Universidad Nacional de San Agustín, Escuela de ..

Árbol AVL

Christian E. Portugal Zambrano

30 de agosto de 2018

1. Introducción

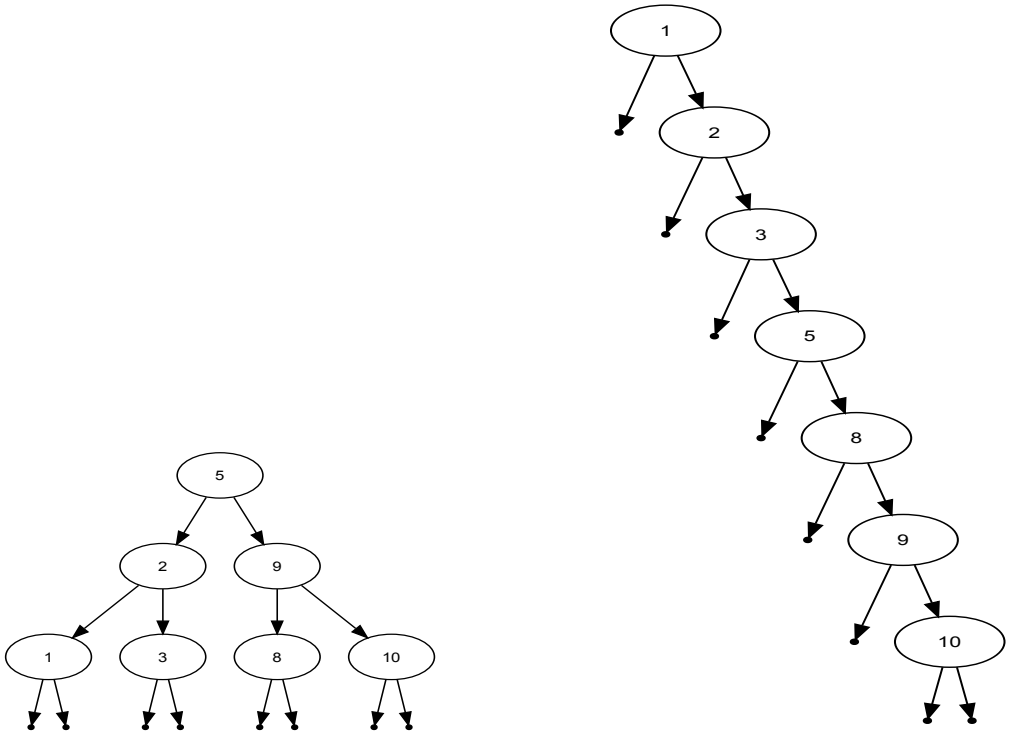
Los árboles de búsqueda binarios son árboles binarios en orden simétrico, cada nodo tiene una llave y cada llave del nodo puede ser mayor que todas las llaves en su sub-árbol izquierdo o menor que todas las llaves de su sub-árbol derecho, también un árbol binario puede estar vacío o formado por dos sub-árboles disjuntos.
Un problema de los árboles de búsqueda binarios es la forma de inserción y borrado, de acuerdo a esto algunos árboles tienden a desnaturalizarse o perder sus propiedades de búsqueda. En este trabajo implementamos un árbol AVL que agrega un comportamiento de balanceo de nodos a los árboles de búsqueda ordinarios para resolver el problema de inserción y borrado.

1.1. Árboles binarios de búsqueda

Descrito anteriormente los árboles de búsqueda binarios dependen en cierto modo de la forma de inserción de sus llaves si existe un orden en los datos el árbol se visualiza y comporta como una lista enlazada, este es el peor de los casos, sin embargo en [2] se propone que si N llaves distintas son insertadas aleatoriamente en un árbol binarios de búsqueda, la altura esperada del árbol es $\sim 4.311 \ln N$ pero en el peor de los casos la altura es N. En la Figura 1.1 se muestra árboles de búsqueda binaria balanceados (a) y no balanceados (b).

