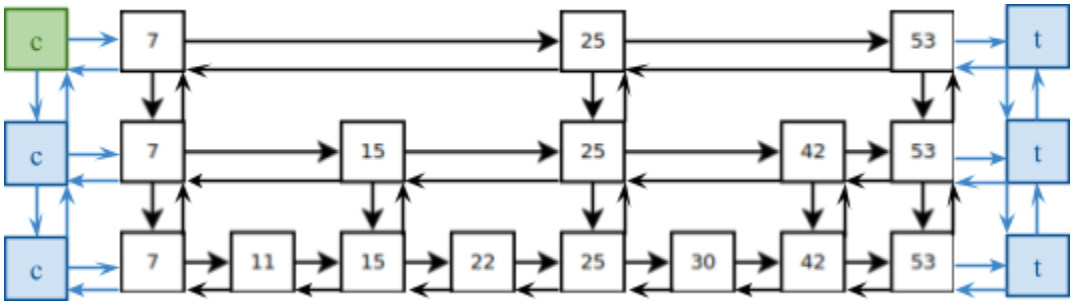
Murillo García Sebastián 191838

Desempeño de la estructura Skip List.

¿Cómo cambia el desempeño de un Skip List dependiendo del número de datos y del orden de las operaciones efectuadas? Para responder a esta pregunta es necesario recordar los puntos más importantes de esta estructura de datos. La idea de un Skip List es tener una lista base con ***n*** elementos y tener ***m*** listas ligadas entre ellas que cada vez tengan menos elementos en su interior para recorrer de una manera más rápida la lista al "saltarse" elementos. La manera ideal para generar saltos eficientes de los elementos es generar listas auxiliares que tengan la mitad de los elementos de la lista anterior. Entonces la pregunta del millón se presenta: ¿cuántas listas máximo podemos tener en una estructura con ***n*** datos? La respuesta es igual a muchas de las respondidas en durante el curso: ***log(n)***. ¿Por qué? Porque idealmente en 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. Ahora bien, ¿cómo se ve el Skip List ideal? Aquí tenemos un ejemplo:



Si nos damos cuenta (por lo dicho en clase y por lo dicho anteriormente), en el **mejor de los casos**, al ir insertando una cantidad n de elementos, la estructura va a tener un desempeño de ***O(log(n))*** [más o menos] porque con pocas preguntas podemos llegar al resultado deseado en las operaciones de búsqueda, de inserción y de borrado. ¿En qué depende que se cumpla este desempeño? De primera mano podría decirse que depende del acomodo de los elementos en las diferentes listas auxiliares de la estructura. Aquí nos encontramos con el "problema" o "inconveniente" de un Skip List: es una estructura randomizada. A pesar de que su resultado es determinista lo que es probabilístico es su desempeño. En el método de inserción no podemos controlar de manera determinista cuáles elementos suben a las listas superiores y cuáles no. De esta manera, no podemos tener el control de cuántos elementos se van a tener en cada lista y cómo va a ser su acomodo. Es así como la estructura ideal puede cumplir en el **peor de los casos** con un terrible ***O(n)*** en su desempeño.

Se gana mucho al no tener que utilizar métodos de balanceo o algo parecido, pues esto hace que sea una estructura más "ligera" que otras, 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. En esta tarea veremos cómo cambia el desempeño de esta estructura dependiendo del número de datos y del orden de operaciones. Si tuviéramos que hacer una hipótesis en estos momentos, diríamos que el desempeño en general va a ser bueno si y solo si el número de elementos en las listas auxiliares están alrededor del ideal y si el acomodo de los mismos facilita el acceso veloz a los demás elementos en el Skip List.

Diseño del experimento.

Para empezar, se va a verificar que la estructura cumpla la condición de que el número de listas se mantenga alrededor de $log_2(n)$ en la creación de ellas al insertar una cantidad n de datos y en el colapso de las mismas al borrar esos mismos n datos de la estructura. ¿Por qué vamos a verificar esto? Para estar seguros de que al insertar pueda existir la posibilidad de tener en cada lista la mitad de los elementos de la inferior; es decir, si se cumple la condición, entonces está presente la base para el acceso veloz a los datos y la pseudo-aleatoriedad de tirar el volado nos puede favorecer.
Ahora bien, después se va a exponer y analizar un caso en particular en el cual el desempeño es bueno y otro más en el cual es malo después de insertar una cantidad n de datos. Para estos dos casos se va a dejar que el valor de probabilidad de que un elemento insertado suba a una lista sea de 0.5 y no se implementará ninguna semilla. Después se va a efectuar una operación de borrado consciente sobre la estructura y se va a evidenciar el cambio en el desempeño. El borrado consciente se va a efectuar para mostrar cómo es que cambia el desempeño después de efectuar esta operación.
Seguido de esto, se van a mostrar algunos casos "forzados". Por forzados nos referimos a que se va a modificar el código base o que se va a implementar algún método para mostrar situaciones en las cuales el desempeño puede ser excelente o cuando puede ser terriblemente malo. Esto se hará solamente para analizar algunos casos "de esquina" **parecidos** (repitamos: parecidos, aunque también se podría "meter mano al código" para llegar al mejor de los casos) en los cuales nos podemos encontrar (ya sea por una suerte bastante buena o por una demasiado mala) al implementar esta estructura.

