PECL3

Estructuras De Datos

Pedro Barquín Ayuso

Wasim El Hallak Díez

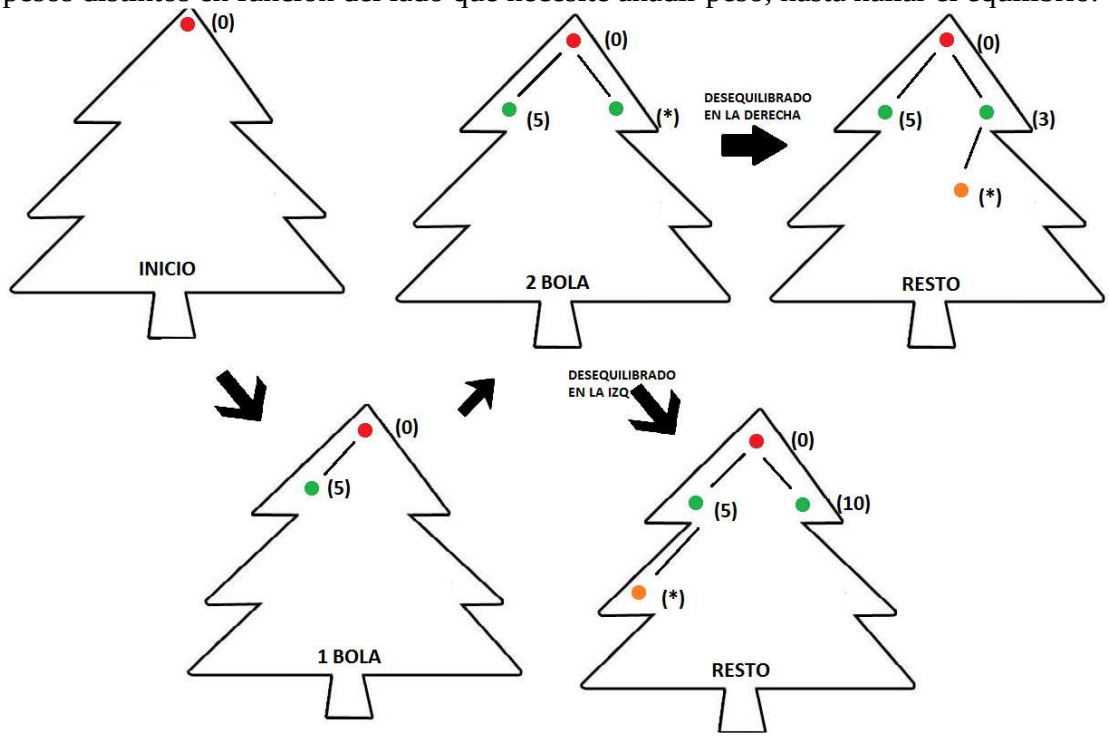
3º GIC

Curso 2015/2016

ÍNDICE

Descripción del Programa y requisitos

Tal y como se especifica en el enunciado de la práctica, se trata de crear un programa que simula la colocación de un árbol de navidad al que se le van añadiendo distintas bolas con un peso aleatorio comprendido entre ciertos pesos, en dicho árbol lo que se busca es que el contenido de la rama derecha y el de la izquierda tengan un mismo peso (o diferencias del 2% del total), para que el árbol este totalmente equilibrado y no se ladee. El funcionamiento se puede ver en la imagen, primero se añade la raíz que posee peso 0 y después la primera bola en uno de los lados, y a partir de ahí se van añadiendo pesos distintos en función del lado que necesite añadir peso, hasta hallar el equilibrio.



Todo este proceso ha de hacerse de manera automática y aleatoria cumpliendo ciertas restricciones.

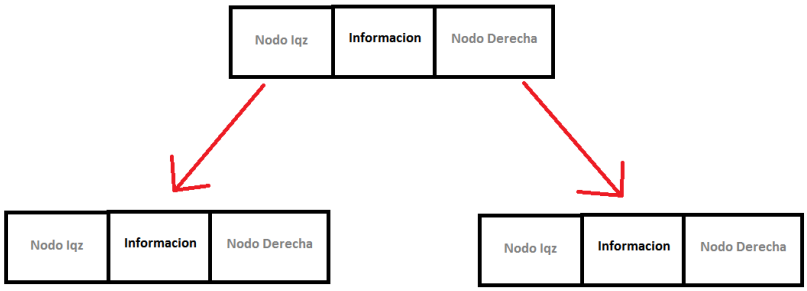
- Se modelará el árbol como un Árbol Binario de Búsqueda.
- Los valores de los pesos que colgarán de las ramas del árbol se generarán de forma aleatoria tomando, en cada turno, uno de los siguientes valores: 1, 2, 3, 5, 7 y 10 kg.
- La diferencia de pesos no ha de superar el 2% del peso total del árbol.

Descripción TAD’s implementados

Para este programa contamos con TAD Nodo y ArbolABB ya que son con los que modelaremos el funcionamiento de este árbol de búsqueda en el cual ha de haber equilibrio.
Los nodos son las “bolas” que se colocan dentro del árbol y que están conectadas entre ellas siguiendo el esquema de los árboles, para permitirnos seguir un esquema sin perder la información. Cada nodo cuenta con el número que es el valor que se le asigna automáticamente para colocarlo dentro del árbol ya que en función delnúmero que se le asigne ira colocado en un lugar u otro, Nodo izq. y derecho que es para la referencia a los nodos que cuelgan de él y el peso que es primordial para el programa ya que es con lo que tenemos que jugar para equilibrarlo. EL valor y el peso no están relacionado y no han de confundirse entre ellos.

```
classNodo{
private:
        int dato;
        Nodo *izquierdo;
        Nodo *derecho;
        int peso;
        friendclassArbolABB;

public:
        Nodo(constintdat, intps, Nodo *izq = NULL, Nodo *der = NULL) : dato(dat),
izquierdo(izq), derecho(der) ,peso(ps) {}
};
typedefNodo *pNodo;
```



