DOCUMENTACIÓN

Árbol AVL

UNIVERSIDAD DE COSTA RICA,
ESCUELA DE INGENIERIA ELECTRICA

Contenidos

1 Iniciar con la estructura de datos

Para comenzar a usar esta estructura de datos se debe crear un archivo principal desde el cual correr las diferentes funciones del programa (functions.cpp). Dentro de este, se deben agregar las librerías <iostream> y <algorithm>, así como el archivo de cabecera "functions.h"; también se deben inicializar punteros tipo "Node*" en los cuales se guardarán los nodos deseados, incluyendo el nodo raíz del árbol. Además es necesario definir la lista de datos que se quieren añadir a la estructura, como se muestra a continuación:

```
1  #include "functions.h"
2  #include<iostream>
3  #include<algorithm>
4
5  int main()
6  {
7      float arr[]={1,2,3,4,5};
8      int size = (int)sizeof(arr)/sizeof(float);
9
10     Node *root = NULL;
11     Node *min= NULL;
12     Node *max= NULL;
13
14 }
```

Figure 1: Codigo inicial para el uso de la estructura AVL tree.

2 Codigos de error

Cada vez que una función se utilice, esta devuelve un valor, cada valor que entregue tiene un significado, a continuación se presenta la lista de valores que puede retornar un función:

- AVL_SUCCES = 0
- AVL_INVALID_PARAM = -1
- AVL_OUT_OF_RANGE = -2
- AVL_TIMEOUT = -3
- AVL_NOT_FOUND = -4

3 Crear un árbol AVL

Para ordenar la lista de datos que ya hemos definido basta con llamar a la función avl_create con los parámetros correspondientes:

3.1 Funciones avl_create y avl_node_add

La función avl_create se encarga de iniciar el árbol con ayuda de otra función llamada avl_node_add, de la cual se hablara más adelante. Para cada elemento de la lista ingresada avl_create se encarga de acomodar cada elemento, al mismo tiempo corrobora que cada dato sea válido y se haya acomodado exitosamente.

```
1   #include "functions.h"
2   #include<iostream>
3   #include<algorithm>
4
5   int main()
6   {
7       float arr[]={1,2,3,4,5};
8       int size = (int)sizeof(arr)/sizeof(float);
9
10      Node *root = NULL;
11      Node *min= NULL;
12      Node *max= NULL;
13
14      avl create(size,arr,&root);
15
16  }
```

Figure 2: Llamado a la funcion avl_create.

En caso de que haya un elemento repetido en el arreglo, el error_code será igual a -1 y no 0 pues 0 es un valor de retorno exitoso, pero en caso de ingresar el mismo valor dos veces en la lista, no se estaría teniendo éxito. Por su parte, la función node_add se encarga de encontrar la posición, mediante una acción recursiva, dónde se debe ir ubicando cada nodo nuevo (iniciando desde la raíz actual). Una vez haya colocado el nuevo nodo, se actualizan las alturas y se procede a balancear el árbol en caso se de ser necesario.

```
int avl create(int size, float* list, Node ** new root){

    int error code = 0;

    for (int i = 0; i < size; i++) {
        error code = avl node add(*new root, list[i], new root);
    }

    return error code;
}
```

Figure 3: Función avl_create.

Estas dos funciones, de acuerdo con la lista que definimos en un un inicio, crean un árbol avl que gráficamente se puede ver de esta manera:

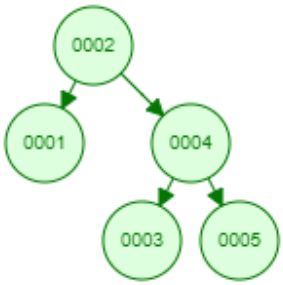

Figure 5: Árbol avl de ejemplo, de acuerdo al arreglo definido.