Para lograr el objetivo propuesto con el diseño de los experimentos antes descritos, nos vamos a valer de los siguientes métodos programados en Java.

Método 1: obtención del número de elementos por lista.

El **objetivo** de este método es conocer la distribución en cantidad de los elementos en las diferentes listas auxiliares de la estructura. De esta manera, podemos comparar la cantidad real de elementos existentes en cada lista auxiliar con la cantidad ideal de los mismos en ellas y podremos acercarnos más para conocer cómo cambia el desempeño de la estructura después de diferentes operaciones de inserción y de borrado. Si bien este método puede darnos más información sobre el cambio en el desempeño de la estructura, un dato sobre cantidad no nos brinda toda la información necesaria. Más bien, necesitamos un dato relacionado a la cualidad de las listas auxiliares, es decir, información sobre cuáles elementos se encuentran en las diferentes listas. De cualquier modo, este método es un gran apoyo para el objetivo de esta tarea.

El método **funciona** de la siguiente manera: por medio de la cabeza se llega a la lista más inferior de la estructura con la variable 'c', mientras esta no sea nula, se irá subiendo por el centinela de cabeza de las diferentes listas. Ahora bien, la variable 'actual' va a recorrer las listas, por lo tanto empieza con el elemento que se encuentra a la derecha del centinela de cabeza y mientras no sea nula, el contador aumenta y 'actual' se mueve a la derecha.

```
public ArrayList<Integer> numElemXList(){
        ArrayList<Integer> res = new ArrayList();
        SkipNode<T> c, actual;
        int cont;

        // Se llega a la lista inferior.
        c = cabeza;
        while(c.getAb() != null){
            c = c.getAb();
        }

        // Se hace el conteo lista por lista.
        while(c != null){
            actual = c.getDer();
            cont = 0;
            while(actual.getElem() != null){
                cont++;
                actual = actual.getDer();
            }
            res.add(cont);
            c = c.getArr();
        }
        return res;
    }
```

Método 2: obtención del primer pilar de la estructura.

El **objetivo** de este método es la búsqueda del primer elemento pilar para posteriormente borrarlo de la estructura. ¿A qué nos referimos con "primer elemento pilar"? Al primer elemento que se encuentre en la lista más superficial o superior de la estructura. Si nos damos cuenta, al borrar a uno de estos elementos, se disminuyen las posibilidades de acceso veloz para las operaciones de búsqueda, de inserción y de borrado. El método se pensó para que se pudieran eliminar distintos pilares de la estructura y se lograra analizar un caso más en el cual el desempeño de la estructura disminuye. En el peor de los casos, si no se tienen suficientes elementos en las listas auxiliares (que van formando los pilares del Skip List en general), se tendrá que recorrer la lista base cada vez más con sus *n* elementos.

El método **funciona** de la siguiente manera: se empieza a recorrer la estructura por medio de la cabeza y se tiene una bandera de fin de datos junto con un contador para bajar en las listas. Mientras que la bandera siga estando en falso y el contador siga siendo menor que el número de listas existentes, 'actual' toma el primer valor que se encuentra a la derecha de la cabeza en la cual nos encontramos. Además, mientras que la bandera no sea falsa y 'actual' no sea nulo, se va a ir moviendo a su siguiente nodo en la lista. Si el elemento que guarda 'actual' es diferente de nulo, entonces el resultado es ese nodo en el que nos encontramos y la bandera de fin de datos se cambia a true.

```
public T pilar(){
        SkipNode<T> actual, c, res;
        boolean var = false;
        int i;

        res = new SkipNode(null);
        actual = cabeza;
        c = actual;
        i = 0;
        while(!var && i < numListas){
            actual = actual.getDer();
            while(!var && actual != null){
                if(actual.getElem() != null){
                    res = actual;
                    var = true;
                }
                actual = actual.getDer();
            }
            actual = c.getAb();
            c = c.getAb();
            i++;
        }
        return res.getElem();
    }
```

Método 3: reestructuración de la estructura de manera determinista.

