Árboles Balanceados

Pablo R. Ramis

Universidad Nacional de Rosario, Instituto Politécnico, Dto. de Informática,
prramis@ips.edu.ar,
WWW home page: http://informatica.ips.edu.ar

Resumen Hemos visto algunos de los conceptos que abarcan a las estructuras de datos abstractas, entre ellos la estructura del Árbol binario, si hacemos un análisis de tiempos de las operaciones para árboles binarios de búsqueda, el comportamiento promedio nos permite ver que al insertar n elementos aleatorios en dicho árbol inicialmente vacío, la longitud de camino promedio de la raíz a una hoja es $O(\log n)$ y por tanto la prueba de pertenencia lleva un tiempo $O(\log n)$.
Si imaginamos a un árbol binario de n nodos completo, o sea, todos sus nodos tiene dos hijos (excepto el los que están en el último nivel) ningún camino tendrá mas de $1 + logn$ nodos. Así, los procedimientos MIEMBRO, INSERTA, SUPRIME y SUPRIME_MIN llevan un tiempo $O(\log n)$, con esto decimos que el "esfuerzo" en llegar a un nodo u otro de la estructura es el mismo, constante.
Sin embargo, al insertar elementos en un orden aleatorio no es seguro que se acomoden formando un arbol completo. Por ejemplo, si el primer elemento insertado es el más pequeño, el árbol resultante se balanceará en su totalidad hacia el hijo derecho del inicio.
Podemos pensar algunas secuencias de inserciones y eliminaciones pueden producir árboles binarios de búsqueda cuya profundidad promedio sea proporcional a n. Esto sugiere que se puede hacer un intento de reordenar el árbol despues de cada inserción o eliminación para que siempre este completo.
Podemos ver en la Figura 1 un ejemplo de lo que decimos. Un árbol de 6 nodos se convierte en completo de 7 cuando insertamos el elemento 1.

1. Árboles Balanceados

Al ver la imagen mencionada, la Figura 1, vemos que de un árbol a otro, todos todos los nodos cambian de padres, esto implica tomarse n pasos para insertar un nuevo elemento, en este caso fue el 1.

Para lograr esto, tendremos que tener otro enfoque, como por ejemplo "árbol AVL".

Un árbol AVL es un tipo de árbol binario de búsqueda autobalanceado que garantiza un tiempo de búsqueda, inserción y eliminación de $O(logn)$. Esto se logra asegurando que la diferencia de alturas entre los subárboles izquierdo y derecho de cualquier nodo (conocida como el factor de balanceo) sea, como máximo, 1.

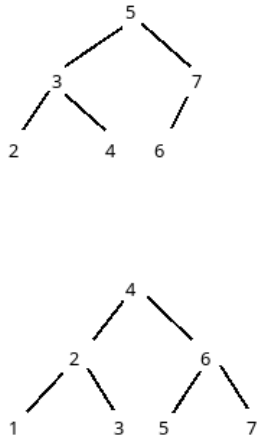

Figura 1. Árboles completos

A continuación, se proporciona una explicación algorítmica en pseudocódigo de las operaciones principales en un árbol AVL: búsqueda, inserción y eliminación, junto con las rotaciones necesarias para mantener el balance del árbol.

1.1. Estructura del Nodo

```

Nodo:
    dato
    altura
    h_izq
    h_der
```

La altura de un nodo es la distancia al nodo más profundo desde él. Se utiliza para calcular el factor de balanceo

Cálculo de la altura

```

altura(nodo);
    si nodo es NULO:
        retornar 0
    sino
        retornar nodo.altura
```

Cálculo de Factor de Balance Se calcula como la diferencia entre la altura del subárbol izquierdo y derecho. Si el factor es mayor a 1 o menor a -1, el árbol está desbalanceado.

- Factor Balance = altura($nodo \rightarrow h\_izq$) - altura($nodo \rightarrow h\_der$)
- Los valores permitidos tiene que ser 1, 0 y -1
- el valor -1 indica que el subárbol derecho contiene uno adicional
- el valor 1 indica que el subárbol izquierdo contiene uno adicional
- el valor 0 indica que ambos subárboles poseen el mismo nivel, o sea, esta perfectamente balanceado.

