DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Tarea 3: Skip Lists y ABB Aleatorizados

CC4102 - Diseño y Análisis de Algoritmos

Profesor:	Gonzalo Navarro
Auxiliar:	Teresa Bracamonte
Alumnos:	Cristián Carreño
	Sergio Maass

Fecha:	18 de Diciembre de 2013

Índice

1. Introducción

Los Skip Lists (SL) son estructuras de datos que permiten almacenar objetos en listas enlazadas ordenadas. A diferencia de las listas enlazadas convencionales, los skip lists aleatorizan el algoritmo de inserción de forma que cada vez que se inserta un nuevo elemento, se pueden generar d duplicados aleatoriamente de forma que estos compongan un atajo para la lista enlazada original y reduzcan el tiempo de búsqueda. La ventaja de los skip list es que aseguran un tiempo promedio de búsqueda de $O(\log n)$, al igual que los árboles de búsqueda binarios.

Los Árboles de Búsqueda Binarios (ABB) aseguran que, en promedio, los tiempo de inserción y búsqueda sean de $O(\log n)$. Sin embargo, en el peor caso podemos terminar con un árbol de altura $O(n)$. Para reducir ese tipo de situaciones, Martinez y Roura proponen aleatorizar las operaciones de inserción y eliminación en el ABB. Específicamente, en la inserción se considera que cada elemento tiene la probabilidad $\frac{1}{n}$ de ser la raíz del árbol; siendo n el número de elementos actualmente en el árbol.

En el presente trabajo se pretende determinar cuál de estas estructuras es mejor en la práctica, para lo cual se implementarán y luego serán sometidas a prueba.

2. Implementación

A continuación se detalla la implementación de las estructuras que serán comparadas: Skip List (SL), Árbol Binario de Búsqueda (ABB), y ABB Aleatorizado. Todas se implementaron en Python.

2.1. Skip List

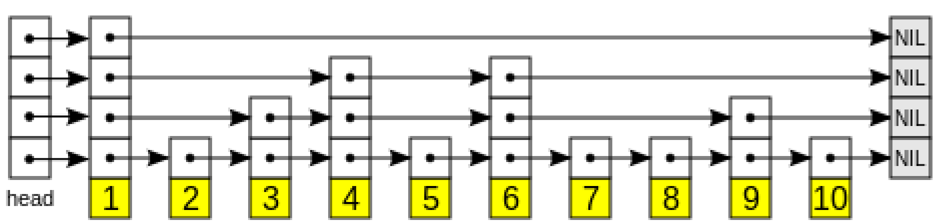

Figura 1: Estructura de Skip List.

La estructura de una Skip List se aprecia en la figura 1. La altura de cada nodo (cantidad de punteros que tiene en listas superiores) se determina del siguiente modo: se elige con probabilidad $\frac{1}{2}$ si el nodo tiene un puntero en el nivel i (comenzando con $i = 2$), y en caso afirmativo se procede con la elección para el nivel $i + 1$, en caso negativo se termina el proceso. Todo nodo tiene al menos altura 1 (está presente en la primera lista).

Para implementar esta estructura se definieron las clases SkipNode y SkipList.

2.1.1. Clase SkipNode

Representa a cada nodo y sus duplicados en listas superiores. Consiste simplemente en una llave *key* y una lista *next* de tamaño n, donde para cada $i \in [0, n)$, $next_i$ es un puntero al siguiente elemento de la i-ésima lista de la Skip List.

2.1.2. Clase SkipList

Representa a la lista en sí, la cual está conformada por un SkipNode *head* y las operaciones de inserción y búsqueda.

Ambas operaciones hacen uso del método $closest\_nodes(k)$, el cual entrega una lista de tamaño igual a la altura de la Skip List, y que contiene punteros a los nodos cuyas llaves tienen los valores más cercanos a k por la izquierda (con valor menor a k) en cada nivel. El método funciona del siguiente modo:

closest_nodes(k):

- Se crea una lista u de tamaño igual a la altura de la Skip List, inicializada a $None$.
- Se asigna $node := head$.
- Se itera i desde $|u|$ a 1:
  - Mientras $node.next[i] \neq None$ y $node.next[i].key < k$
    - $node := node.next[i]$
  - $u[i] := node$
- Retorna u

Luego, podemos definir la inserción del siguiente modo:

insert(k):

- Se crea un SkipNode *node* con la llave k que se desea insertar, y una altura determinada con el método explicado en 2.1.
- Si la altura de *node* es mayor a la de *head*, se añaden a *head* los niveles faltantes (con $None$).
- Se obtiene la lista de nodos más cercanos por la izquierda en cada nivel, $u := closest\_nodes(k)$
- Para cada nivel i de la altura de *node*:
  - $node.next[i] := u[i].next[i]$
  - $u[i].next[i] := node$

Finalmente, la búsqueda queda:

search(k):

- Se obtiene la lista de nodos más cercanos al valor de k por la izquierda en cada nivel, $u := closest\_nodes(k)$.
- Si la longitud de u es mayor que cero:
  - Se asigna $c := u[0].next[0]$ (si el nodo cuya llave es k está en el árbol, ahora será apuntado por c).
  - Si $c \neq None$ y $c.key = k$, c es el nodo buscado y por ende se retorna.
- Si se llegó a este punto la búsqueda fue infructuosa y se retorna $None$.