1.2. Árboles balanceados

Uno de los primeros trabajos en abordar el problema de la inserción aleatoria en árboles binarios de búsqueda fue el de [1]; conocido también como AVL debido a los apellidos



(a) (b)

Figura 1.1: Árboles de búsqueda binarios (a) Inserción ideal en el árbol, se logra una altura mínima y árbol completo (lleno), (b) inserción de datos ordenados, adquiere el comportamiento de una lista enlazada en donde el tiempo de búsqueda en el peor de los casos es lineal.

de los dos autores; su propuesta se basa en mantener un factor de balance no mayor que 1 o menor que -1, entonces después de cada operación de inserción o borrado se debe de revisar la altura de los sub-árboles izquierdo y derecho, al encontrar una violación al factor de balance los autores proponen un conjunto de rotaciones con la finalidad de balancear la altura del árbol final.

1.2.1. Altura

Se presenta el Algoritmo 1 para calcular la altura de un nodo. En caso de que el nodo no exista se retornará cero.

Algoritmo 1 Cálculo la altura de un nodo

```
1: procedure HEIGHT(node)
2:     if x == NULL then
3:         Return 0
4:     else
5:         Return node.height
```

1.2.2. Factor de balance

Se define como la diferencia entre la altura del sub- árbol derecho menos la altura del sub-árbol izquierdo. Se presenta el Algoritmo 2 para calcular este factor de balance.

Algoritmo 2 Cálculo del factor de balance de un nodo

```
1: procedure BALANCEFACTOR(node)
2:     Return Height(node.right) - Height(node.left)
```

1.2.3. Actualización de altura

En cada inserción o borrado se debe de revisar la altura del sub-árbol afectado, se presenta el Algoritmo 3 para realizar esta tarea.

Algoritmo 3 Revisión de la altura de un sub-árbol

```
1: procedure UPDATEHEIGHT(node)
2:     HL ← Height(node.left)
3:     HR ← Height(node.right)
4:     if  HL >HR then
5:         node.height ← HL + 1
6:     else
7:         node.height ← HR + 1
```

1.2.4. Rotación derecha

Se utiliza cuando se inserta en un sub-árbol izquierdo de otro sub-árbol, el Algoritmo 4 lo detalla. Otros autores consideran un conjunto de rotaciones como rotación derecha izquierda y aún más complicadas rotación derecha izquierda derecha, algunos autores proponen que aún realizando mayores rotaciones la altura del árbol es la misma que sólo realizando rotaciones simples.

1.2.5. Rotación izquierda

La rotación izquierda es simétrica a la rotación derecha y se origina cuando se inserta en un sub-árbol derecho de otro sub-árbol derecho. En el Algoritmo 5 se describe su función.

Algoritmo 4 Rotación derecha en un sub-árbol

```
1: procedure RightRotation(node)
2:     temp ← node.left
3:     node.left ← temp.right
4:     temp.right ← node
5:     updateHeight(node)
6:     updateHeight(temp)
```

Algoritmo 5 Rotación izquierda en un sub-árbol

```
1: procedure leftRotation(node)
2:     temp ← node.right
3:     node.right ← temp.left
4:     temp.left ← node
5:     updateHeight(node)
6:     updateHeight(temp)
```

2. Implementación

Para la visualización del árbol AVL luego de un conjunto de operaciones se ha utilizado GraphViz[1], esta biblioteca utiliza un lenguaje para la generación de grafos con nodos, se ha utilizado un método show que permita generar este código a partir de la estructura del árbol y luego desde una consola de sistema se utiliza el programa dot de Graphviz para la generación de la imagen. Ej. Para generar una imagen a partir de un archivo bstree.dot se ejecuta el siguiente comando:

dot -Tpdf bstree.dot -o bstree.pdf

La opción -T es para indicar el formato de salida de la imagen, puede ser png, jpg, pdf, etc., la opción -o es para indicar la salida del programa, en este caso se indica el nombre del formato de salida. Todas las imágenes en este documento fueron generadas son el soporte de esta herramienta.
Para la implementación de esta estructura se ha utilizado dos clases **avlNode** y **avlTree**, y se han implementado los métodos básicos de un árbol como insert, search, remove y algunos métodos privados como min, max y deleteMin, estos últimos sirven para alcanzar los objetivos de los primeros métodos. También debemos de resaltar que un árbol AVL posee los métodos de un árbol binario de búsqueda y adiciona algunos para el manejo de rotaciones y cálculo de la altura.
Implementamos nuestra clase AVLNode como sigue:

```
template <class key, class value>
class AVLNode{
   key _Key;
   value _Value;
```