4 Añadir un dato

Para ingresar un nuevo dato, solamente bastante llamar la función nuevamente avl_node_add con el dato que se desee ingresar como se observa en la siguiente figura.

```
int avl node add(Node* in root, float num, Node** new root)
{

    int error code=0;
    Node* nuevo = new Node();
    //Caso en que el nodo este vacio
    if (in root == NULL){

        nuevo->num = num;
        nuevo->left = NULL;
        nuevo->right = NULL;
        nuevo->height = 1;
        *new root=nuevo;

        return error code;
    }
    if (num > in root->num)
    {
        error code = avl node add(in root->right, num, &in root->right);
    }
    else if (num < in root->num)
    {
        error code = avl node add(in root->left, num, &in root->left);
    }
    else {// No se pueden numeros iguales

        return -1;
    }

    // Actualizamos las alturas

    in root->height = 1 + max(height(in root->left),
                           height(in root->right));

    // Encuentra el balance, y si no es adecuado, lo balancea, retorna el puntero a la raiz
    int balance = balanceAVL(in root);


    // Left Left Case
    if (balance > 1 && num < in root->left->num)
        return rightRotate(in root, new root);

    // Right Right Case
    if (balance < -1 && num > in root->right->num)
        return leftRotate(in root, new root);

    // Left Right Case
    if (balance > 1 && num > in root->left->num)
    {
        error code = leftRotate(in root->left, &in root->left);
        return rightRotate(in root, new root);
    }

    // Right Left Case
    if (balance < -1 && num < in root->right->num)
    {
        error code = rightRotate(in root->right, &in root->right);
        return leftRotate(in root, new root);
    }

    *new root = in root;
    return error code;
}
```

Figure 4: Función node_add.

```
1   #include "functions.h"
2   #include<iostream>
3   #include<algorithm>
4
5   int main()
6   {
7       float arr[]={1,2,3,4,5};
8       int size = (int)sizeof(arr)/sizeof(float);
9
10      Node *root = NULL;
11      Node *min= NULL;
12      Node *max= NULL;
13
14      avl create(size,arr,&root);
15      avl node add(root,6, &root);
```

Figure 6: Uso de función avl_node_add.

De acuerdo con el ejemplo, así se ve el árbol con el nuevo dato añadido.

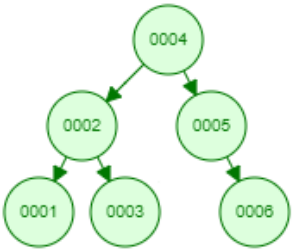

Figure 7: Árbol avl de ejemplo, dato añadido.

5 Eliminar un dato

Si se quiere eliminar un dato, basta con llamar a esta función desde el main, como se muestra a continuación:

```
#include "functions.h"
#include<iostream>
#include<algorithm>

int main()
{
    float arr[]={1,2,3,4,5};
    int size = (int)sizeof(arr)/sizeof(float);

    Node *root = NULL;
    Node *min= NULL;
    Node *max= NULL;

    avl create(size,arr,&root);
    avl node add(root,6, &root);
    avl node remove(root,6, &root)

}
```

Figure 8: Uso de función avl_node_remove.

5.1 Función avl_node_remove

Esta función, para eliminar un nodo, recibe el puntero del nodo raíz, el valor del nodo que se desea eliminar y la dirección de un puntero en el cual se guardara el nuevo árbol, una vez se haya terminado el proceso.

Esta función opera de manera recursiva, llamándose a si misma con el nodo actual como nueva raíz de la siguiente iteración; cuya condición de parada es cuando se llega al final del árbol por alguna de sus ramas. En caso de no encontrar al nodo, la función recorre todo el árbol y retorna un -4 (valor no encontrado).

En caso de encontrar al nodo buscado, pueden pasar algunos escenarios. El primero es cuando el nodo eliminado no tiene hijos, en cuyo caso este nodo simplemente se borra. El segundo cuando el nodo tiene un hijo, en cuyo caso se elimina el nodo deseado y el hijo pasa a ser el nuevo nodo raíz. El tercer caso sucede cuando el nodo a eliminar tiene ambos hijos, en este caso lo que se realiza es intercambiar el nodo actual (se sabe que es el que se desea eliminar) con el nodo mínimo de la rama derecha del árbol y, se realiza una llamada recursiva a la función con el primer nodo de la rama derecha como raíz, Esto lo que propicia es que el nodo buscado este al final del árbol, de manera que cuando la

recursión termine, se llegue al caso anterior: el nodo buscado no tiene hijos y simplemente se elimina.