El ArbolABB es el “Árbol” donde van colocadas las “bolas” cumple la función de un árbol de búsqueda binario y contiene todas las funciones para su correcto funcionamiento pero adema se le han añadido algunas extra explicadas en los siguientes apartados para que realice el “equilibrado” de manera correcta. A parte de los métodos de insertado y tiene guardada la información del nodo que actúa como raíz para no perder única la referencia y un nodo aux con el cual no iremos moviendo para poder realizar las operaciones, también tiene algunas variables como contador, altura, etc. Variables que usaremos para mostrar datos de interés.

```
classArbolABB {
public:
        // Constructor y destructor básicos:
        ArbolABB() {raiz = NULL; pesoTotal = 0; pesoIzquierda = 0; pesoDerecha = 0; numNodos =
1; altura = 0;}
        ~ArbolABB() { podar(raiz); }
        // Insertar en árbol ordenado
        void insertar(constint dat, int ps);
        // Borrar un elemento del árbol:
        void borrar(constint dat);
        // Función de búsqueda:
        bool buscar(constint dat);
        // Comprobar si está un nodo vacío:
```

```
        bool esVacio(Nodo *r) { returnr==NULL; }
        // Comprobar si el nodo es una hoja:
        bool esHoja(Nodo *r) { return !r->derecho && !r->izquierdo; }
        // Devuelve número de nodos
        int getNumNodos();
        // Devuelve altura del árbol
        int getAltura();
        // Calcula la altura del árbol
        void calcularAlturaArbol(Nodo *nodo, int a);
        // Devuelve la altura de un nodo:
        int alturaNodo(constint dat);
        // Devuelve el valor del peso total del arbol
        int getPesoTotal();
        // Devuelve el valor que pesa la izq
        int getPesoIzquierda();
        // Devuelve el valor que pesa la der
        int getPesoDerecha();
        // Devolver el peso de un nodo concreto
        int pesoEspecifico(constint dat);
        // Comprobar que se cumple el margen de peso
        bool estaEquilibrado();
        // Función que muestra un nivel del árbol
        void mostrarNivel(Nodo *raid, int level, int i);
        // Función que retorna la raíz para realizar algunos cálculos
        Nodo *getRaiz();
        // Poda: borrar todos los nodos a partir de uno, incluído
        void podar(Nodo *nodo);
        // Inicializa valores del árbol
        void inicializar();
        // Funciones que muestran los nodos en diferentes órdenes
        void inOrden(void(*func)(int&), Nodo *nodo = NULL, bool r = true);
        void preOrden(void(*func)(int&), Nodo *nodo = NULL, bool r = true);
        void postOrden(void(*func)(int&), Nodo *nodo = NULL, bool r = true);

private:
        Nodo *raiz;
        int pesoTotal;
        int pesoDerecha;
        int pesoIzquierda;
        int numNodos;
        int altura;
};
```

Estos TAD's son auxiliares para poder realizar las funciones de eliminación de un nodo "Bola" del árbol y poder reordenarlo correctamente, ambos son similares a los utilizados en la 2 practica salvo por los valores y alguna que otra función especial.

```
classNodoLista                                  //Declaracion del nodo que forman las listas
{
private:                                                //Variables y punteros a informacion
        int dato;
        int peso;
        NodoLista *siguiente;                           //puntero al siguiente
        friendclassLista;                               //clase amiga

public:
        NodoLista(intd, intp, NodoLista *sig = NULL)    //constructor con la informacion y *sig
se pasa como inicial NULL y luego va cambiando
        {
                dato = d;
                peso = p;
                siguiente = sig;
        }
        int getPeso() {return peso;}
        NodoLista *getSiguiente() {return siguiente;}
};
typedefNodoLista *pNodoLista;           //un *pNodoLista es un tipo definido de NodoLista


classLista
{
public:
        Lista() : primero(NULL), ultimo(NULL), longitud(0) {}   //constructor
```

```
        ~Lista();                                               //destructor
        void inicializar();
        pNodoLista getNodoLista(int d);
        pNodoLista getPrimero();
        pNodoLista getUltimo();
        void insertarNodoLista(int d, int p);
        void insertarFinalNodoLista(int d, int p);
        void borrarNodoLista(int d);
        bool esListaVacia();
        void mostrarLista();
        int getLongitud();

private:
        pNodoLista primero, ultimo;     //definimos dos pNodoLista primero y ultimo
        int longitud;                   //variable para la longitud
};
```

Definición de operaciones de los TAD's

Árboles

En este método se realiza la inserción de un nodo al que se le ha pasado el valor con el cual se va a insertar que según el número irá en una zona u otro y además se pasa el peso con el cual se va a colgar, peso que se ha de añadir al total y al peso parcial en función del lado donde se inserte, para la inserción contamos con varios casos, el principal es que sea el primer nodo entonces se asigna como raíz y luego los posteriores nodos se van calcando a un lado u otro en función de su valor respecto a la raíz y al resto de nodos de su rama.

```
voidArbolABB::insertar(constintdat, intps)
{
        Nodo *padre = NULL;
        Nodo *aux = raiz;

        // Buscar el int en el árbol, manteniendo un puntero al nodo padre
        while (!esVacio(aux) &&dat != aux->dato) {
                padre = aux;
                if (dat> aux->dato) aux = aux->derecho;
                elseif (dat< aux->dato) aux = aux->izquierdo;
        }

        // Si se ha encontrado el elemento, regresar sin insertar
        if (!esVacio(aux)) return;
        // Si padre es NULL, entonces el árbol estaba vacío, el nuevo nodo será
        // el nodo raiz
        if (esVacio(padre)) {
                raiz = newNodo(dat, ps);
                numNodos = 1;
                pesoTotal += ps;
        }
        // Si el int es menor que el que contiene el nodo padre, lo insertamos
        // en la rama izquierda
        elseif (dat< padre->dato) {
                if (dat<raiz->dato)
                {
                        pesoIzquierda += ps;
                }
                else {
                        pesoDerecha += ps;
                }
                padre->izquierdo = newNodo(dat, ps);
                numNodos++;
                pesoTotal += ps;
```

```
        }
        // Si el int es mayor que el que contiene el nodo padre, lo insertamos
        // en la rama derecha
        elseif (dat> padre->dato) {
                if (dat<raiz->dato)
                {
                        pesoIzquierda += ps;
                }
                else {
                        pesoDerecha += ps;
                }
                padre->derecho = newNodo(dat, ps);
                numNodos++;
                pesoTotal += ps;
        }
}
```