[1] http://www.graphviz.org/Home.php

```
    AVLNode<key,value>* left;
    AVLNode<key,value>* right;
    short _Height;
public:
    AVLNode(key _key, value _value):
        _Key(_key),_Value(_value),left(0),right(0),_Height(1){}
    AVLNode<key,value>*& Left() {return left;}
    AVLNode<key,value>*& Right(){ return right;}
    value GetValue(){return _Value;}
    void SetValue(value _value){ _Value = _value;}
    short GetHeight(){ return _Height;}
    void SetHeight(int _height){ _Height = _height;}
    key Key(){return _Key;}
};
```

También implementamos nuestro árbol AVL como:

```
template <class key,class value>
class AVLTree{
    AVLNode<key,value>* root;
    int nullCounter;  //to print null in the dot file
    AVLNode<key,value>* insert(AVLNode<key,value>* _node,
          key _key, value _value);
    AVLNode<key,value>* remove(AVLNode<key,value>*,key);
    AVLNode<key,value>* min(AVLNode<key,value>*);
    AVLNode<key,value>* max(AVLNode<key,value>*);
    AVLNode<key,value>* deleteMin(AVLNode<key,value>*);
    AVLNode<key,value>* rightRotation(AVLNode<key,value>*);
    AVLNode<key,value>* leftRotation(AVLNode<key,value>*);
    AVLNode<key,value>* balance(AVLNode<key,value>*);
    short height(AVLNode<key,value>*);
    short balanceFactor(AVLNode<key,value>*);
    void updateHeight(AVLNode<key,value>*);
    void deleteMin();
    void show(AVLNode<key,value>*,std::ostream &);

public:
    AVLTree():root(0),nullCounter(0){}
    void insert(key _key, value _value);
    void remove(key _key); //Hibbard deletion
    value search(key _key);
    void show(std::ostream&);
};
```

Debemos observar que para el método de eliminación de nodos se utiliza e algoritmo de Hibbard con un paso adicional sobre la revisión de alturas. Para detalles de la implementación de este código revisar el archivo AVLTree.h que acompaña este documento.

3. Resultados y conclusiones

Se realizaron pruebas validando el funcionamiento de la implementación con un conjunto de inserciones y eliminaciones, en la Figura 3.1 presentamos el resultado de insertar las llaves 10, 8, 11, 14, 20, 3, 5, 30, 24, 16, 11, 1, 32, 21, 13, 9, 17, 27, 31, 2. en un árbol binario de búsqueda.

