CENTRO UNIVERSITARIO DE CIENCIAS EXÁCTAS E INGENIERÍAS

Arboles de Búsqueda AVL

Tarea 13

Aldo Alexandro Vargas Meza 213495653

05/05/2017

Actividad de Aprendizaje 13. El Árbol AVL, implementación dinámica

Problema:

Reutilice el programa resultante de la actividad 13. Agregue y/o modifique los métodos necesarios para que el árbol sea implementado como AVL.

La actividad busca la implementación de un árbol de búsqueda AVL con valores enteros, estos deben de estar en un rango de 0 – 65535 y se deben de mostrar conforme se inserten, después mostrar los diferentes órdenes que aplican a el árbol binario.

El nuevo programa reutiliza los códigos anteriores y le añade algunas funciones para la inserción del nodo, cumpliendo con los requerimientos del árbol AVL.

Se modificó el rango de números a generar, para que puedan ser encontrados en los aleatorios y sean borrados eficientemente en la simulación.

Función Main

La función main crea un objeto del tipo árbol AVL, al igual que variables de apoyo. En su ejecución, se solicita un numero al usuario para generar números aleatorios, después se insertan por medio de un ciclo for para luego ser ordenados. Al final, se imprimen los stats del árbol, como lo son altura y numero de nodos.

Al final, se borran números aleatorios y se reimprimen los stats asegurando el funcionamiento.

Clase Nodo

Para el manejo de los datos por medio de punteros, fue necesaria la entidad nodo, que consta del tipo de dato, y dos direcciones hacia otros elementos, que se manejaran como hojas. Siguiendo esta lógica, se programó una clase Nodo, que contiene ambas direcciones hacía la izquierda y derecha en forma de puntero y un dato almacenado. Esta entidad conformará cada uno de los datos de la lista ligada.

Además de dos constructores, la entidad Nodo entre sus métodos contiene getters y setters para los atributos y una función especial de impresión.

Cabe destacar que el árbol inserta de manera muy diferente a los arboles binarios, llevando a cabo rotaciones y rotaciones dobles. De esta manera el árbol se equilibra cuando lleva a cabo cada inserción.

Clase Árbol

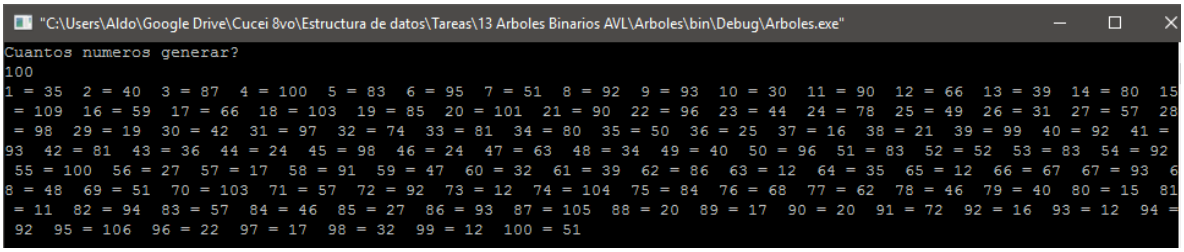
La clase árbol es una entidad que se podría pensar como una lista con diferentes niveles los cuales hacen que la implementación de este sea relativamente sencilla debido a su misma estructura de decisión cierto – falso.

Contiene funciones parecidas a la lista, y se maneja enteramente con un nodo principal, que es compuesto por un puntero del tipo Nodo dentro de los atributos privados.

