niveles de los nodos se seleccionan en forma aleatoria, podría ocurrir que la *SkipList* termine teniendo una distribución de nodos muy desfavorable: suponga por ejemplo que todos los nodos de nivel más

[1] La gráfica que se muestra está tomada del paper original de W. Pugh, ya citado en esta misma ficha.

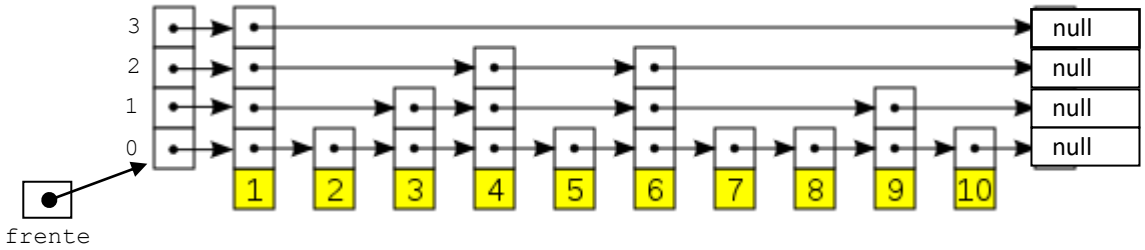
alto terminen estando juntos al principio de la lista… eso haría que se elimine la posibilidad de avanzar tramos grandes en la lista, volviendo a un recorrido casi de orden lineal. Sin embargo, puede probarse que esas distribuciones tan desfavorables son probabilísticamente raras (sobre todo si *n* es grande o muy grande), por lo cual puede esperarse que en la práctica las *SkipList* tendrán un rendimiento favorable.

4.] SkipLists: elementos de implementación. (Referencia: Proyecto TSBSkipLists)

Una *SkipList* es básicamente una lista ordenada común, a la cual se le anexan varias "capas" o "niveles" de punteros que forman a su vez otras listas. La lista de cada nivel contiene punteros para enlazar nodos más adelante, y de esta forma cada una de estas listas de nivel superior suele entenderse como una especie de "carril rápido" para avanzar con más velocidad en la lista subyacente (ver figura más abajo).

Cada nodo de una *SkipList* contiene un campo de datos (el consabido *info* que en la gráfica se muestra en color amarillo) y en lugar de contener un único puntero al nodo sucesor, cada nodo contiene un *arreglo de punteros* (llamaremos *next* a ese arreglo de punteros). En *todos* los nodos, el puntero contenido en la casilla *0(cero)* del arreglo *next* apunta al sucesor inmediato de ese nodo en la *SkipList* (siempre recuerde que una *SkipList* es una *lista ordenada*). Esto implica que si se quiere recorrer la lista en orden desde el principio y hasta el final (por ejemplo, en el método *toString()*), sin saltear ninguno, solo se debe recorrer la lista comenzando en el nodo inicial y tomando siempre la dirección almacenada en *next[0]*. Como siempre, el valor *null* en el casillero *next[k]* indica que ese nodo es el último de la lista. Vea la clase *Node* completa en el proyecto *SkipList* que acompaña a esta ficha.

Esquema de implementación de una SkipList[2]:



Para la implementación de la *SkipList* propiamente dicha, el puntero *frente* apunta a un nodo especial (que llamaremos *nodo de cabecera*), el cual contiene el ya citado arreglo de punteros. En el nodo de cabecera, el campo *info* se deja sin valor (si *info* es un puntero polimórfico, entonces en el nodo *frente* ese puntero se deja en *null*). Observe que el nodo *frente* solo se mantiene en la *SkipList* para referenciar al arreglo de punteros de entrada a todos los niveles de la estructura, *y por lo tanto no debe considerarse él mismo como un nodo más*. En otras palabras, la *SkipList* del gráfico anterior contiene 10 nodos (cuyos *info* están numerados de 1 a 10 en color amarillo). El nodo *frente* NO DEBE contarse como uno más en la estructura, ni tampoco se debe usar el *info* de ese nodo como válido (que de todos modos es siempre *null*). Vea la clase *SkipList* completa en el proyecto *SkipList* que acompaña a esta ficha.