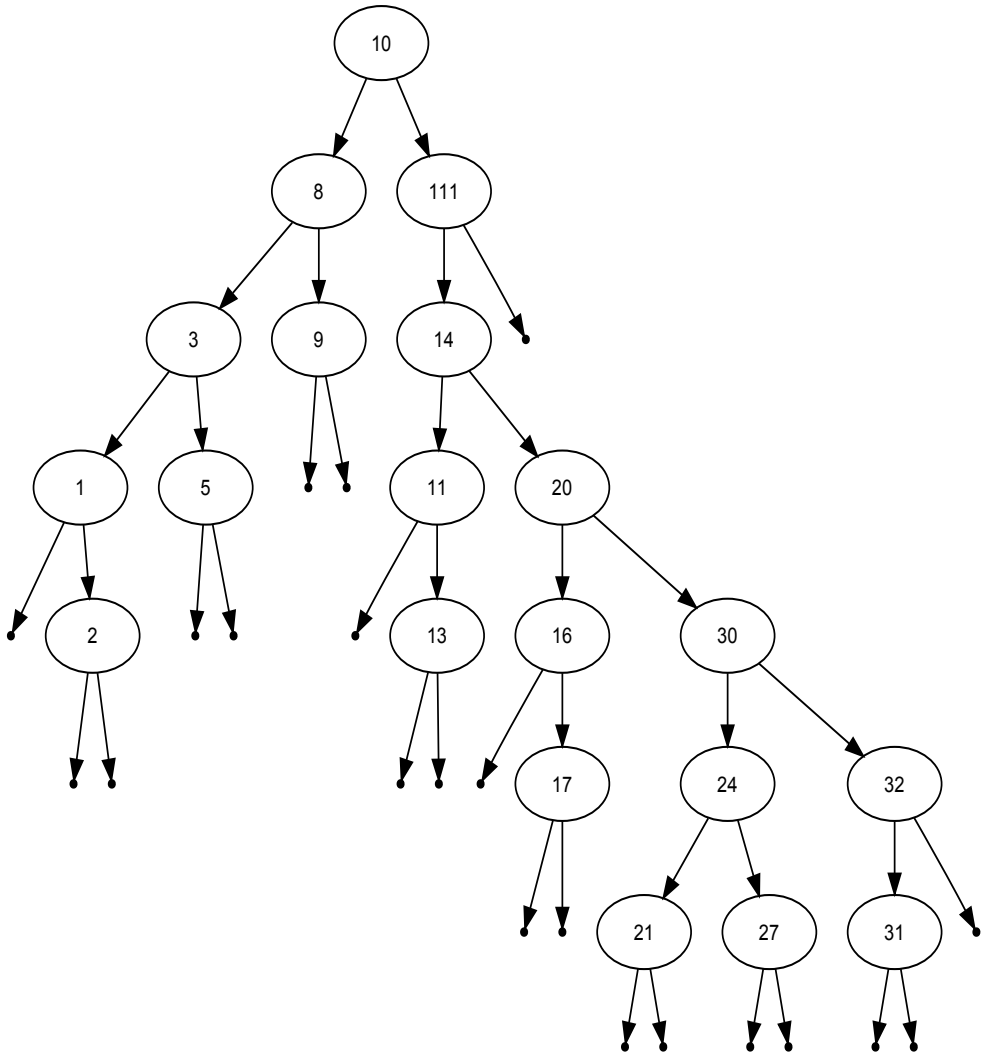


Figura 3.1: Inserción de llaves 10, 8, 11, 14, 20, 3, 5, 30, 24, 16, 11, 1, 32, 21, 13, 9, 17, 27, 31, 2. en un árbol binario de búsqueda

Luego realizamos un conjunto de eliminaciones con las siguientes llaves 20, 9, 30. y obtenemos el árbol mostrado en la Figura 3.2.