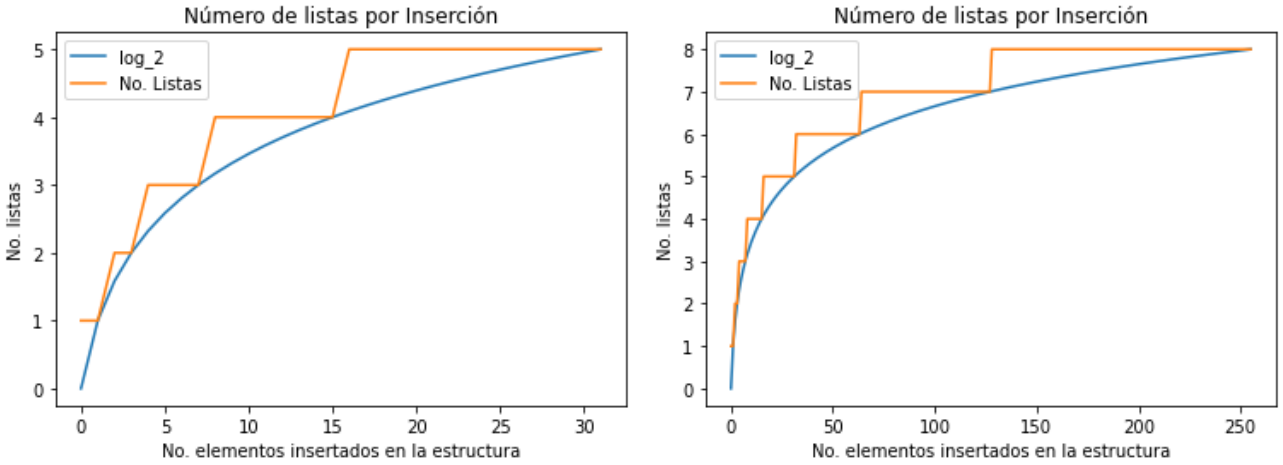
El **objetivo** de este método es el "acomodo" de la estructura en su versión más eficiente, de una manera determinista. ¿A qué nos referimos "en su versión más eficiente"? Tomando en cuenta a ***n*** como la cantidad de elementos en la estructura, $\boldsymbol{log_2(n)}$ el número de listas de la estructura y la **lista 1** la lista base de la estructura, la versión más eficiente es cuando cada lista auxiliar tiene la mitad de elementos de su lista inferior. En otras palabras, la lista 1 tiene n elementos, la lista 2 tiene n/2 elementos, la lista 3 tiene n/4 elementos y así sucesivamente. ¿Además, cómo determinamos qué mitad de elementos sube? Para asemejar un poco la estructura de un árbol binario, se determinó en clase que una buena manera para reestructurar es que un elemento suba y otro no. De esta manera, el recorrido para buscar en las operaciones de inserción o borrado será el más eficiente.

```
public void reestructura(){
        SkipNode<T> actual,cab,c,r, tail;
        double n;
        int i;

        n = (Math.log(cont)/Math.log(2))+1;
        while(numListas < n){
            cab = cabeza;
            c = new SkipNode(cab.getElem());
            ligaArrAb(c,cab);
            cabeza = c;
            r = c;
            actual = cab.getDer();
            i = 1;
            while(actual.getElem() != null){
                if(i%2 == 0){
                    SkipNode<T> a = new SkipNode(actual.getElem());
                    ligaArrAb(a,actual);
                    ligaIzqDer(r,a);
                    r = a;
                }
                actual = actual.getDer();
                i++;
            }
            tail = cola;
            SkipNode<T> t = new SkipNode(tail.getElem());
            ligaArrAb(t,tail);
            ligaIzqDer(r,t);
            cola = t;
            numListas++;
        }
    }
```

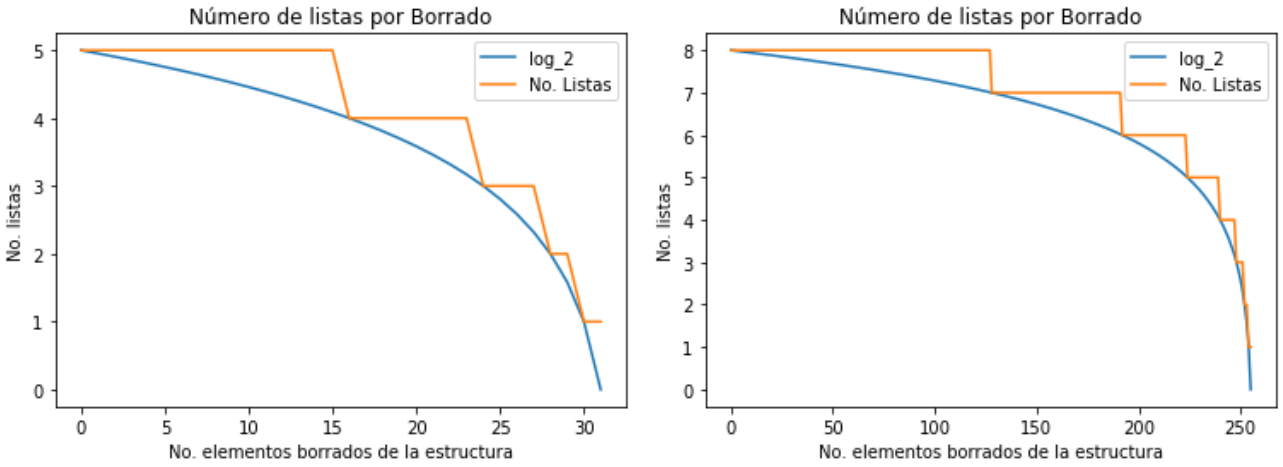
Resultados.