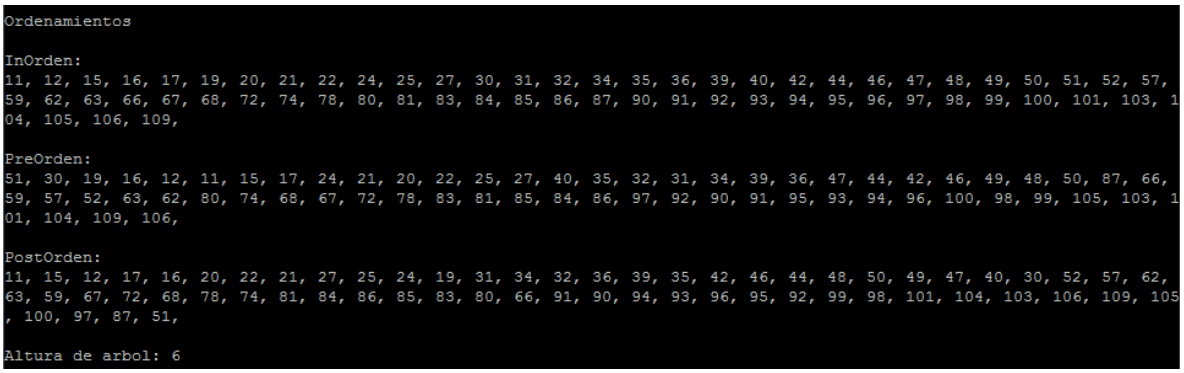
El árbol AVL a diferencia del binario también cuenta con las funciones de rotación, que identifican cada nodo para saber si es necesario llevar a cabo una rotación hacia un lado u otro.

Funcionamiento

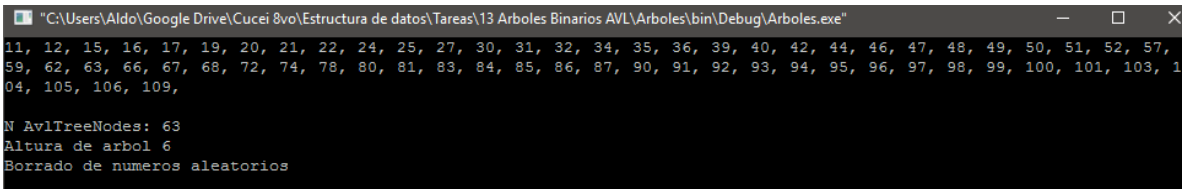
Primero se generar 100 números aleatorios



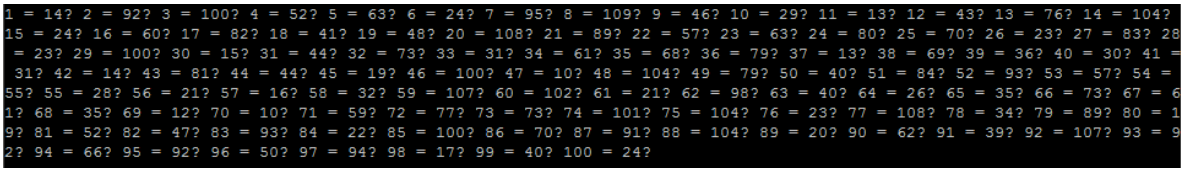
Después se les aplica ordenamiento



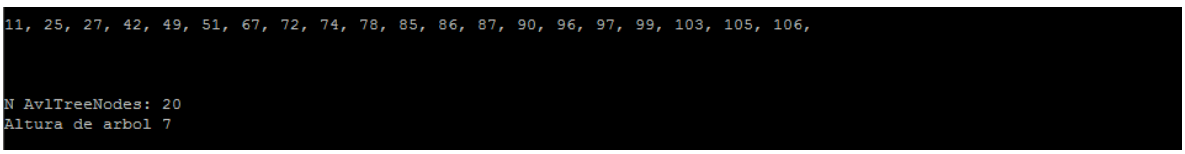
A los datos, se les aplica una función con números aleatorios a borrar, por lo tanto la lista inicial es