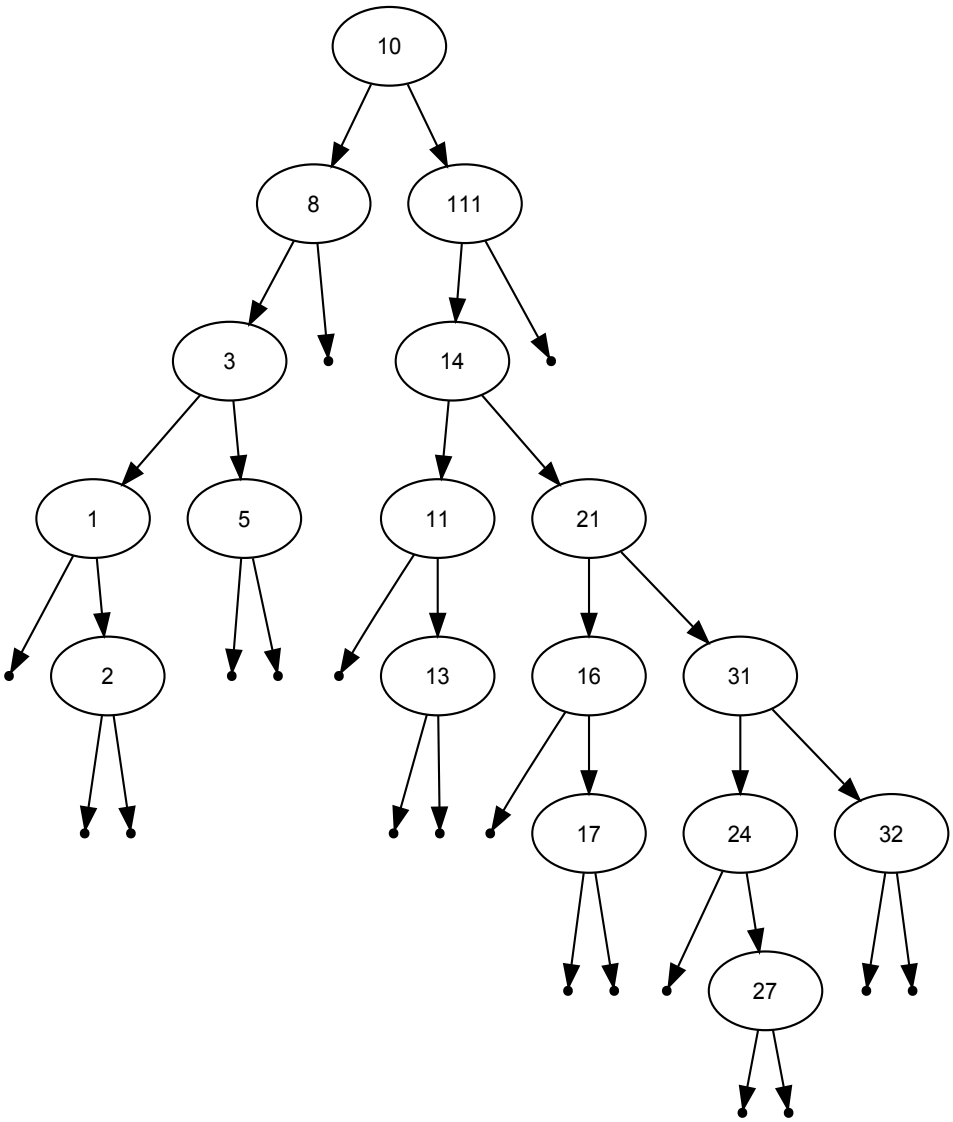


Figura 3.2: Árbol resultado de eliminar las llaves $20, 9, 30$ en el árbol de búsqueda binaria de la Figura 3.1

También, en la Figura 3.3 presentamos el resultado de insertar las mismas llaves pero en un árbol AVL.