Elimina un elemento concreto del árbol para ello hemos de pasar el valor del nodo que queremos eliminar, antes de eliminar dicho valor hemos de restar el peso de ese nodo al peso total del árbol para poder mantener correctamente la funcionalidad. Edemas después de eliminar reorganiza la colocación del árbol.

```
voidArbolABB::borrar(constintdat)
{
        Nodo *padre = NULL;
        Nodo *nodo;
        Nodo *actual = raiz;

        int aux;

        pesoTotal -= pesoEspecifico(dat);//restar el peso específico de la bola, en caso de no
estar resta 0

        // Mientras sea posible que el valor esté en el árbol
        while (!esVacio(actual)) {
                if (dat == actual->dato) { // Si el valor está en el nodo actual
                        if (esHoja(actual)) { // Y si además es un nodo hoja: lo borramos
                                if (padre) { // Si tiene padre (no es el nodo raíz)
                                        // Anulamos el puntero que le hace referencia
                                        if (padre->derecho == actual) padre->derecho = NULL;
                                        elseif (padre->izquierdo == actual) padre->izquierdo =
NULL;
                                }
                                delete actual; // Borrar el nodo
                                numNodos--;
                                return;
                        }
                        else { // Si el valor está en el nodo actual, pero no es hoja
                                // Buscar nodo
                                padre = actual;
                                // Buscar nodo más izquierdo de rama derecha
                                if (actual->derecho) {
                                        nodo = actual->derecho;
                                        while (nodo->izquierdo) {
                                                padre = nodo;
                                                nodo = nodo->izquierdo;
                                        }
                                }
                                // O buscar nodo más derecho de rama izquierda
                                else {
                                        nodo = actual->izquierdo;
                                        while (nodo->derecho) {
                                                padre = nodo;
                                                nodo = nodo->derecho;
                                        }
                                }
                                // Intercambiar valores de nodo a borrar o nodo encontrado
                                // y continuar, cerrando el bucle. El nodo encontrado no tiene
                                // porqué ser un nodo hoja, cerrando el bucle nos aseguramos
                                // de que sólo se eliminan nodos hoja.
                                aux = actual->dato;
                                actual->dato = nodo->dato;
                                nodo->dato = aux;
                                actual = nodo;
```

```
                        }
                }
                else { // Todavía no hemos encontrado el valor, seguir buscándolo
                        padre = actual;
                        if (dat> actual->dato) actual = actual->derecho;
                        elseif (dat< actual->dato) actual = actual->izquierdo;
                }
        }
}
```

Función que recorre el árbol en búsqueda de un valor si esta lo confirma y en caso contrario lo desmiente.

```
boolArbolABB::buscar(constintdat)
{
        Nodo *aux = raiz;

        // Todavía puede aparecer, ya que quedan nodos por mirar
        while (!esVacio(aux)) {
                if (dat == aux->dato) returntrue; // int encontrado
                elseif (dat> aux->dato) aux = aux->derecho; // Seguir
                elseif (dat< aux->dato) aux = aux->izquierdo;
        }
        returnfalse; // No está en árbol
}
```

Función que nos proporciona el valor del número de nodos en el árbol en el momento de la consulta.

```
intArbolABB::getNumNodos()
{
        return numNodos;
}
```

Función que nos retorna el valor de la altura del árbol en el momento de su consulta el cual se calcula con la función `calcularAlturaArbol()`.

```
intArbolABB::getAltura()
{
return altura;
}
```

Procedimiento que calcula la altura máxima del árbol, este cálculo lo hace recorriendo el árbol de manera recursiva pasando por todas las ramas y guardando solo el valor de la maxima para su posterior consulta.

```
voidArbolABB::calcularAlturaArbol(Nodo *nodo, inta)
{
        // Recorrido postorden
        if (nodo->izquierdo) calcularAlturaArbol(nodo->izquierdo, a+1);
        if (nodo->derecho)   calcularAlturaArbol(nodo->derecho, a+1);
        // Proceso, si es un nodo hoja, y su altura es mayor que la actual del
        // árbol, actualizamos la altura actual del árbol
        if (esHoja(nodo) &&a> altura) altura = a;
}
```

Función que nos da el valor de la altura a la que se encuentra un nodo concreto dentro del árbol, para ello va recorriendo todo el árbol desde la raíz hasta toparse con el nodo en cuestión del cual extraemos la información.

```
intArbolABB::alturaNodo(constintdat)
{
        int alt = 0;
        Nodo *aux = raiz;

        // Todavía puede aparecer, ya que quedan nodos por mirar
        while (!esVacio(aux)) {
                if (dat == aux->dato) return alt; // int encontrado
                else {
                        alt++; // Incrementamos la altura, seguimos buscando
                        if (dat> aux->dato) aux = aux->derecho;
                        elseif (dat< aux->dato) aux = aux->izquierdo;
                }
        }
        return -1; // No está en árbol
```

```
}
```

Función que retorna el valor del peso total del árbol.

```
intArbolABB::getPesoTotal()
{
        return pesoTotal;
}
```

Función que retorna el valor del peso izquierdo del árbol.

```
intArbolABB::getPesoIzquierda()
{
        return pesoIzquierda;
}
```

Función que retorna el valor del peso derecho del árbol.

```
intArbolABB::getPesoDerecha()
{
        return pesoDerecha;
}
```

Función auxiliar que nos da el peso de un nodo concreto para ello hemos recibir el valor del nodo y recorrer el árbol entro buscando, como los valores no pueden estar repetidos solo va a haber uno y por lo tanto solo nos va a dar el peso de ese.

```
intArbolABB::pesoEspecifico(constintdat)
{
        Nodo *aux = raiz;

        // Todavía puede aparecer, ya que quedan nodos por mirar
        while (!esVacio(aux)) {
                if (dat == aux->dato) return aux->peso; // int encontrado devolver el peso
                elseif (dat> aux->dato) aux = aux->derecho; // Seguir
                elseif (dat< aux->dato) aux = aux->izquierdo;
        }
        return 0; // No está en árbol así que peso 0
}
```

Función en la cual se verifican los pesos de las dos ramas los cuales son variables del árbol cuales aumentan o disminuyen al insertar o eliminar y con esos pesos se calcula si la diferencia entre ambas es menor al 2% del peso total de ser así podemos dar por terminada la ejecución.

```
boolArbolABB::estaEquilibrado()
{
        //variable aux para los cálculos
        double resultado = 0.0;

        //calculamos el % de variación entre los dos lados
        if (pesoIzquierda > pesoDerecha){
resultado = pesoIzquierda - pesoDerecha;
        }
        else{
resultado = pesoDerecha - pesoIzquierda;
        }

        cout<<"La diferencia de pesos es de "<< resultado << endl;
        resultado = resultado * 100;
        resultado = resultado / pesoTotal;
        cout<<"Lo que supone un desequilibrio del "<< resultado <<"%"<< endl;
        cout<< endl;

        //devolver si ha terminado o no
        if (resultado<2.0){
                returntrue;
        }
        else {
                returnfalse;
        }
}
```