Los números a buscar para ser borrados son



Después de el borrado, los datos resultantes



Codigos

```
#include <iostream>

#include <time.h>

#include <stdlib.h>


using namespace std;


class AvlTreeNode {

    public:

        AvlTreeNode(const int dat, AvlTreeNode *pad=nullptr, AvlTreeNode *izq=nullptr,
AvlTreeNode *der=nullptr) :

            data(dat), father(pad), left(izq), right(der), FE(0) {}


        int data;

        int FE;


        AvlTreeNode *left;

        AvlTreeNode *right;

        AvlTreeNode *father;

        friend class AvlTree;

    };


class AvlTree {

    private:


        enum {left, right};

        int contador;

        int altura;


        AvlTreeNode *raiz;

        AvlTreeNode *actual;


        void Equilibrar(AvlTreeNode *AvlTreeNode, int, bool);
```

```
    void RSI(AvlTreeNode* AvlTreeNode);
    void RSD(AvlTreeNode* AvlTreeNode);
    void RDI(AvlTreeNode* AvlTreeNode);
    void RDD(AvlTreeNode* AvlTreeNode);

    void Podar(AvlTreeNode* &);
    void auxContador(AvlTreeNode*);
    void auxAltura(AvlTreeNode*, int);

public:

    AvlTree() : raiz(nullptr), actual(nullptr) {}
    ~AvlTree() {
        Podar(raiz);
        }

    void avlInsert(const int dat);
    void avlDelete(const int dat);
    bool Buscar(const int dat);
    bool isEmpty(AvlTreeNode *r) {
        return r==nullptr;
        }
    bool EsHoja(AvlTreeNode *r) {
        return !r->right && !r->left;
        }

    const int NumeroAvlTreeNodes();
    const int AlturaArbol();

    int Altura(const int dat);
    int &ValorActual() {
        return actual->data;
        }
    void Raiz() {
```

```
            actual = raiz;
            }

        void InOrden(void (*func)(int&, int), AvlTreeNode *AvlTreeNode=nullptr, bool
r=true);
        void PreOrden(void (*func)(int&, int), AvlTreeNode *AvlTreeNode=nullptr, bool
r=true);
        void PostOrden(void (*func)(int&, int), AvlTreeNode *AvlTreeNode=nullptr, bool
r=true);

    };


void AvlTree::Podar(AvlTreeNode* &AvlTreeNode) {
    if(AvlTreeNode) {
        Podar(AvlTreeNode->left);
        Podar(AvlTreeNode->right);
        delete AvlTreeNode;
        AvlTreeNode = nullptr;
        }
    }

void AvlTree::avlInsert(const int dat) {
    AvlTreeNode *father = nullptr;

    actual = raiz;

    while(!isEmpty(actual) && dat != actual->data) {
        father = actual;
        if(dat > actual->data) actual = actual->right;
        else if(dat < actual->data) actual = actual->left;
        }

    if(!isEmpty(actual)) return;
```

```
    if(isEmpty(father)) raiz = new AvlTreeNode(dat);
    else if(dat < father->data) {
        father->left = new AvlTreeNode(dat, father);
        Equilibrar(father, left, true);
        }

    else if(dat > father->data) {
        father->right = new AvlTreeNode(dat, father);
        Equilibrar(father, right, true);
        }
    }



void AvlTree::Equilibrar(AvlTreeNode *AvlTreeNode, int rama, bool nuevo) {
    bool salir = false;

    while(AvlTreeNode && !salir) {
        if(nuevo)
            if(rama == left) AvlTreeNode->FE--;
            else                  AvlTreeNode->FE++;
        else if(rama == left) AvlTreeNode->FE++;
        else                  AvlTreeNode->FE--;
        if(AvlTreeNode->FE == 0) salir = true;



        else if(AvlTreeNode->FE == -2) {
            if(AvlTreeNode->left->FE == 1) RDD(AvlTreeNode);
            else RSD(AvlTreeNode);
            salir = true;
            }
        else if(AvlTreeNode->FE == 2) {
            if(AvlTreeNode->right->FE == -1) RDI(AvlTreeNode);
            else RSI(AvlTreeNode);
            salir = true;
```

```
            }
        if(AvlTreeNode->father)
            if(AvlTreeNode->father->right == AvlTreeNode) rama = right;
            else rama = left;
        AvlTreeNode = AvlTreeNode->father;
        }
    }

void AvlTree::RDD(AvlTreeNode* nodo) {

    AvlTreeNode *father = nodo->father;