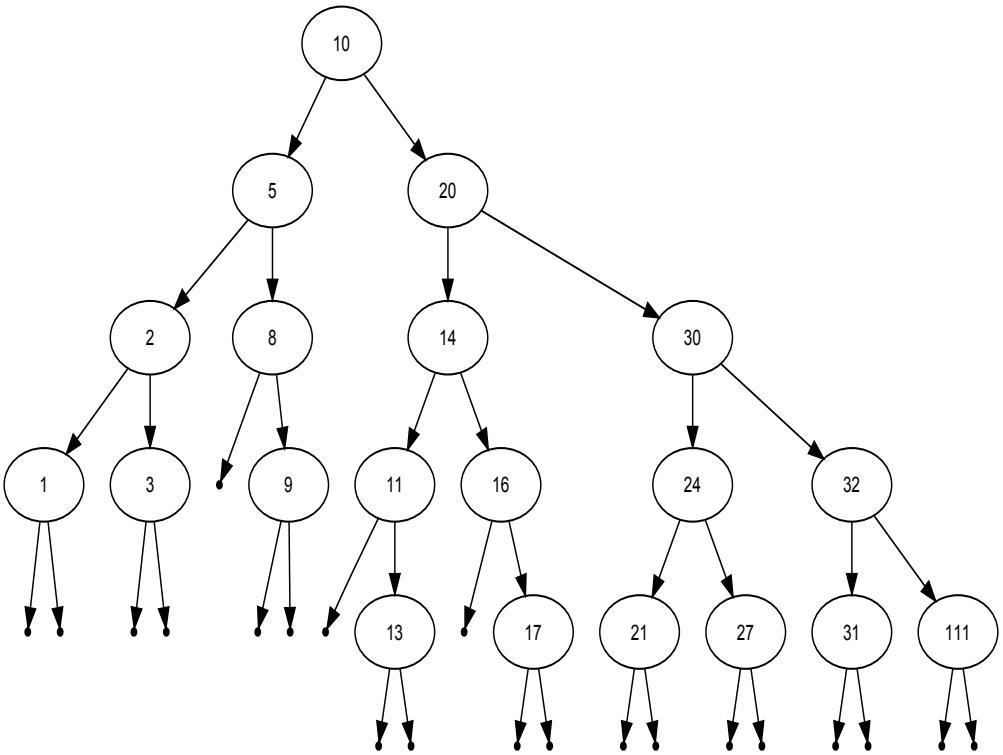
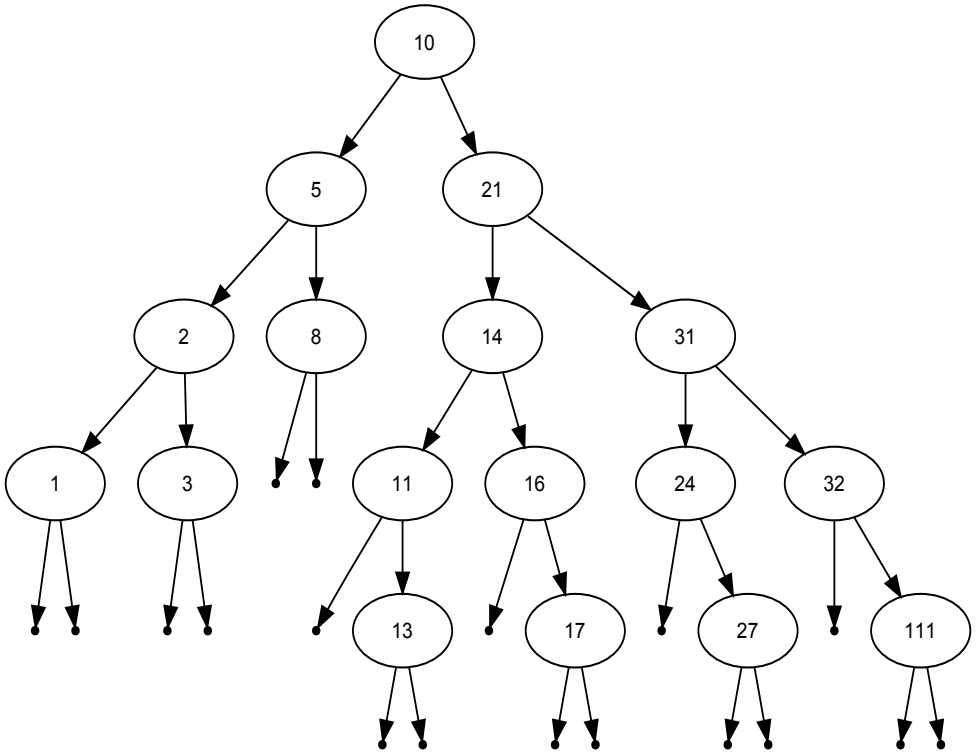


Figura 3.3: Inserción de las llaves 10, 8, 11, 14, 20, 3, 5, 30, 24, 16, 11, 1, 32, 21, 13, 9, 17, 27, 31, 2. en un árbol AVL

Finalmente, en la Figura 3 se presenta el resultado de eliminar las llaves 20, 9, 30 en el árbol AVL de la Figura 3.3.



3.1. Conclusiones

El árbol AVL agrega métodos para evitar el crecimiento en altura de los árboles binarios de búsqueda, se estima que mantiene su valor en $\sim 1.44 \ln N$. Existen un conjunto de rotaciones que ayudan a mantener la altura, pero se logran buenos resultados utilizando las rotaciones básicas a la derecha e izquierda.
Existen otros árboles como el árbol rojo-negro que considera otros factores que permiten obtener tiempos logarítmicos en inserción y búsqueda, sin embargo el árbol AVL permite sustentar las bases de estos algoritmos más avanzados.

Referencias

[1] *An information organization algorithm*, Adelson-Velskii, Georgy Maximovic and Landis, Evgenii Mikhailovich, Doklady Akademia Nauk SSSR, vol. 146, pag. 263–266, 1962

[2] *How tall is a tree?*, Reed, Bruce, Proceedings of the thirty-second annual ACM symposium on Theory of computing, pag. 479–483, ACM, 2000