Función que muestra los nodos de un nivel concreto la cual es usada para mostrar el árbol completo ya que se la va llamando pasado todos los niveles para que muestre el árbol completo, se le ha añadido un espaciado para intentar que la imagen mostrada sea lo más similar a la simulación del árbol que se ha estudiado en clase.

```
voidArbolABB::mostrarNivel(Nodo *raid, intlevel, inti)
{
int esp;

switch (i)
        {   // Ajustes de espacios para mostrar según el nivel
                case 0: esp = 49; break;
                case 1: esp = 31; break;
                case 2: esp = 17; break;
                case 3: esp = 8; break;
                case 4: esp = 3; break;
                default: esp = 0; break;
        }

        if (raid==NULL) //Si es nulo mostrar una equis para evitar fallos lo que quiere decir
que ahí no hay nada ni en sus inferiores y termina
        {
                cout<< setw (esp+3) <<"X";
        }
        else {
                if (level == 0) {       //si ya hemos llegado donde queremos mostramos
                        cout<< setw (esp) <<raid->dato <<"("<<raid->peso <<")";
                }
                else {          //si aún no hemos llegado entonces volvemos a llamar pasando
otro nivel inferior
                        mostrarNivel(raid->izquierdo, level - 1, i);
                        mostrarNivel(raid->derecho, level - 1, i);
                }
        }
}
```

Esta función se encarga de retornar el nodo raíz para poder realizar ciertos cálculos en otras funciones.

```
Nodo *ArbolABB::getRaiz()
{
        return raiz;
}
```

Función para eliminar el contenido completo del árbol de forma recursiva.

```
voidArbolABB::podar(Nodo *nodo)
{
//      // Algoritmo recursivo, recorrido en postorden
if (nodo) {
podar(nodo->izquierdo);
podar(nodo->derecho);
deletenodo;
nodo = NULL;
    }
}
```

Esta función se encarga de crear el nodo raíz para cada una de los nuevos árboles que se inicialicen durante la ejecución del programa.

```
voidArbolABB::inicializar() {
raiz = NULL; pesoTotal = 0; pesoIzquierda = 0; pesoDerecha = 0; numNodos = 1; altura = 0;
}
```

Funciones que recorren el árbol de una forma concreta para poder mostrar la información de los distintos modos posibles.

```
voidArbolABB::inOrden(void(*func)(int&), Nodo *nodo, boolr)
{
        if (r) nodo = raiz;
        if (nodo->izquierdo) inOrden(func, nodo->izquierdo, false);
        func(nodo->dato);
        if (nodo->derecho) inOrden(func, nodo->derecho, false);
}voidArbolABB::preOrden(void(*func)(int&), Nodo *nodo, boolr)
{
        if (r) nodo = raiz;
        func(nodo->dato);
        if (nodo->izquierdo) preOrden(func, nodo->izquierdo, false);
        if (nodo->derecho) preOrden(func, nodo->derecho, false);
}voidArbolABB::postOrden(void(*func)(int&), Nodo *nodo, boolr)
{
        if (r) nodo = raiz;
        if (nodo->izquierdo) postOrden(func, nodo->izquierdo, false);
        if (nodo->derecho) postOrden(func, nodo->derecho, false);
        func(nodo->dato);
}
```

Listas

Destructor de una lista para empezar una nueva

```
Lista::~Lista()                         //destructor
{
        pNodoLista aux;
        while (primero)
        {
                aux = primero;
                primero = primero->siguiente;
                delete aux;
        }
}
```

Inicializa los valores de una lista para crear una nueva

```
voidLista::inicializar()
{
primero = NULL;
ultimo = NULL;
longitud = 0;
}
```

Devuelve el nodo de la lista que contiene el dato pasado como parámetro

```
pNodoListaLista::getNodoLista(intd)
{
        pNodoLista aux;
        aux = primero;

        while (aux->dato!=d&& (aux->siguiente)!=NULL)
        {
                aux = aux->siguiente;
        }
return aux;
}
```

Devuelve el primer nodo de la lista

```
pNodoListaLista::getPrimero()
{
return primero;
}
```

Devuelve el último nodo de la lista

```
pNodoListaLista::getUltimo()
{
return ultimo;
}
```

Inserta un nuevo nodo por la izquierda en la lista con el dato y el peso que contienen pasados como parámetro

```
voidLista::insertarNodoLista(intd, intp)                //inserta nodoLista por la izquierda
{
        pNodoLista aux;
        aux = newNodoLista(d, p);
        aux->siguiente = primero;
        primero = aux;
        if (ultimo==NULL) ultimo = aux;
        longitud++;
}
```

Inserta un nuevo nodo por la derecha en la lista con el dato y el peso que contienen pasados como parámetro

```
voidLista::insertarFinalNodoLista(intd, intp)                  //inserta nodoLista por la
derecha
{
        pNodoLista aux;
        aux = newNodoLista(d, p);
        if (ultimo!=NULL) ultimo->siguiente = aux;
        ultimo = aux;
        if (primero==NULL) primero = aux;
        longitud++;
}
```

Borra el nodo de la lista que contenga el dato pasado como parámetro y depende de si lo contiene o no o de dónde se sitúe el nodo dentro de la lista procederá de una manera u otra

```
voidLista::borrarNodoLista(intd)                //elimina uno de los nodos
{
        pNodoLista anterior;
        pNodoLista aux;
        anterior = primero;
        aux = primero;

if(primero!=NULL)
    {
while (aux->dato!=d&& (aux->siguiente)!=NULL)
        {
aux = aux->siguiente;
if (aux==anterior->siguiente->siguiente)
anterior = anterior->siguiente;
        }
if(aux->dato==d)
        {
if (aux==primero)
            {
primero = aux->siguiente;
            }
else
            {
anterior->siguiente = aux->siguiente;
if (aux==ultimo)
                {
ultimo = anterior;
                }
            }
aux->siguiente = NULL;
if (longitud!=0) longitud--;
        }
    }
}
```

Devuelve true si la lista está vacía

```
boolLista::esListaVacia()                               //comprueba si esta vacia la lista
{
        return primero==NULL;
}
```

Muestra los datos que contiene la lista mientras que no esté vacía

```
voidLista::mostrarLista()                               //muestra todo el contenido
{
        pNodoLista aux;
        aux = primero;
if(aux == NULL) cout <<" Lista Vacia"<< endl;
        while (aux)
        {
                cout<<"Dato: "<< aux->dato;
                cout<<"  Peso: "<< aux->peso << endl;
                aux = aux->siguiente;
        }
        cout<< endl;
}
```

Devuelve la longitud de la lista

```
intLista::getLongitud()                         // Devuelve la longitud
{
        return longitud;
}
```

Explicación funcionamiento y métodos más importantes

El programa principal sin ningún tipo de mejora funciona de la siguiente manera, primero se crea un nodo raíz con peso 0 que es del cual partimos como referencia, acto seguido llamamos al método equilibrar () que es donde se realiza la creación de las bolas con un peso aleatorio y se realiza la inserción en uno de los lados hasta que se complete el requisito de pesos que pide el programa y una vez completado se muestra la información del resultado y una aproximación del árbol.