    AvlTreeNode *P = nodo;
    AvlTreeNode *Q = P->left;
    AvlTreeNode *R = Q->right;
    AvlTreeNode *B = R->left;
    AvlTreeNode *C = R->right;

    if(father)
        if(father->right == nodo) father->right = R;
        else father->left = R;
    else raiz = R;



    Q->right = B;
    P->left = C;
    R->left = Q;
    R->right = P;



    R->father = father;
    P->father = Q->father = R;
    if(B) B->father = Q;
    if(C) C->father = P;
```

```
    switch(R->FE) {
        case -1:
            Q->FE = 0;
            P->FE = 1;
            break;
        case 0:
            Q->FE = 0;
            P->FE = 0;
            break;
        case 1:
            Q->FE = -1;
            P->FE = 0;
            break;
        }
    R->FE = 0;
    }


void AvlTree::RDI(AvlTreeNode* nodo) {

    AvlTreeNode *father = nodo->father;
    AvlTreeNode *P = nodo;
    AvlTreeNode *Q = P->right;
    AvlTreeNode *R = Q->left;
    AvlTreeNode *B = R->left;
    AvlTreeNode *C = R->right;

    if(father)
        if(father->right == nodo) father->right = R;
        else father->left = R;
    else raiz = R;
```

```
    P->right = B;
    Q->left = C;
    R->left = P;
    R->right = Q;

    R->father = father;
    P->father = Q->father = R;
    if(B) B->father = P;
    if(C) C->father = Q;

    switch(R->FE) {
        case -1:
            P->FE = 0;
            Q->FE = 1;
            break;
        case 0:
            P->FE = 0;
            Q->FE = 0;
            break;
        case 1:
            P->FE = -1;
            Q->FE = 0;
            break;
        }
    R->FE = 0;
    }



void AvlTree::RSD(AvlTreeNode* nodo) {

    AvlTreeNode *father = nodo->father;
    AvlTreeNode *P = nodo;
    AvlTreeNode *Q = P->left;
```

```
    AvlTreeNode *B = Q->right;

    if(father)
        if(father->right == P) father->right = Q;
        else father->left = Q;
    else raiz = Q;


    P->left = B;
    Q->right = P;

    P->father = Q;
    if(B) B->father = P;
    Q->father = father;


    P->FE = 0;
    Q->FE = 0;
    }


void AvlTree::RSI(AvlTreeNode* nodo) {

    AvlTreeNode *father = nodo->father;
    AvlTreeNode *P = nodo;
    AvlTreeNode *Q = P->right;
    AvlTreeNode *B = Q->left;

    if(father)
        if(father->right == P) father->right = Q;
        else father->left = Q;
    else raiz = Q;
```

```
    P->right = B;
    Q->left = P;


    P->father = Q;
    if(B) B->father = P;
    Q->father = father;

    P->FE = 0;
    Q->FE = 0;
    }


void AvlTree::avlDelete(const int dat) {
    AvlTreeNode *father = nullptr;
    AvlTreeNode *AvlTreeNode;
    int aux;

    actual = raiz;

    while(!isEmpty(actual)) {
        if(dat == actual->data) {
            if(EsHoja(actual)) {
                if(father)

                    if(father->right == actual) father->right = nullptr;
                    else if(father->left == actual) father->left = nullptr;
                delete actual;
                actual = nullptr;

                if((father->right == actual && father->FE == 1) ||
                        (father->left == actual && father->FE == -1)) {
                    father->FE = 0;
                    actual = father;
```

```
            father = actual->father;
            }
        if(father)
            if(father->right == actual) Equilibrar(father, right, false);
            else                        Equilibrar(father, left, false);
        return;
        }
    else {

        father = actual;
        if(actual->right) {
            AvlTreeNode = actual->right;
            while(AvlTreeNode->left) {
                father = AvlTreeNode;
                AvlTreeNode = AvlTreeNode->left;
                }
            }

        else {
            AvlTreeNode = actual->left;
            while(AvlTreeNode->right) {
                father = AvlTreeNode;
                AvlTreeNode = AvlTreeNode->right;
                }