I. Gráficas sobre la verificación del número de listas por cantidad de datos insertados.



Como podemos observar, si se insertan *n* datos en la estructura (en estos casos se insertaron 32 y 256 elementos), el número de listas sí toca en las potencias de 2 a la función $\boldsymbol{log_2(n)}$; por lo tanto, se cumple la condición adecuada para que los elementos puedan distribuirse en las diferentes listas auxiliares y el desempeño de la estructura en sus diferentes operaciones pueda acercarse a $\boldsymbol{O(log(n))}$.

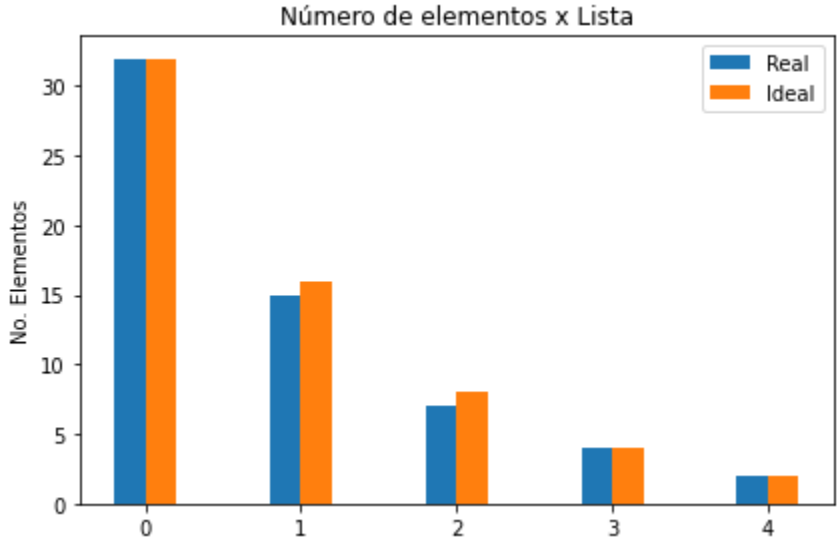
II. Gráficas sobre la verificación del número de listas por cantidad de datos borrados.



Por otro lado, aquí podemos observar que se cumple el colapso de la cantidad de listas cuando estas ya no son necesarias en la estructura. De esta manera, podemos estar seguros de que siempre se tendrán las listas necesarias dada la cantidad de elementos en el Skip List y nuevamente se mantendrá la base para que el desempeño ronde alrededor de $\boldsymbol{O(log(n))}$.

Ahora bien, desde aquí hay que dejar en claro que no porque esta base esencial en el número de listas se cumpla quiere decir que la estructura va a ser lo más eficiente. Como se dijo anteriormente, también tiene mucho que ver la distribución de los elementos en la misma y eso se probará en las siguientes gráficas. De todas maneras se hace la prueba correspondiente a esta característica para evidenciar que la estructura brinda la posibilidad de que el desempeño sea de ***O(log(n))***.

III. Gráfica sobre el caso particular en el cual el desempeño es bueno. [con n = 32 datos]



- Número de elementos por Lista Real

[**L1:** 32 **L2:** 15 **L3:** 7 **L4:** 4 **L5:** 2]

- Número de elementos por Lista Ideal

[**L1:** 32 **L2:** 16 **L3:** 8 **L4:** 4 **L5:** 2]

- Estructura representada en la gráfica:

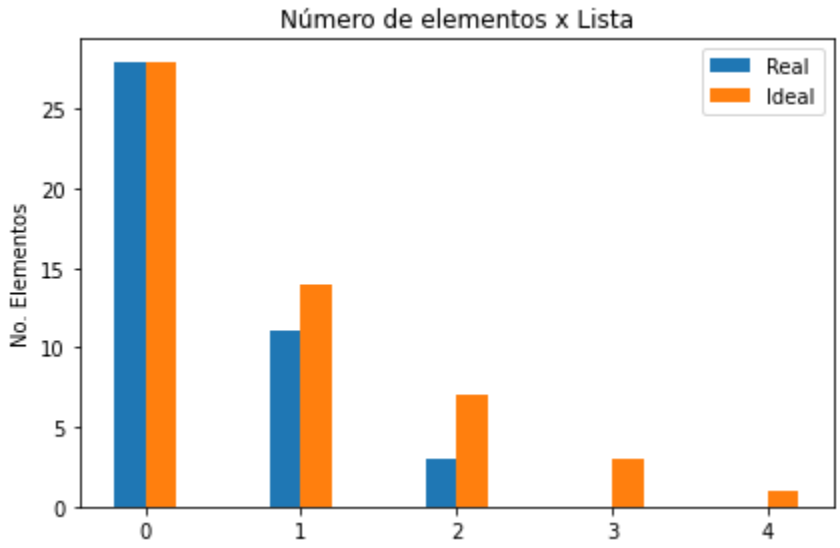
[2, 22]
[2, 5, 14, 22]
[2, 5, 6, 14, 22, 23, 25]
[0, 2, 5, 6, 8, 10, 12, 14, 17, 22, 23, 25, 27, 28, 30]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

Podemos decir que esta estructura es relativamente buena en cuanto a su desempeño debido a dos diferentes factores. Para empezar, como la gráfica y los datos lo demuestran, la cantidad de elementos esperados en cada una de sus listas es muy cercana a la cantidad ideal, simplemente les falta 1 elemento a la lista 2 y uno a la 3 para cumplir con el número ideal.