Por ultimo se procede a balancear el árbol de modo que cumpla con la condición de balance.

```
int avl node remove(Node* in root, float num, Node** new root)
{
    Node* temp = new Node();
    int error code=0;

    if (in root == NULL){
        *new root=in root;
        return -4;
    }

    // Si el valor buscado es
    //menor, se va a la izquierda
    if ( num < in root->num )

        error code = avl node remove(in root->left, num, &(in root->left));

    // Si el valor buscado es
    //mayor, se va a la derecha
    else if( num > in root->num )

        error code = avl node remove(in root->right, num, &(in root->right));

    else
    {

        if( (in root->left == NULL) ||
            (in root->right == NULL) )
        {
            temp = in root->left ?
                        in root->left :
                        in root->right;

            if (temp == NULL)
            {
                temp = in root;
                in root = NULL;
            }
            else
            *in root = *temp;

            free(temp);
        }
        else
        {

            error code = avl min get(in root->right, &temp);


            in root->num = temp->num;

            error code = avl node remove(in root->right, temp->num, &(in root->right));

        }
    }

    if (in root == NULL)
    {
        *new root=in root;
        return 0;
    }


    in root->height = 1 + max(height(in root->left),
                          height(in root->right));

    int balance = balanceAVL(in root);

    if (balance > 1 &&
        balanceAVL(in root->left) >= 0)
        return rightRotate(in root, new root);


    if (balance > 1 &&
        balanceAVL(in root->left) < 0)
    {
        error code = leftRotate(in root->left, &(in root->left));
        return rightRotate(in root, new root);
    }

    if (balance < -1 &&
        balanceAVL(in root->right) <= 0)
        return leftRotate(in root, new root);

    if (balance < -1 &&
        balanceAVL(in root->right) > 0)
    {
        error code = rightRotate(in root->right, &(in root->right));
        return leftRotate(in root, new root);
    }

    *new root=in root;
    return error code;
}
```

Figure 9: Uso de función avl_search.

De acuerdo con el ejemplo (borrar el número 6), el esquema gráfico que se tendría en este momento es igual al de la figura 5.

6 Buscar un dato

En el caso de que deseemos buscar un dato, podemos llamar a la función avl_search como se muestra a continuación:

```
#include "functions.h"
#include<iostream>
#include<algorithm>

int main()
{
    float arr[]={1,2,3,4,5};
    int size = (int)sizeof(arr)/sizeof(float);

    Node *root = NULL;
    Node *min= NULL;
    Node *max= NULL;

    avl create(size,arr,&root);
    avl node add(root,6, &root);
    avl node remove(root,6, &root)
    avl search(root,5, &searched)
}
```

Figure 10: Uso de función avl_search.

6.1 Función avl_search

Esta función empieza comparando el elemento que se desea buscar con elemento que está en la raíz, pues por este elemento es donde comienza la búsqueda. En caso de que el elemento buscado no coincida con el número de la raíz entonces se procede a buscar la ubicación de donde sí debería estar el elemento. Si la función encuentra al elemento nos devuelve un 0 (caso exitoso), si por el contrario esta no encuentra al elemento devuelve un -4, lo cual significa que elemento no se encontró.

```
int avl search(Node * root, float num searched, Node ** nodo){

    if (root == NULL) {
        *nodo = NULL;
        return -4;
    }
    else if (root->num == num searched){

        *nodo=root;
        return 0;
    }
    else if (num searched > root->num) {

        return avl search(root->right, num searched, nodo);
    }
    else if (num searched < root->num){

        return avl search(root->left, num searched, nodo);
    }

    return -1;
}
```

Figure 11: Función avl_search.

De acuerdo con el ejemplo, gráficamente la función estaría buscando al elemento como se muestra en la figura 12.

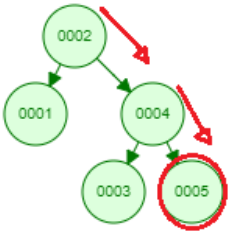

Figure 12: Elemento encontrado.

Si por ejemplo, llamáramos la función para buscar un dato que no esta en el árbol, por ejemplo el número 7, gráficamente lo que sucedería es lo que se observa en la figura 13 y la función retorna un -4.

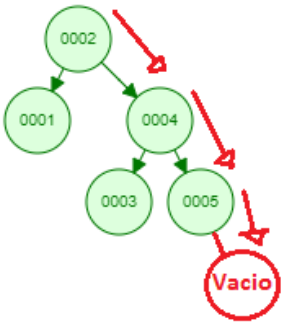

Figure 13: Elemento buscado no esta en el árbol.

7 Obtener el valor mínimo y valor máximo del arreglo

De igual manera que en los casos anteriores, es suficiente con llamar a las funciones enviándole los parámetros correspondientes (punteros inicializados).

```
1  #include "functions.h"
2  #include<iostream>
3  #include<algorithm>
4
5  int main()
6  {
7      float arr[]={1,2,3,4,5};
8      int size = (int)sizeof(arr)/sizeof(float);
9
10     Node *root = NULL;
11     Node *min= NULL;
12     Node *max= NULL;
13
14     avl create(size,arr,&root);
15     avl node add(root,6, &root);
16     avl node remove(root,6, &root);
17     avl search(root,5, &searched);
18     avl min get(root, &min);
19     avl max get(root, &max);
20
21
22 }
```

Figure 14: Uso de las funciones avl_min_get y avl_max_get.