Un extracto de implementación de la clase *Node* se muestra aquí, sin mayores comentarios:

[2] La gráfica que se muestra está tomada (y luego modificada) de *Wikipedia*: http://es.wikipedia.org/wiki/Skip_list.

```
package estructuras;

public class Node <E extends Comparable>
{
      private E info;
      private Node <E> [] next;

      public Node(int level, E data)
      {
        info = data;
        next = new Node[level + 1];
        for(int i = 0; i <= level; i++) { next[i] = null; }
      }

      public E getInfo()
      {
        return info;
      }

      public void setInfo(E x)
      {
        info = x;
      }

      public Node getNext(int i)
      {
        return next[i];
      }

      public void setNext(Node <E> x, int i)
      {
        next[i] = x;
      }

      @Override
      public String toString()
      {
         return info.toString();
      }
}
```

Y un extracto resumido de la clase *SkipList* se muestra aquí, solo a nivel de atributos, constructores y métodos sencillos:

```
public class SkipList<E extends Comparable>
{
      public static final int MAX_LEVEL = 18;

      private final Node<E> frente;
      private int cantidad;
      private int level;

      public SkipList()
      {
```

```
        this(MAX_LEVEL);
    }

    public SkipList(int ml)
    {
      frente = new Node<>(ml, null);
      cantidad = 0;
      level = 0;
    }

    public boolean isEmpty()
    {
       return (cantidad == 0);
    }

    public int size()
    {
        return cantidad;
    }

    // resto de la clase aquí…
}
```

La constante *MAX_LEVEL* se utiliza para establecer por default el número máximo de niveles que tendrá la lista. En el citado paper original de *W. Pugh* se prueba que el número óptimo de niveles para una *SkipList*, puede calcularse como $ml = log_2(N)$, siendo *N* una cota superior para la cantidad de datos *n* que se estima serán almacenados en la lista. Por ejemplo, puede verse que un valor *MAX_LEVEL = 18* permitiría procesar con eficiencia hasta 2^{18} valores, que es igual a 262144 datos… El primer constructor de la clase asume 18 niveles como máximo, pero el segundo constructor admite que ese número de niveles máximo se tome como parámetro.

El atributo *cantidad* se usa simplemente para llevar la cuenta de cuántos datos contiene la *SkipList* (se incrementa o decrementa cuando se inserta o se remueve un nodo). El valor de ese atributo indica si la *SkipList* está vacía (*cantidad == 0*) o no. El atributo *level* se usa para recordar en todo momento cuál es el máximo nivel al que se ha llegado hasta un momento dado, para que las búsquedas comiencen desde allí (y no necesariamente desde *MAX_LEVEL*, ya que en las primeras inserciones no se llega a cubrir tantos niveles…) Ambos constructores inicializan *level* en 0 y luego el método *add()* (que efectivamente inserta nuevos nodos) va actualizando ese valor para quedarse siempre con el mayor que se haya alcanzado.