Como segundo factor, y el más importante, la estructura es eficiente y se acerca a un desempeño de ***O(log(n))*** debido al "acomodo" de los elementos en las listas internas. ¿Por qué podemos afirmar eso? Si bien los dos elementos más superficiales [2, 22] no están exactamente a la mitad de la estructura, con solo dos preguntas podemos saber en qué rango de colocación se encuentra un elemento que se desea buscar, insertar o borrar: entre [0,1] , [3,21] o [23, 31]. Conforme vamos descendiendo en las listas de la estructura, esta función sigue siendo eficiente en su mayor parte, puesto que sí contempla buenos rangos en los cuales se pueden hacer preguntas rápidas y descartar "secciones" de la estructura para tomar acción. Claro que podemos encontrar algunos puntos en los cuales la estructura no es completamente eficiente, como por ejemplo si se desea eliminar el elemento 20; ¿por qué no es completamente eficiente? Porque en dos ocasiones tiene que moverse entre 14 y 22 en la tercera y cuarta lista. Estos detalles son mínimos en la estructura, por lo que no debe de afectar tanto en el desempeño general de la estructura ya que se ve inclinada más a ser eficiente a que cause problemas graves en el momento de realizar operaciones.

Ahora bien ¿qué pasaría si se borran los cuatro pilares más representativos de la estructura?, ¿afectará al desempeño? La respuesta es un rotundo sí. Veamos la siguiente gráfica, los datos y la estructura si se eliminan a los 4 pilares principales: 2, 22, 5 y 14.

IV. Gráfica sobre el caso particular después de haber sido borrados sus pilares



- Número de elementos por Lista Real

[**L1:** 28 **L2:** 11 **L3:** 3 **L4:** 0 **L5:** 0]

- Número de elementos por Lista Ideal

[**L1:** 28 **L2:** 14 **L3:** 7 **L4:** 3 **L5:** 1]

- Estructura representada en la gráfica:

[ ]

[ ]

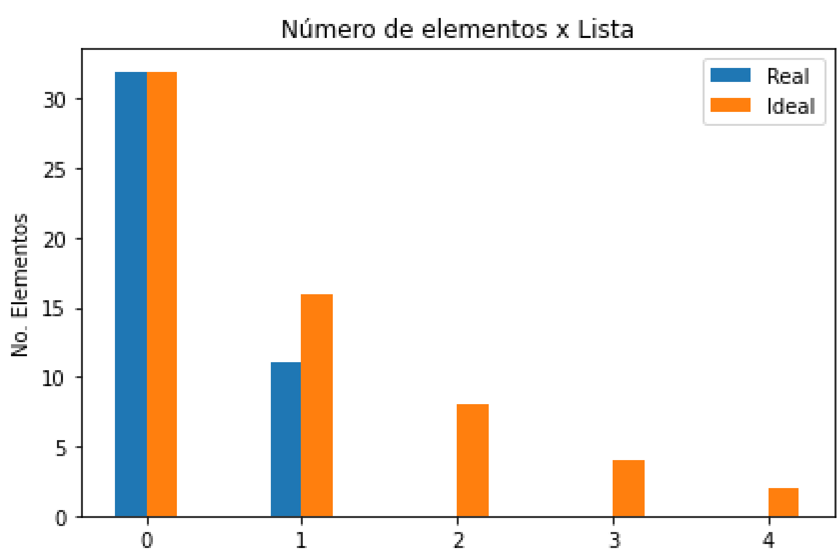
[6, 23, 25]

[0, 6, 8, 10, 12, 17, 23, 25, 27, 28, 30]

[0, 1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31]

¿Qué fue lo que sucedió? En primer lugar, como la gráfica y los datos lo demuestran, se están desperdiciando dos listas que aún están presentes en la estructura pero que ya no contienen ningún elemento en su interior. Esto disminuye la efectividad del desempeño porque ahora se pueden hacer menos preguntas a la estructura para conocer el rango de movilidad en el cual puede operar. ¿Qué provoca esto? Recordemos que la razón de ser de esta estructura es que se puedan "saltar" elementos para conocer la ubicación veloz de un elemento en la estructura al hacer preguntas y descartar datos para seguir avanzando en los saltos. Al eliminar los pilares representativos de la estructura y no aprovechar estas dos listas (que ahora resultan sobrantes) lo que sucede es que el desempeño se aleja de ser ***O(log(n))*** por la cantidad de preguntas que ahora se tendrán que hacer para el acceso veloz a los datos. Por lo tanto, ahora se está acercando un poco más a un desempeño de ***O(n)***. Claro, debe decirse que no es tan drástico este cambio de un desempeño bueno a uno completamente malo porque todavía siguen existiendo preguntas efectivas que se pueden hacer para encontrar al elemento, pero sí es necesario mencionar que con estas operaciones poco a poco la estructura se aleja de un buen desempeño.