7.1 Funciones avl_min_get y avl_max_get.

Estas funciones se encargan de recorrer todos los nodos hasta llegar al último mayor y ultimo menor. Por ejemplo, si se quiere obtener el valor mínimo la función va a empezar a recorrer los nodos hacia la izquierda comenzando desde la raíz. La condición de paro, para poder salir del bucle es cuando se encuentra un nodo vacío, se sabe que si se llegó a un nodo vacío es porque el nodo previo (que no está vacío) era el menor elemento de todos. En su defecto, ocurre lo mismo para obtener el valor máximo.

```
int avl max get(Node* root, Node** max node)
{
    Node* current=root;
    while (current->right != NULL)
    {
        current= current->right;
    }

    *max node=current;
    return 0;
}


int avl min get(Node* root, Node** min node)
{
    Node* current=root;
    while (current->left != NULL)
    {
        current= current->left;
    }

    *min node=current;
    return 0;
}
```

Figure 15: Funciones avl_min_get y avl_max_get.

Gráficamente (figura 16), el ejemplo se puede entender mejor. Para obtener el valor máximo hay que desplazarse hacia la derecha hasta el final, para obtener el valor mínimo hay que desplazarse hacia lo izquierda hasta el final.

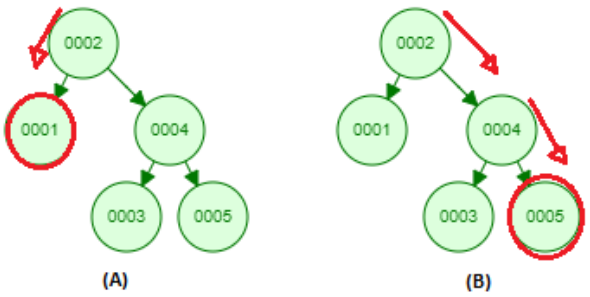

Figure 16: (A) La función obtiene el valor mínimo. (B) La función obtiene el valor máximo.

8 Imprimir los elementos del árbol

Se llama a la función avl_print enviándole como parámetro de entrada un puntero del nodo raíz del árbol, ver Figura 17. Esta función imprime el árbol avl completo, recorriendo e imprimiendo los nodos en preorden; esto significa que primeramente imprime la raíz del

árbol, luego recorre e imprime todo el subárbol izquierdo y posteriormente imprime el subárbol derecho.

Como se observa en la implementación de la función, ésta es recursiva, por lo que este método previamente describo se ejecuta para cada nodo, utilizando como raíz el nodo actual. Entendiendo este método, se puede dibujar el árbol completo fácilmente a partir de la impresión.

```
#include "functions.h"
#include<iostream>
#include<algorithm>

int main()
{
    float arr[]={1,2,3,4,5};
    int size = (int)sizeof(arr)/sizeof(float);

    Node *root = NULL;
    Node *min= NULL;
    Node *max= NULL;

    avl create(size,arr,&root);
    avl node add(root,6, &root);
    avl node remove(root,6, &root);
    avl search(root,5, &searched);
    avl min get(root, &min);
    avl max get(root, &max);
    avl print(root);

}
```

Figure 17: Uso de la función avl_print.

9 Consideraciones temporales

Cuando se hace uso de la función node_add puede ser muy ventajoso conocer características algunas de ella, como por ejemplo que tanto tiempo dura añadiendo los nodos al árbol, la importancia de tener conocimiento -o al menos una noción- radica en que no se sabe de que tamaño sera la lista de datos que algún usuario desee ingresar.

Como respuesta a esto, se corrieron pruebas para conocer de manera experimental esta característica llamada complejidad temporal, a continuación se muestran como los resultados de estas pruebas arrojan aproximaciones del tipo:

$$T(n) = O(log(n)) \tag{1}$$

9.1 Prueba 1

Se realizaron 100 pruebas distintas para distintos tamaños de estructuras, en este caso el tamaño de estas estructuras varía entre 100 y 20 000.

En color azul se aprecia la curva experimental, en color naranja se observar la curva de mejor ajuste.