            }



        aux = actual->data;
        actual->data = AvlTreeNode->data;
        AvlTreeNode->data = aux;
        actual = AvlTreeNode;
        }
    }
else {
```

```
            father = actual;
            if(dat > actual->data) actual = actual->right;
            else if(dat < actual->data) actual = actual->left;
            }
        }
    }


void AvlTree::InOrden(void (*func)(int&, int), AvlTreeNode *AvlTreeNode, bool r) {
    if(r) AvlTreeNode = raiz;
    if(AvlTreeNode->left) InOrden(func, AvlTreeNode->left, false);
    func(AvlTreeNode->data, AvlTreeNode->FE);
    if(AvlTreeNode->right) InOrden(func, AvlTreeNode->right, false);
    }


void AvlTree::PreOrden(void (*func)(int&, int), AvlTreeNode *AvlTreeNode, bool r) {
    if(r) AvlTreeNode = raiz;
    func(AvlTreeNode->data, AvlTreeNode->FE);
    if(AvlTreeNode->left) PreOrden(func, AvlTreeNode->left, false);
    if(AvlTreeNode->right) PreOrden(func, AvlTreeNode->right, false);
    }


void AvlTree::PostOrden(void (*func)(int&, int), AvlTreeNode *AvlTreeNode, bool r) {
    if(r) AvlTreeNode = raiz;
    if(AvlTreeNode->left) PostOrden(func, AvlTreeNode->left, false);
    if(AvlTreeNode->right) PostOrden(func, AvlTreeNode->right, false);
    func(AvlTreeNode->data, AvlTreeNode->FE);
    }
```

```
bool AvlTree::Buscar(const int dat) {
    actual = raiz;

    while(!isEmpty(actual)) {
        if(dat == actual->data) return true;
        else if(dat > actual->data) actual = actual->right;
        else if(dat < actual->data) actual = actual->left;
        }
    return false;
    }

int AvlTree::Altura(const int dat) {
    int altura = 0;
    actual = raiz;

    while(!isEmpty(actual)) {
        if(dat == actual->data) return altura;
        else {
            altura++;
            if(dat > actual->data) actual = actual->right;
            else if(dat < actual->data) actual = actual->left;
            }
        }
    return -1;
    }

const int AvlTree::NumeroAvlTreeNodes() {
    contador = 0;

    auxContador(raiz);
```

```
    return contador;
    }


void AvlTree::auxContador(AvlTreeNode *AvlTreeNode) {
    contador++;

    if(AvlTreeNode->left) auxContador(AvlTreeNode->left);
    if(AvlTreeNode->right)   auxContador(AvlTreeNode->right);
    }


const int AvlTree::AlturaArbol() {
    altura = 0;

    auxAltura(raiz, 0);
    return altura;
    }


void AvlTree::auxAltura(AvlTreeNode *AvlTreeNode, int a) {

    if(AvlTreeNode->left) auxAltura(AvlTreeNode->left, a+1);
    if(AvlTreeNode->right)   auxAltura(AvlTreeNode->right, a+1);

    if(EsHoja(AvlTreeNode) && a > altura) altura = a;
    }


void Mostrar(int &d, int FE) {
    cout << d <<", ";
    }

int main() {
```

```
AvlTree avlInt;
srand (time(NULL));
int delnum = 0, num = 0;



cout<<"Cuantos numeros generar?"<<endl;
cin >> num;

for(int i=1; i<=num; i++) {
    int temp = rand() % 100 + 10;
    avlInt.avlInsert(temp);
    cout<<i<<" = "<<temp<<"  ";
    }
cout<<endl<<endl;

cout <<"Ordenamientos " <<endl<<endl;

cout << "InOrden: "<<endl;
avlInt.InOrden(Mostrar);
cout << endl<<endl;
cout << "PreOrden: "<<endl;
avlInt.PreOrden(Mostrar);
cout << endl<<endl;
cout << "PostOrden: "<<endl;
avlInt.PostOrden(Mostrar);
cout << endl<<endl;



cout << "Altura de arbol: " << avlInt.AlturaArbol() << endl;
system("pause");
system("cls");

avlInt.InOrden(Mostrar);
```

```
cout <<endl<<endl<< "N AvlTreeNodes: " << avlInt.NumeroAvlTreeNodes() << endl;
cout << "Altura de arbol " << avlInt.AlturaArbol() << endl;


cout <<"Borrado de numeros aleatorios " <<endl<<endl;


for(int i=1; i<=num; i++) {
    int temp = rand() % 100 + 10;;
    avlInt.avlDelete(temp);
    cout<<i<<" = "<<temp<<"? ";
    }
cout<<endl<<endl;

avlInt.InOrden(Mostrar);
cout<<endl<<endl;
cout <<endl<<endl<< "N AvlTreeNodes: " << avlInt.NumeroAvlTreeNodes() << endl;
cout << "Altura de arbol " << avlInt.AlturaArbol() << endl;


cin.get();
return 0;
}
```