V. *Gráfica sobre el caso particular en el cual el desempeño es malo*. [con n = 32 datos]



- Número de elementos por Lista Real

[**L1:** 32 **L2:** 11 **L3:** 0 **L4:** 0 **L5:** 0]

- Número de elementos por Lista Ideal

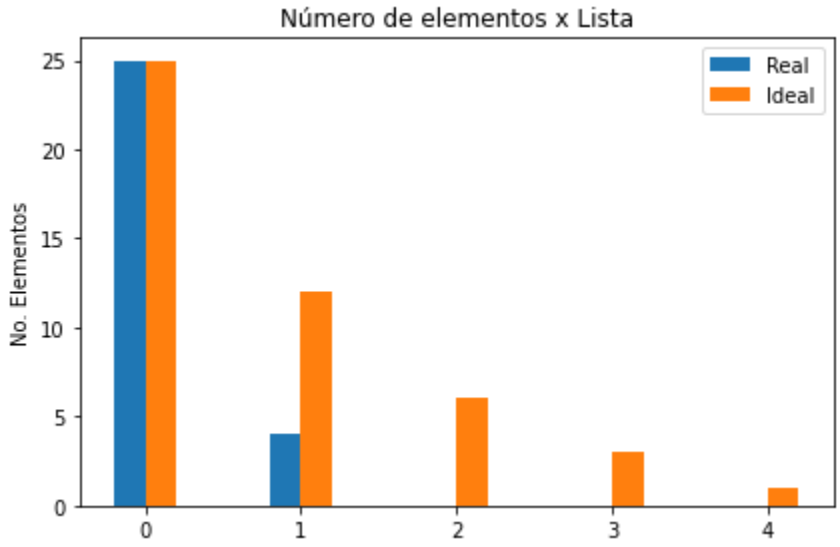
[**L1:** 32 **L2:** 16 **L3:** 8 **L4:** 4 **L5:** 2]

- Estructura representada en la gráfica:

[ ]
[ ]
[ ]
[1, 6, 7, 9, 16, 17, 18, 21, 26, 27, 30]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

¿Qué es lo que estamos viendo aquí? Para empezar, podemos observar que 3 listas están siendo completamente desaprovechadas al no tener ni un elemento en su interior. Como se dijo anteriormente, al tener pocos elementos en las listas auxiliares se disminuye la cantidad de preguntas que se pueden hacer a la estructura para acceder a los elementos. En este ejemplo ya es más claro cuál es el problema: la estructura parece acercarse más a una simple lista ligada. Si bien todavía existen elementos en la segunda lista que pueden "acotar" el acceso, estos no resultan ser los más efectivos debido a que se encuentran demasiado juntos (por ejemplo el caso de los elementos 16, 17 y 18). Por lo tanto, esta estructura en este caso se encuentra más cerca de un desempeño de ***O(n)*** que de ***O(log(n))***. Al igual que el caso anterior, ¿qué pasará si se borran ahora 7 elementos pilares de la estructura?

VI. Gráfica sobre el caso particular después de haber sido borrados sus pilares.



- Número de elementos por Lista Real

[**L1:** 25 **L2:** 4 **L3:** 0 **L4:** 0 **L5:** 0]

- Número de elementos por Lista Ideal

[**L1:** 25 **L2:** 12 **L3:** 6 **L4:** 3 **L5:** 1]

- Estructura representada en la gráfica:

[ ]
[ ]
[ ]
[21, 26, 27, 30]
[0, 2, 3, 4, 5, 8, 10, 11, 12, 13, 14, 15, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

Aquí podemos ver más claramente que al ir borrando elementos importantes de la estructura, el desempeño se va deteriorando hasta el punto de llegar casi al terrible ***O(n)***. De 25 elementos, al menos los primeros 14 van a tener que accederse uno por uno para encontrar algún dato. Si bien la otra mitad todavía tiene elementos para poder descartar algunos rangos después de las preguntas obligadas, no podemos no obviar que una mitad es completamente ineficiente en la estructura. Ahora solo hay que imaginarnos cuál sería el desempeño si se eliminaran los elementos 21, 26, 27 y 30. ¿Qué es lo que tenemos? Una simple lista ligada.