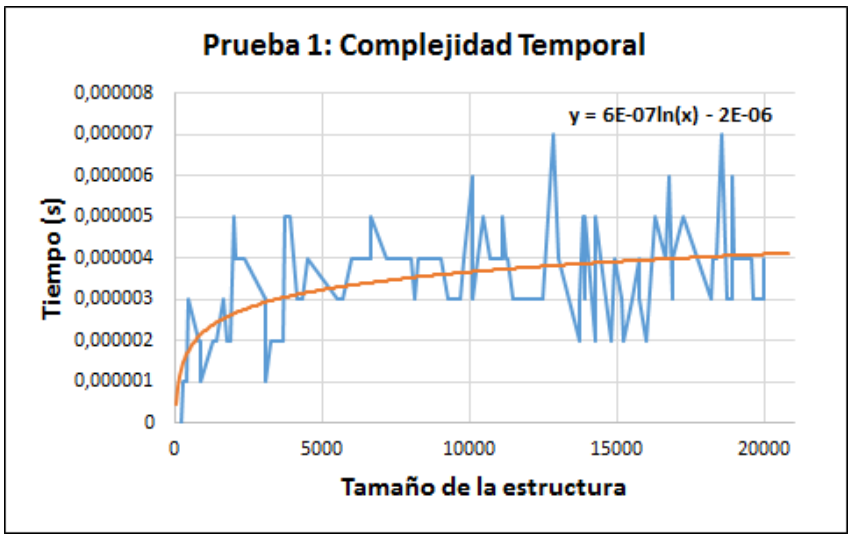

Figure 18: Gráfico Prueba 1.

9.2 Prueba 2

Se realizaron 100 pruebas distintas para distintos tamaños de estructuras, en este caso el tamaño de estas estructuras varía entre 100 y 100 000.

En color azul se aprecia la curva experimental, en color naranja se observar la curva de mejor ajuste.

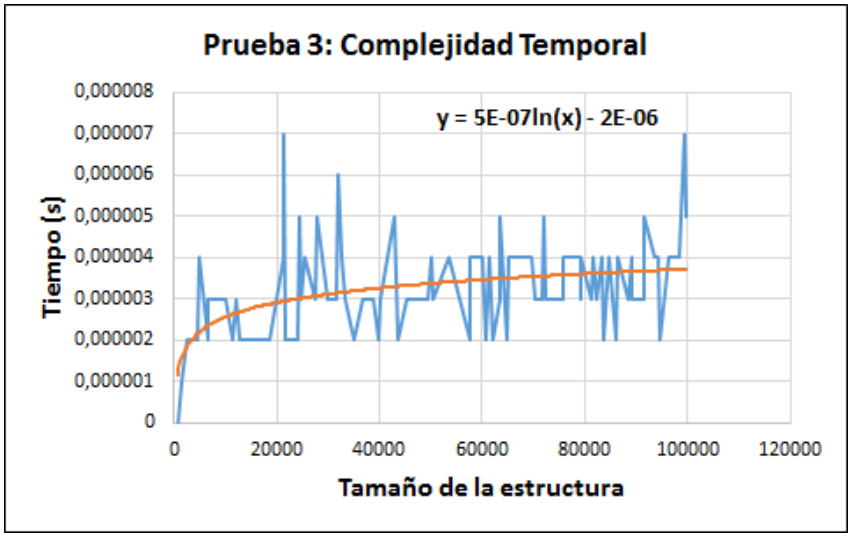

Figure 19: Gráfico Prueba 3.

9.3 Prueba 3

Se realizaron 400 pruebas distintas para distintos tamaños de estructuras, en este caso el tamaño de estas estructuras varía entre 100 y 100 000.

En color azul se aprecia la curva experimental, en color naranja se observar la curva de mejor ajuste.

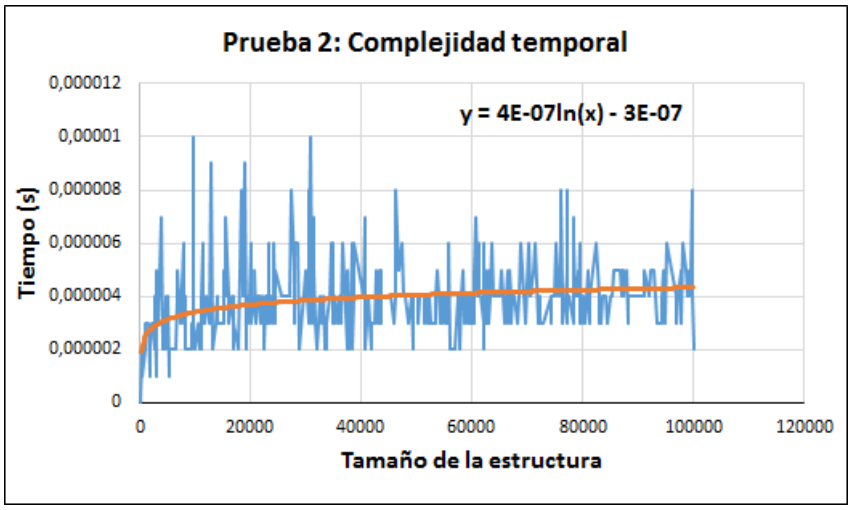

Figure 20: Gráfico Prueba 2.