En la figura 2 vemos un ejemplo en donde se indica en los nodos la altura del mismo y podemos apreciar que el nodo indicado con el asterico no cumple con la condición mencionada antes.

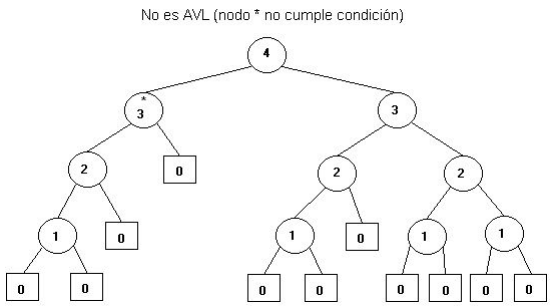

Figura 2. Árbol desbalanceado

```
1
2  factorBalanceo(nodo):
3      si nodo es NULO:
4          retornar 0
5      sino altura(nodo.h_izq) - altura(nodo.h_der)
```

Teniendo en cuenta este factor, se debe tomar la estrategia de corrección del balance del árbol, a esto lo definiremos como ***rotaciones***

- Rotación simple a la derecha: corrige desbalance “left-left”
- Rotación simple a la izquierda: corrige desbalance “right-right”
- Rotación doble izquierda-derecha: corrige un desbalance “left-right”
- Rotación doble derecha-izquierda: corrige un desbalance “right-left”

Rotación simple a la Derecha

```

rotacionDerecha(y):
    x = y.h_izq
    T = x.h_der

    x.h_der = y
    y.h_izq = T

    actualizarAltura(y)
    actualizarAltura(x)

    retornar x
```

Rotación simple a la Izquierda

```

rotacionIzquierda(x):
    y = x.h_der
    T = y.h_izq

    y.h_izq = x
    x.h_der = T

    actualizarAltura(x)
    actualizarAltura(y)

    retornar y
```

Rotación doble izquierda-derecha

```

rotacionIzquierdaDerecha(nodo):
    nodo.h_izq = rotacionIzquierda(nodo.h_izq)
    retornar rotacionDerecha(nodo)
```

Rotación doble derecha-izquierda

```

rotacionDerechaIzquierda(nodo):
    nodo.h_der = rotacionDerecha(nodo.h_der)
    retornar rotacionIzquierda(nodo)
```

1.2. Insertar

La función *insertar* tiene una mayor complejidad que en el árbol binario tradicional ya que por cada nuevo elemento analizará el balance y utilizará las funciones anteriores para mantener equilibrado el árbol.

```

insertar(nodo, valor):
    si nodo es NULO:
        retornar nuevo Nodo(valor)

    si valor < nodo.dato:
        nodo.izquierdo = insertar(nodo.h_izq, valor)
    sino si valor > nodo.dato:
        nodo.derecho = insertar(nodo.h_der, valor)
    sino:
        devolver nodo  // Valor duplicado no permitido

    // Actualizar altura del nodo
    nodo.altura = 1 + max(altura(nodo.h_izq), altura(nodo.
        h_der))

    // Calcular factor de balanceo
    balance = factorBalanceo(nodo)

    // Casos de desbalance
    si balance > 1 y valor < nodo.h_izq.dato:
        retornar rotacionDerecha(nodo)

    si balance < -1 y valor > nodo.h_der.dato:
        retornar rotacionIzquierda(nodo)

    si balance > 1 y valor > nodo.h_izq.dato:
        retornar rotacionIzquierdaDerecha(nodo)

    si balance < -1 y valor < nodo.h_der.dato:
        retornar rotacionDerechaIzquierda(nodo)

    retornar nodo
```