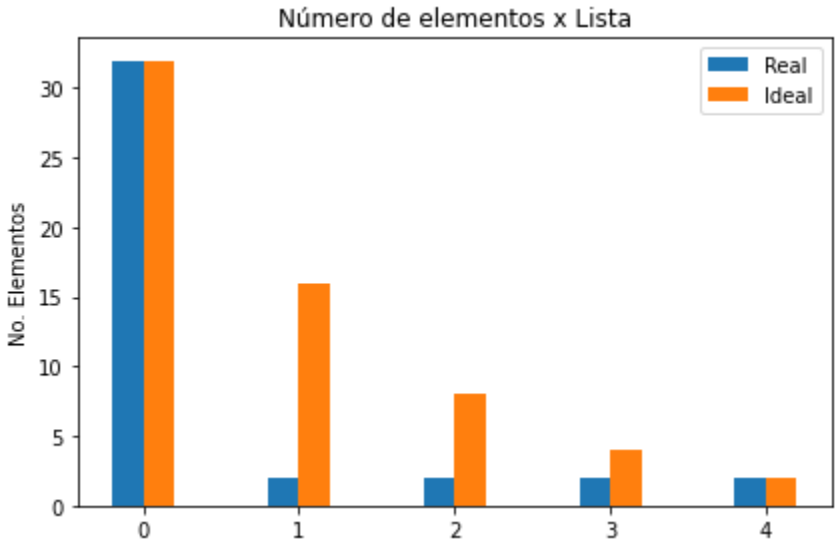
Ahora bien, aquí es importante mencionar algo: la cantidad de listas auxiliares es completamente independiente de que si hay datos en ellas o no. Estas listas se crean dependiendo de la cantidad de elementos en la lista base, así que es evidente que sí podemos tener la mala suerte de que al tirar los volados para insertar un elemento en la lista superior nos toque un caso parecido a este y la estructura sea más ineficiente que eficiente.

Algunos casos forzados.

Después de presentar estos casos en donde podemos ver un desempeño bastante bueno para la estructura y uno en el cual es completamente malo, se van a presentar rápidamente tres casos forzados: uno en el cual un Skip List puede tener un desempeño excelente y dos en los cuales un Skip List resulta ser ineficiente. Estos casos los presentamos como “casos esquina”, es decir, los casos más cercanos a los extremos en los cuales nos podemos encontrar.

Entonces, ¿primero las malas o la buena?

VII. Gráfica sobre el caso forzado en el cual la probabilidad para la inserción en una lista superior es de 0.1



- Número de elementos por Lista Real

[**L1:** 32 **L2:** 2 **L3:** 2 **L4:** 2 **L5:** 2]

- Número de elementos por Lista Ideal

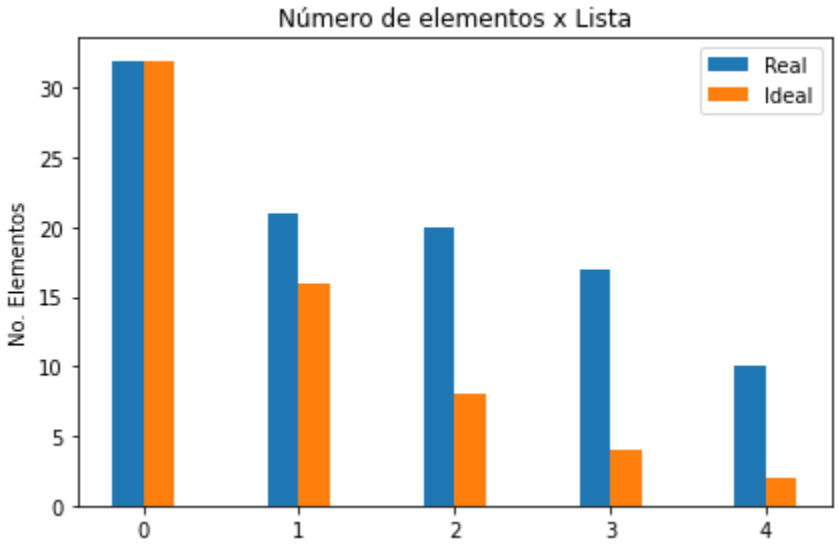
[**L1:** 32 **L2:** 16 **L3:** 8 **L4:** 4 **L5:** 2]

- Estructura representada en la gráfica:

[4, 15]
[4, 15]
[4, 15]
[4, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

Si bien en todas las listas hay elementos, en este caso sería exactamente lo mismo tener dos listas que tener cinco, pues en 4 de ellas tenemos a los mismos dos elementos y eso retrasa la operación. ¿Por qué se retrasa la operación? Porque el método de búsqueda pregunta 4 veces la misma pregunta si es que queremos encontrar a un elemento en cualquiera de los tres rangos que delimitan los pilares de la estructura. Por lo tanto, aquí nos encontramos en un caso con un desempeño mucho más cercano al de ***O(n)***.

VIII. Gráfica sobre el caso forzado en el cual la probabilidad para la inserción en una lista superior es de 0.8



- Número de elementos por Lista Real

[**L1:** 32 **L2:** 21 **L3:** 20 **L4:** 17 **L5:** 10]