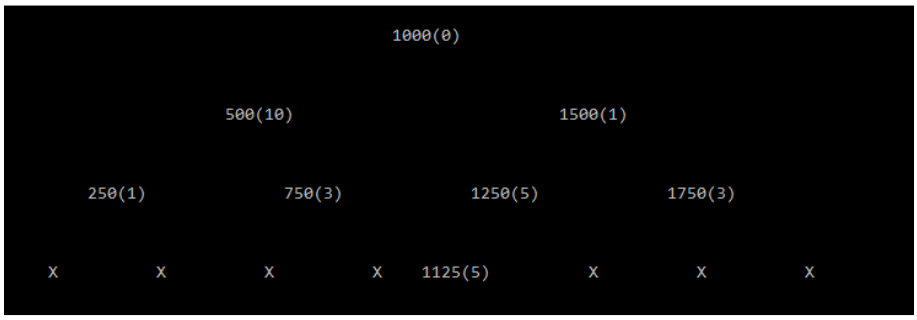
Esta función nos retorna un valor aleatorio que será el peso con el que se creara la bola para su posterior inserción en el árbol, este método es pseudo aleatorio.

```
int pesoAleatorio() {
        int a = rand() % 6;
        switch (a)
        {
                case 0: cout <<"la bola con peso 1"<< endl; return  1; break;
                case 1: cout <<"la bola con peso 2"<< endl; return  2; break;
                case 2: cout <<"la bola con peso 3"<< endl; return  3; break;
                case 3: cout <<"la bola con peso 5"<< endl; return  5; break;
                case 4: cout <<"la bola con peso 7"<< endl; return  7; break;
                default: cout <<"la bola con peso 10"<< endl; return    10; break;
        }
}
```

Estas funciones calculan el valor de la bola a insertar en el árbol dicho calcula se realiza con la información pasada que hace referencia al número de veces que se ha realizado la inserción en un lado para poder así calcular el número que le corresponde.

```
int numeroDerecha(intND, intPD) {
        int aux = (int)(intermedio / ND);
        return (aux*PD);
}
int numeroIzquierda(intNI, intPI) {
        int aux = (int)(intermedio / NI);
        return (aux*PI);
}
```

En este método lo que se realiza es un recorrido por capas llamando a mostrar una altura concreta del árbol y con ayuda se los saltos de línea se puede mostrar de una manera más eficiente la colocación del árbol. Se han añadido algunas funciones de control para evitar fallos como el querer mostrar algo estando vacío.




```
void mostrarFormaArbol() {

if (arbolInt.esVacio(arbolInt.getRaiz())) {

cout<<" Arbol vacio"<< endl;
cout<< endl;
cout<< endl;
    }
else {

Nodo *aux = arbolInt.getRaiz();//nodo auxiliar para recorrer
cout<< endl;
for (int i = 0; i < arbolInt.getAltura()+1; i++)//Recorremos todas las capas pidiendo que se
muestren
        {
arbolInt.mostrarNivel(aux, i, i);
cout<< endl << endl << endl << endl;
        }
    }
}
```

Este método empieza eliminando el árbol, la lista, e inicializando todo lo anterior si es que lo hubiera para evitar que se solapen entre ellos, tiene un bucle do while ya que siempre se tiene que ejecutar una vez y la condición de salida es que la diferencia entre ramas no supere el 2% del peso total tal y como se pide en el enunciado. Dentro del bucle primero comprobamos si el peso de la derecha es menor al de la izquierda de ser así hemos de añadir la bola en el lado derecho gracias a las funciones que nos dan su peso y su número además de ello también añadimos la información a la lista para poder realizar otras operaciones como la eliminación de un nodo concreto, después de insertarlo en el árbol en su posición correspondiente aumentamos los valores con los que realizamos el cálculo del número de bola y repetimos el proceso. En caso de ser mayor el peso de la derecha se realiza la inserción en el izquierdo.