1.3. Eliminación

La función *eliminar* tiene la misma situación y criterio que la *insertar*, se tendrá que rebalancear el árbol por cada operación que se realice.

```

obtenerMinimo(nodo):
```

```
    mientras nodo.h_izq no sea NULO:
        nodo = nodo.h_izq
    devolver nodo

eliminar(nodo, valor):
    si nodo es NULO:
        retornar nodo

    si valor < nodo.dato:
        nodo.h_izq = eliminar(nodo.h_izq, valor)
    sino si valor > nodo.dato:
        nodo.h_der = eliminar(nodo.h_der, valor)
    sino:
        // Nodo encontrado
        si nodo.h_izq es NULO o nodo.h_der es NULO:
            nodo = nodo.h_izq si nodo.h_izq no es NULO sino
                nodo.h_der
        sino:
            sucesor = obtenerMinimo(nodo.h_der)
            nodo.dato = sucesor.dato
            nodo.h_der = eliminar(nodo.h_der, sucesor.dato)

    si nodo es NULO:
        retornar nodo

    // Actualizar altura
    nodo.altura = 1 + max(altura(nodo.h_izq), altura(nodo.
        h_der))

    // Calcular factor de balanceo
    balance = factorBalanceo(nodo)

    // Casos de desbalance
    si balance > 1 y factorBalanceo(nodo.h_izq) >= 0:
        retornar rotacionDerecha(nodo)

    si balance > 1 y factorBalanceo(nodo.h_izq) < 0:
        retornar rotacionIzquierdaDerecha(nodo)

    si balance < -1 y factorBalanceo(nodo.h_der) <= 0:
        retornar rotacionIzquierda(nodo)

    si balance < -1 y factorBalanceo(nodo.h_der) > 0:
        retornar rotacionDerechaIzquierda(nodo)

    retornar nodo
```

1.4. Mostrar el árbol

Al ser un simple árbol binario, cualquiera de los metodos de recorrido que hemos visto nos permitirian mostrar los datos, sin embargo, para ver si está balanceado realmente, tendriamos que buscar un método que nos permita verlos por niveles.

A este método lo llamaremos recorrido por niveles o a lo ancho *(bft - breadth first traversal*. Para esto se tiene que utilizar una estructura de *cola*. En este recorrido, se visitan primero los nodos de cada nivel, comenzando desde la raíz, y luego se avanza hacia los niveles inferiors.

Pasos de implementación

- encolar el nodo raíz
- mientras la cola no este vacía
  - desencolar el nodo
  - procesar ese nodo (imprimir el valor)
  - encolar sus hijos izquierdo y derecho si existen

La cola puede ser arreglo circular o una lista enlazada.
El código para la cola sería el siguiente:

```
1
2  // Estructura para la cola
3  typedef struct Cola {
4      Nodo** arreglo; // Arreglo de punteros a nodos
5      int frente;
6      int ultimo;
7      int capacidad;
8  } Cola;
9
10 // Funciones para manejar la cola
11 Cola* crearCola(int capacidad) {
12     Cola* cola = (Cola*)malloc(sizeof(Cola));
13     cola->arreglo = (Nodo**)malloc(capacidad * sizeof(Nodo*))
           ;
14     cola->frente = 0;
15     cola->ultimo = -1;
16     cola->capacidad = capacidad;
17     return cola;
18 }
19
20 int colaVacia(Cola* cola) {
21     return cola->ultimo < cola->frente;
22 }
23
24 void encola(Cola* cola, Nodo* nodo) {
25     cola->ultimo++;
```

```
26      cola->arreglo[cola->ultimo] = nodo;
27  }
28
29  Nodo* desencola(Cola* cola) {
30      Nodo* nodo = cola->arreglo[cola->frente];
31      cola->frente++;
32      return nodo;
33  }
```

Recorrido a lo ancho

```
1
2   recorridoHorizontal(nodo):
3       si nodo es nulo
4               retornar;
5           cola = crearCola;
6           encolar(cola, nodo);
7
8           mientras cola distinto a vacio
9               nodoNuevo = desencolar(cola);
10              imprimir(nodoNuevo);
11
12              si nodoNuevo.h_izq distinto a nulo
13                  encolar(cola, nodoNuevo.h_izq);
14              si nodoNuevo.h_der distinto a nulo
15                  encolar(cola, nodoNuevo.h_der);
```