- Número de elementos por Lista Ideal

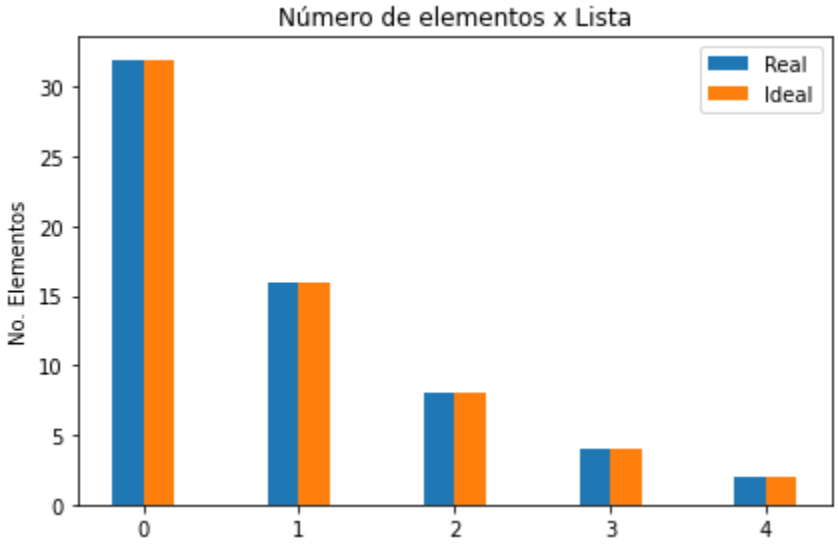
[**L1:** 32 **L2:** 16 **L3:** 8 **L4:** 4 **L5:** 2]

- Estructura representada en la gráfica:

[1, 6, 8, 10, 13, 16, 17, 20, 24, 25]
[0, 1, 6, 7, 8, 10, 11, 13, 14, 16, 17, 20, 21, 24, 25, 29, 31]
[0, 1, 2, 6, 7, 8, 9, 10, 11, 13, 14, 16, 17, 20, 21, 22, 24, 25, 29, 31]
[0, 1, 2, 6, 7, 8, 9, 10, 11, 13, 14, 16, 17, 20, 21, 22, 24, 25, 28, 29, 31]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

Aquí tenemos el otro caso forzado, en el cual existe un exceso de elementos en las distintas listas. Esto hace que sea ineficiente la estructura y que el desempeño igual se acerque a ***O(n)*** debido a que las listas auxiliares se parecen bastante a la lista base y recordemos que la lista base es una lista ligada con *n* elementos. el problema entonces es que no se está "saltando efectivamente" a los elementos en la lista. Incluso podemos observar que la segunda y la tercera lista solamente difieren en 1 en la cantidad de sus elementos. Este problema entonces se presenta cuando hay un exceso en la cantidad de elementos en las listas, además de que la distribución claramente no va a ser muy favorable para el Skip List.

IX. El caso ideal.



- Número de elementos por Lista Real

[**L1:** 32 **L2:** 16 **L3:** 8 **L4:** 4 **L5:** 2]

- Número de elementos por Lista Ideal

[**L1:** 32 **L2:** 16 **L3:** 8 **L4:** 4 **L5:** 2]

- Estructura representada en la gráfica:

[15, 31]
[7, 15, 23, 31]
[3, 7, 11, 15, 19, 23, 27, 31]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

Esta es la estructura de Skip List ideal debido a dos factores importantes. En primer lugar, se cumple que cada lista auxiliar contiene exactamente la mitad de los elementos de su lista inferior; en otras palabras, se cumple la cantidad de datos ideales en cada lista para la estructura. En segundo lugar, y lo más importante, la distribución de los datos en las listas es perfecta en el sentido en que al hacer las preguntas para el acceso a los datos, se van eliminando exactamente la mitad de los datos. ¿Esto a qué se parece? A un árbol binario de búsqueda. ¿Y cuál es el desempeño de esta estructura? En este caso, el desempeño es de un perfecto ***O(log(n))***. Bello, ¿no? Esto se logró gracias al método de reestructura, que es una buena práctica para ocupar cuando se tiene que recorrer la lista base obligatoriamente (por ejemplo en un toString). Este se presenta como **el mejor caso** para un Skip List.

Conclusiones.

El desempeño de un Skip List en general suele ser de ***O(log(n))*** y eso es bastante bueno. Claro, al no tener el control de la estructura como tal ya que es una estructura randomizada, esto nos brinda ventajas debido a que no es necesario hacer algún tipo de balanceo para mantener un desempeño bueno. El problema con el que nos encontramos es aquel relacionado con las operaciones de borrado. Si tenemos la mala suerte de querer borrar a un elemento importante de la estructura como lo es un pilar, el desempeño se va a ir disminuyendo poco a poco. Parece ser que la complejidad está en una constante recta en la que en los extremos nos encontramos con ***O(log(n))*** y con ***O(n)*** como las complejidades. Este movimiento entre ambos extremos se puede suplir al insertar nuevos elementos en la estructura, ya que tendremos la posibilidad de tener la suerte de que el elemento suba a las listas auxiliares y se establezca como un pilar importante, aunque también tenemos la posibilidad de llenar nuestras listas de elementos que solo van a empeorar el desempeño. De todas maneras nos encontramos con la bella opción de reestructurar deterministicamente el Skip List cuando hacemos un llamado al método de toString, por ejemplo. Esto hace que la estructura se mantenga cerca del mejor de los casos.

La Skip List es una buena estructura de datos que mantiene su desempeño alrededor de ***O(log(n))***, pero hay que tener en cuenta siempre lo que se dijo al principio de la tarea: es una estructura randomizada y lo que es probabilístico es su desempeño.