El booleano hayLista nos ayuda a ejecutar unas instrucciones o no dependiendo de si se ha ejecutado anteriormente la solicitud de borrar un nodo con la función reequilibrar(), si se ha ejecutado, no creará una lista nueva, creará el árbol desde la lista ya existente. El árbol se terminará de crear cuando esté equilibrado, aun si quedan elementos en la lista por incluir al árbol, y si se termina la lista y sigue sin estar equilibrado se rellenará el árbol con datos aleatorios.

```
Insertamos en el lado izquierdo la bola con peso 1
Actualizamos informacion:
Peso total del arbol 1
Peso total de la derecha 0
Peso total de la izquierda 1

La diferencia de pesos es de 1
Lo que supone un desequilibrio del 100%

Insertamos en el lado derecho la bola con peso 3
Actualizamos informacion:
Peso total del arbol 4
Peso total de la derecha 3
Peso total de la izquierda 1

La diferencia de pesos es de 2
Lo que supone un desequilibrio del 50%
```

```
Insertamos en el lado derecho la bola con peso 5
Actualizamos informacion:
Peso total del arbol 21
Peso total de la derecha 10
Peso total de la izquierda 11

La diferencia de pesos es de 1
Lo que supone un desequilibrio del 4.7619%

Insertamos en el lado derecho la bola con peso 1
Actualizamos informacion:
Peso total del arbol 22
Peso total de la derecha 11
Peso total de la izquierda 11

La diferencia de pesos es de 0
Lo que supone un desequilibrio del 0%
```

```
void equilibrar() {
        //variables para los cálculos
        int dato;
        int peso;
        int NI = 2;     //Nivel
        int ND = 2;
        int DI = 1;     //Divisiones o veces a realizar
        int DD = 1;
        int PI = 1;     //Posición
        int PD = 3;
        int CD = 1;     //Contador veces
        int CI = 1;
        pNodoLista aux;
        aux = pesos.getPrimero();

// Podamos el árbol para crear uno nuevo
arbolInt.podar(arbolInt.getRaiz());

// Borra la lista
if(!hayLista) pesos.~Lista();

// Inicializa los valores del árbol
arbolInt.inicializar();

// Inicializa los valores de la lista
if(!hayLista) pesos.inicializar();

        //iniciamos el nodo raíz con el valor intermedio y peso 0 para que no cuente en el total
        arbolInt.insertar(intermedio, 0);

        //realizamos las operaciones hasta que el arbol este equilibrado
        do
        {
                if (arbolInt.getPesoDerecha()<arbolInt.getPesoIzquierda())      //Insertamos en
la derecha por tener un peso inferior
                {
                        cout<<"Insertamos en el lado derecho ";
```

```
                        dato = numeroDerecha(ND,PD);
                        if(!hayLista) peso = pesoAleatorio();
                        if(hayLista)
                {
if(aux)
                {
peso = aux->getPeso();
cout<<"la bola con peso "<< peso << endl;
aux = aux->getSiguiente();
                } else hayLista = false;
                }
                        arbolInt.insertar(dato, peso);
                        if(!hayLista) pesos.insertarFinalNodoLista(dato, peso);
                        cout<<"Actualizamos informacion: "<< endl;
                        cout<<"Peso total del arbol "<< arbolInt.getPesoTotal() << endl;
                        cout<<"Peso total de la derecha "<< arbolInt.getPesoDerecha() << endl;
                        cout<<"Peso total de la izquierda "<< arbolInt.getPesoIzquierda() <<
endl;
                        cout<< endl;
                        if (CD==DD)                     //Se igualan los valores contador y
divisiones y pasamos al próx nivel
                        {
                                ND = ND * 2;    //Pasamos de nivel
                                DD = DD * 2;    //Aumento el número de divisiones
                                PD = PD + 2;    //Aumento de la posisción de 2 en 2 para evitar
usar el número del padre
                                CD = 1;                 //Iniciamos contador
                        }
                        else {                          //Si el contador no es igual al número
de divisiones
                                PD = PD + 2;    //Aumento de la posiscion de 2 en 2 para evitar
usar el número del padre
                                CD++;                   //Aumento contador
                        }
                }
                else {
                        //Si no insertamos en la izquierda
                        cout<<"Insertamos en el lado izquierdo ";
                        dato = numeroIzquierda(NI,PI);
                        if(!hayLista) peso = pesoAleatorio();
                        if(hayLista)
                {
if(aux)
                {
peso = aux->getPeso();
cout<<"la bola con peso "<< peso << endl;
aux = aux->getSiguiente();
                } else hayLista = false;
                }
                        arbolInt.insertar(dato, peso);
                        if(!hayLista) pesos.insertarFinalNodoLista(dato, peso);
                        cout<<"Actualizamos informacion: "<< endl;
                        cout<<"Peso total del arbol "<< arbolInt.getPesoTotal() << endl;
                        cout<<"Peso total de la derecha "<< arbolInt.getPesoDerecha() << endl;
                        cout<<"Peso total de la izquierda "<< arbolInt.getPesoIzquierda() <<
endl;
                        cout<< endl;
                        if (CI == DI)           //Se igualan los valores contador y divisiones y
pasamos al próx nivel
                        {
                                NI = NI * 2;    //Pasamos de nivel
                                DI = DI * 2;    //Aumento el número de divisiones
                                PI = 1;                 //Inicializamos la posición
                                CI = 1;                 //Iniciamos contador
                        }
                        else {                          //Si el contador no es igual al número
de divisiones
                                PI = PI + 2;    //Aumento de la posisción de 2 en 2 para evitar
usar el número del padre
                                CI++;                   //Aumento contador
                        }
                }
        } while (!arbolInt.estaEquilibrado());//Repetir hasta que se cumpla
```

```
        arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);

        cout<< endl;
        cout<<"El arbol ha sido equilibrado"<< endl;
        cout<< endl;

        hayLista = false;
}
```

Este método lo utilizamos para mostrar la información más relevante del árbol toda de golpe en una sola consulta, para así evitar tener que realizarlas una a una.

```
Informacion del arbol:
 Altura del arbol 1
 Peso total del arbol 2
 Peso total de la derecha 1
 Peso total de la izquierda 1
 N nodos: 3
```

```
void infoArbol() {

cout<<" Informacion del arbol: "<< endl;
cout<<"  Altura del arbol "<< arbolInt.getAltura() << endl;
cout<<"  Peso total del arbol "<< arbolInt.getPesoTotal() << endl;
cout<<"  Peso total de la derecha "<< arbolInt.getPesoDerecha() << endl;
cout<<"  Peso total de la izquierda "<< arbolInt.getPesoIzquierda() << endl;
cout<<"  N nodos: "<< arbolInt.getNumNodos() << endl;
cout<< endl;
cout<< endl;
}
```

Con esta llamada mostramos la información auxiliar o menos relevante sobre el árbol, además mostramos la información en los distintos órdenes.

```
Diferentes formas de representacion de la colocacion:

 InOrden: 250,500,750,1000,1054,1125,1250,1375,1500,1625,1750,1875,
 PreOrden: 1000,500,250,750,1500,1250,1125,1054,1375,1750,1625,1875,
 PostOrden: 250,750,500,1054,1125,1375,1250,1625,1875,1750,1500,1000,
```

```
void infoX() {

if (arbolInt.esVacio(arbolInt.getRaiz())) {

cout<<" Arbol vacio"<< endl;
cout<< endl;
cout<< endl;
    }
else {

cout<<" Diferentes formas de representacion de la colocacion: "<< endl << endl;
cout<<"  InOrden: ";
arbolInt.inOrden(mostrar);
cout<< endl;
cout<<"  PreOrden: ";
arbolInt.preOrden(mostrar);
cout<< endl;
cout<<"  PostOrden: ";
arbolInt.postOrden(mostrar);
cout<< endl;
    }
}
```

Extra eliminar una bola concreta

Con esta función extra añadida se borra un nodo del árbol a elección del usuario y también se borra de la lista para poder llamar a la función equilibrar y que cree el nuevo árbol a partir de la lista por medio de la modificación de hayLista a true para que equilibrar realice las operaciones convenientes.

Las funciones mostrarFormaArbol y mostrarLista se invocan para comprobar que el nodo elegido ya no está en el árbol y ver la lista antes y después de borrar dicho nodo respectivamente.