2.2. ABB

El ABB se implementó con una clase $Node$ y una clase ABB. Cada objeto $Node$ contiene una llave key y los punteros a sus hijos $left$ y $right$. Una instancia de ABB contiene a un objeto $Node$ $root$ y las operaciones $insert$ y $search$. Ambas operaciones consisten en una mera búsqueda binaria, que en el caso de la inserción al encontrar un nodo vacío ($None$) inserta un nuevo nodo en ese espacio. Las operaciones se implementaron de forma iterativa en vez de recursiva para evitar el desborde de la pila al insertar secuencias ordenadas extensas, ya que Python no provee optimización de llamada por la cola.

2.3. ABB Aleatorizado

En cada ABB aleatorizado T, debe cumplirse que para todo sub-árbol T' cada nodo tiene probabilidad $\frac{1}{|T'|}$ de ser la raíz, donde $|T'|$ es la cantidad de nodos de T'. Esta propiedad se cumple cuando al insertar un nodo en un sub-árbol T', este nodo tiene probabilidad $\frac{1}{|T'|+1}$ de convertirse en la raíz de T'. Esto puede lograrse generando un número n pseudoaleatorio distribuido uniformemente entre 0 y $|T'|$, y luego si $n = |T'|$ insertamos al nodo como raíz de T'. Para calcular $|T'|$ en cada T' fue necesario agregar el campo $children$ a la clase $Node$, el cual almacena la cantidad de descendientes de este nodo.

Para que un nodo se convierta en raíz de un árbol manteniendo la consistencia del mismo lo que se hace es rotarlo desde su posición de inserción hasta que quede en la raíz. A continuación se muestra el algoritmo de inserción, dividido entre la inserción normal y la inserción en la raíz.

insert(k):

- $self.root := insert\_rec(self.root, k)$

insert_rec(node, k):

- Si $node = None$, retorna un nuevo nodo de llave k.
- Si $random(0, node.children + 1) = node.children$
  - Retorna $insert\_at\_root(node, k)$
- Si $k < node.key$, $node.left := insert\_rec(node.left, k)$
- En caso contrario, $node.right := insert\_rec(node.right, k)$
- $node.children := node.children + 1$
- Retorna $node$

insert_at_root(node, k):

- Si $node = None$, retorna un nuevo nodo de llave k.
- Si $k < node.key$:
  - $node.left := insert\_at\_root(node.left, k)$
  - $node = rotate\_right(node)$
- En caso contrario:
  - $node.right := insert\_at\_root(node.right, k)$
  - $node = rotate\_left(node)$
- Retorna $node$

Las rotaciones de nodo se implementaron según el esquema de la figura 2. Estas además actualizan el campo *children* del nodo rotado.

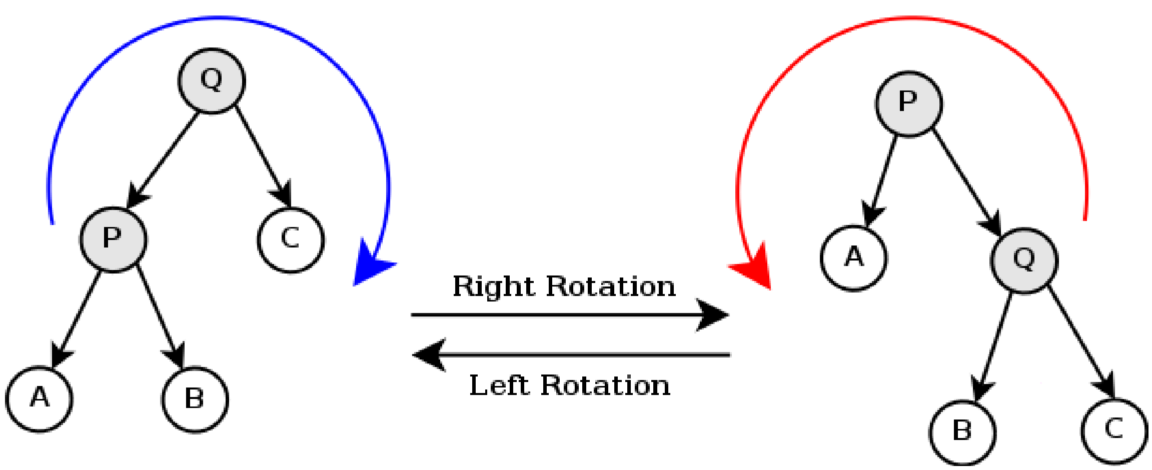

Figura 2: Operaciones de rotación de nodos, hacia la derecha e izquierda.

3. Experimentos

3.1. Tiempos de inserción y búsqueda

3.2. Altura promedio de Skip List

3.3. Comparación entre ABB y ABB aleatorizado

4. Conclusión