Si la *SkipList* ya está armada, entonces el algoritmo de búsqueda de un valor *x* es simple: se comienza con la lista de más alto nivel (en el gráfico de más arriba, la lista de la casilla *3* en el nodo *frente*) y en esa lista se avanza hasta encontrar a *x* (en cuyo caso se detiene la búsqueda) o hasta encontrar el primer nodo cuyo *info* sea mayor a *x*, en cuyo caso se retrocede hasta el nodo anterior en ese nivel, se desciende al puntero de la capa o nivel inmediato más abajo, y se prosigue la búsqueda desde allí, repitiendo el esquema hasta dar con el nivel 0. Hemos visto que si la lista se mantiene aceptablemente equilibrada en cuanto a la distribución de nodos en cada nivel, este algoritmo tendrá un tiempo de ejecución de *O(log(n))*. En la clase *SkipList*, el método *search()* toma como parámetro el objeto *data* a buscar y lo busca en base a las ideas antes expuestas. Si no lo encuentra o *data* no es compatible con el contenido de la *SkipList*, retorna *null*. Si lo encuentra, retorna una referencia al *objeto que se encontró en la lista*:

```
public E search(E data)
{
    Node <E> x = frente;
    for(int i = level; i >= 0; i--)
    {
        while(x.getNext(i) != null && x.getNext(i).getInfo().compareTo(data) < 0) { x = x.getNext(i); }
    }
    x = x.getNext(0);
    if(x != null && x.getInfo().compareTo(data) == 0) { return x.getInfo(); }
    return null;
}
```

La inserción de un nuevo elemento *x* en la lista requiere primero una búsqueda para determinar en qué lugar debe ir el nuevo nodo. El método privado *randomLevel()* de la clase *SkipList* se usa para calcular en forma aleatoria el número de nivel en el cual deberá agregarse *x*. Cuando la búsqueda termine y estemos listos para enlazar el nuevo nodo en el nivel que le haya tocado, se usa un vector auxiliar llamado *update* de forma que cada casilla *update[i]* contenga un puntero al nodo de *nivel igual o mayor que i*, que se encuentre a la derecha del nodo que será el anterior a *x* una vez enlazado (en otras palabras, *update* permite actualizar los enlaces de todos los nodos entre los cuales se insertará *x*). Si la inserción de *x* se hace en un nivel mayor al que hasta ese momento era el más alto en la lista, se actualiza el valor del atributo *level* para que pase a valer ese nuevo nivel mayor. En una eliminación de nodo, se procede en forma similar pero se chequea si el nodo removido era el último del mayor nivel y en ese caso se disminuye el valor de *level*. Los métodos *add()* y *remove()* de la clase *SkipList* aplican estas ideas, y se deja para el alumno el análisis detallado de sus funcionamientos:

```
private int randomLevel()
{
        float p = 0.5f;
        int newLevel = 0;
        while(Math.random() < p) { newLevel++; }
        if(newLevel < num_levels) { return newLevel; }
        return num_levels;
}

public void add(E data)
{
            int i, newLevel;
            Node <E> x = frente;
            Node <E> update[] = new Node[num_levels + 1];
            for(i = level; i >= 0; i--)
            {
                    while(x.getNext(i) != null && x.getNext(i).getInfo().compareTo(data) < 0) { x = x.getNext(i); }
                    update[i] = x;
            }

            x = x.getNext(0);
            if(x != null && x.getInfo().compareTo(data) == 0) { x.setInfo(data); }
            else
            {
                    newLevel = randomLevel();
                    if(newLevel > level)
                    {
                        for(i = level + 1; i <= newLevel; i++) { update[i] = frente; }
```

```
                    level = newLevel;
                }
                x = new Node <>(newLevel, data);
                for( i = 0; i <= newLevel; i++)
                {
                    x.setNext(update[i].getNext(i), i);
                    update[i].setNext(x, i);
                }
                cantidad++;
            }
}

public void remove(E data)
{
            int i;
            Node <E> x = frente;
            Node <E> update[] = new Node[num_levels];
            for(i = level; i >= 0; i--)
            {
                while(x.getNext(i) != null && x.getNext(i).getInfo().compareTo(data) < 0) { x = x.getNext(i); }
                update[i] = x;
            }

            x = x.getNext(0);
            if(x != null && x.getInfo().compareTo(data) == 0)
            {
                for(i = 0; i <= level; i++)
                {
                    if(update[i].getNext(i) != x) { break; }
                    update[i].setNext(x.getNext(i), i);
                }
                x = null;
                while(level > 0 && frente.getNext(level) == null) { level--; }
                cantidad--;
            }
}
```