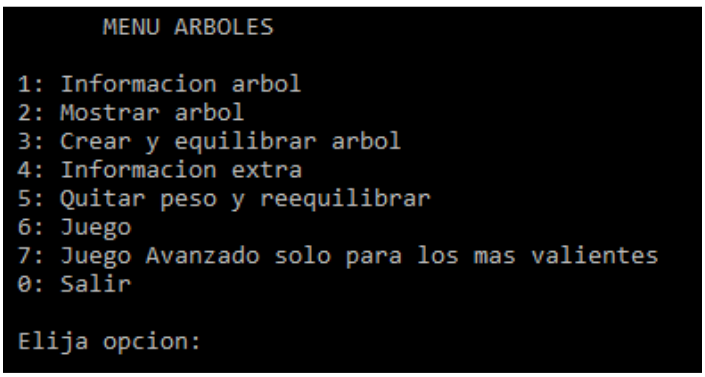
```
// Quita la bola con el valor especificado y reequilibra
void reequilibrar() {
int dato;

cout<<" El peso de que dato desea quitar?: ";
cin>> dato;
cout<< endl;
if (arbolInt.buscar(dato) && dato!=intermedio)
    {
arbolInt.borrar(dato);
mostrarFormaArbol();
pesos.mostrarLista();
pesos.borrarNodoLista(dato);
pesos.mostrarLista();
hayLista = true;
equilibrar();
    }
else
    {
cout<<" Ese dato no se puede borrar"<< endl;
return;
    }

}
```

Extramenú:

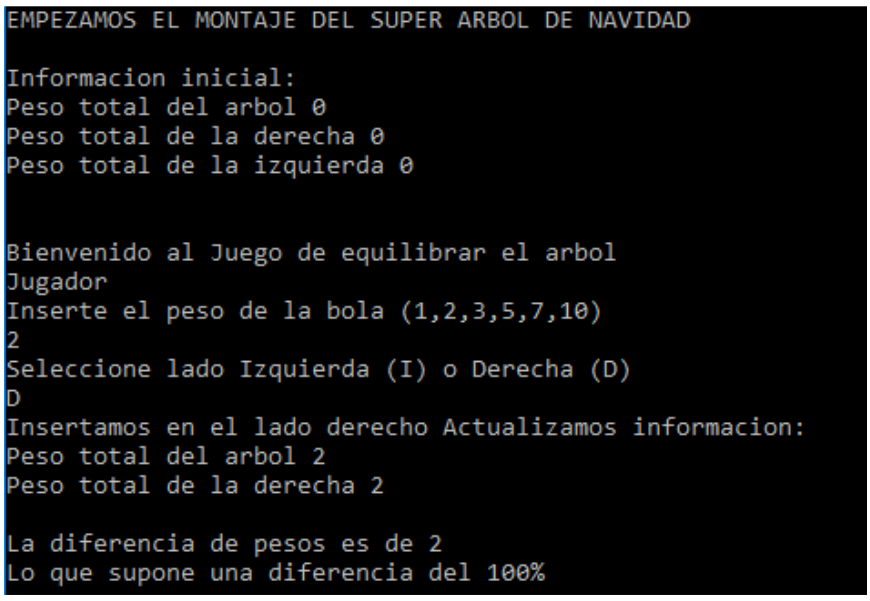
Este extra es para poder realizar las acciones de manera sencilla y pudiendo elegir entre varias opciones, si necesidad de tener que re-escribir el código, algunas de estas opciones son mostrar información realizar el equilibrado jugar a un juego o salir, podemos seleccionar cualquiera de las que aparecen en la lista para explorar varias opciones.



```
while (valorMenu!=0)
        {
        cout<< endl;
                cout<<"       MENU ARBOLES"<< endl;
                cout<< endl;
                cout<<" 1: Informacion arbol"<< endl;
                cout<<" 2: Mostrar arbol"<< endl;
                cout<<" 3: Crear y equilibrar arbol"<< endl;
                cout<<" 4: Informacion extra"<< endl;
                cout<<" 5: Quitar peso y reequilibrar"<< endl;
                cout<<" 6: Juego"<< endl;
                cout<<" 7: Juego Avanzado solo para los mas valientes"<< endl;
                cout<<" 0: Salir"<< endl;
                cout<< endl;
                cout<<" Elija opcion: ";
                cin>> valorMenu;
                cout<< endl;
                switch (valorMenu)
                {
                case 0: cout <<" Bye!"<<endl<< endl; break;
case 1: infoArbol(); break;
case 2: mostrarFormaArbol(); break;
case 3: equilibrar(); break;
case 4: infoX(); break;
case 5: reequilibrar(); break;
case 6: equilibrarJuego(); break;
case 7: equilibrarJuegoAvanzado(); break;
default: cout <<" Opcion erronea"<< endl << endl; break;
                }
        }
```

Extra juego fácil:

Uno de los extras de este programa es el de la simulación de un juego en el que la maquina va insertando números y el usuario alternándose para ver quién es el que inserta el último número que equilibra el árbol.
El funcionamiento es igual al del método equilibrar pero en este caso uno de los pasos se realiza pidiendo el peso al usuario (solo datos validos) y pidiendo en qué lado se quiere insertar (solo datos valido), y después de insertarlo se comprueba si está equilibrado y de no ser así se pasa el mando a la máquina que lo hace de forma automática en este nivel la maquina actúa de manera aleatoria.



```
Marquina
Insertamos en el lado izquierdo la bola con peso 10
Actualizamos informacion:
Peso total del arbol 12
Peso total de la izquierda 10


Arbol actualizado

                1000(0)
        500(10)         1500(2)

La diferencia de pesos es de 8
Lo que supone una diferencia del 66.6667%
```

```
El arbol ha sido equilibrado se termino el juego
HA GANADO EL JUGADOR

Informacion del arbol equilibrado:
Altura de arbol 2
Peso total del arbol 22
Peso total de la derecha 11
Peso total de la izquierda 11
N nodos: 6
Altura de arbol 2
Peso total del arbol 22


Diferentes formas de representacion de la colocacion:
InOrden: 250,500,1000,1250,1500,1750,
PreOrden: 1000,500,250,1500,1250,1750,
PostOrden: 250,500,1250,1750,1500,1000,

Mostramos el arbol con su forma final

                        1000(0)
                500(10)                 1500(2)
        250(1)    -     1250(7)         1750(2)
```

```
void equilibrarJuego() {
	//variables para los calculos
	int NI = 2;   //Nivel
	int ND = 2;
	int DI = 1;   //Divisiones o veces a realizar
	int DD = 1;
	int PI = 1;   //Posicion
	int PD = 3;
	int CD = 1;   //Contador veces
	int CI = 1;
	int contadorTurno = 0;
	char direccion;
	int pesoBola = 0;
	bool pesado = true;
	bool direccionado = true;

// Podamos el árbol para crear uno nuevo
arbolInt.podar(arbolInt.getRaiz());
// Inicializa los valores del árbol
arbolInt.inicializar();
```

```
	//iniciamos el nodo raiz con el valor intermedio y peso 0 para que no
cuente en el total
	arbolInt.insertar(intermedio, 0);
	//Mensajes de informacion del juego
	cout<<"Bienvenido al Juego de equilibrar el arbol"<< endl;
	cout<<  endl;
	cout<<"Reglas:"<< endl;
	cout<<"1- El juego consiste ir añadiendo pesos hasta en equilibrar el
arbol de tal manera"<< endl;
cout<<"   que el oponente no pueda terminarlo y tu si."<<endl;
cout<<"2- Solo se pueden insertar caracteres validos en caso contrario volvera a
solicitar"<< endl;
cout<<"3- La maquina solo genera pesos aleatorios en este nivel"<< endl;
cout<<"4- Para ganar el equilibrio tiene que ser exacto"<< endl;
cout<<"5- El jugador que coloque la ultima bola sera el ganador"<< endl;
cout<<  endl;

	//realizamos las operaciones hasta que el arbol este equilibrado
	do
	{
		if (contadorTurno % 2 == 0)
		{
			cout<<"Jugador"<< endl;
			do
			{
				pesado = true;
				cout<<"Inserte el peso de la bola (1,2,3,5,7,10)"<<
endl;
				cin>> pesoBola;
				if (pesoBola==1 || pesoBola==2 || pesoBola==3 ||
pesoBola==5 || pesoBola==7 || pesoBola==10)
				{
					pesado = false;
				}
			} while (pesado);
			do
			{
				direccionado = true;
				cout<<"Seleccione lado Izquierda (I) o Derecha (D)"<<
endl;
				cin>> direccion;
				if (direccion == 'D' || direccion == 'd' || direccion
== 'I' || direccion == 'i')
				{
					direccionado = false;
				}
			} while (direccionado);
			if (direccion=='D' || direccion == 'd')
			{
				cout<<"Insertamos en el lado derecho ";
				cout<<"la bola con peso "<< pesoBola << endl;
				arbolInt.insertar(numeroDerecha(ND, PD), pesoBola);
				arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
				cout<<"Actualizamos informacion: "<< endl;
				cout<<"Peso total del arbol "<<
arbolInt.getPesoTotal() << endl;
				cout<<"Peso total de la derecha "<<
arbolInt.getPesoDerecha() << endl;
				cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
				cout<< endl;
```

```
                        if (CD == DD)              //Se igualan los valores
contador y divisiones y pasamos al prox nivel
                        {
                                ND = ND * 2;  //Pasamos de nivel
                                DD = DD * 2;  //Aumento el numero de divisiones
                                PD = PD + 2;  //Aumento de la posiscion de 2 en
dos para evitar usar el numero del padre
                                CD = 1;                    //Iniciamos contador
                        }
                        else {                     //Si el contador no es
igual al numero de divisiones
                                PD = PD + 2;  //Aumento de la posiscion de 2 en
dos para evitar usar el numero del padre
                                CD++;              //Aumento contador
                        }
                }
                else {
                        cout<<"Insertamos en el lado izquierdo ";
                        cout<<"la bola con peso "<< pesoBola << endl;
                        arbolInt.insertar(numeroIzquierda(NI, PI), pesoBola);
                        arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
                        cout<<"Actualizamos informacion: "<< endl;
                        cout<<"Peso total del arbol "<<
arbolInt.getPesoTotal() << endl;
                        cout<<"Peso total de la derecha "<<
arbolInt.getPesoDerecha() << endl;
                        cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
                        cout<< endl;
                        if (CI == DI)       //Se igualan los valores contador
y divisiones y pasamos al prox nivel
                        {
                                NI = NI * 2;  //Pasamos de nivel
                                DI = DI * 2;  //Aumento el numero de divisiones
                                PI = 1;                    //Inicializamos la
posicion
                                CI = 1;                    //Iniciamos contador
                        }
                        else {                     //Si el contador no es
igual al numero de divisiones
                                PI = PI + 2;  //Aumento de la posiscion de 2 en
dos para evitar usar el numero del padre
                                CI++;              //Aumento contador
                        }
                }
                contadorTurno++;
        }
        else {
                cout<<"Maquina"<< endl;
                if (arbolInt.getPesoDerecha()<arbolInt.getPesoIzquierda())
      //Insertamos en la derecha por tener un peso inferior
                {
                        cout<<"Insertamos en el lado derecho ";
                        arbolInt.insertar(numeroDerecha(ND, PD),
pesoAleatorio());
                        arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
                        cout<<"Actualizamos informacion: "<< endl;
                        cout<<"Peso total del arbol "<<
arbolInt.getPesoTotal() << endl;
                        cout<<"Peso total de la derecha "<<
arbolInt.getPesoDerecha() << endl;
```

```
                        cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
                        cout<< endl;
                        if (CD == DD)              //Se igualan los valores
contador y divisiones y pasamos al prox nivel
                        {
                                ND = ND * 2;  //Pasamos de nivel
                                DD = DD * 2;  //Aumento el numero de divisiones
                                PD = PD + 2;  //Aumento de la posiscion de 2 en
dos para evitar usar el numero del padre
                                CD = 1;                    //Iniciamos contador
                        }
                        else {                     //Si el contador no es
igual al numero de divisiones
                                PD = PD + 2;  //Aumento de la posiscion de 2 en
dos para evitar usar el numero del padre
                                CD++;              //Aumento contador
                        }
                }
                else {
                        //Si no insertamos en la izquierda
                        cout<<"Insertamos en el lado izquierdo ";
                        arbolInt.insertar(numeroIzquierda(NI, PI),
pesoAleatorio());
                        arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
                        cout<<"Actualizamos informacion: "<< endl;
                        cout<<"Peso total del arbol "<<
arbolInt.getPesoTotal() << endl;
                        cout<<"Peso total de la derecha "<<
arbolInt.getPesoDerecha() << endl;
                        cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
                        cout<< endl;
                        if (CI == DI)       //Se igualan los valores contador
y divisiones y pasamos al prox nivel
                        {
                                NI = NI * 2;  //Pasamos de nivel
                                DI = DI * 2;  //Aumento el numero de divisiones
                                PI = 1;                    //Inicializamos la
posicion
                                CI = 1;                    //Iniciamos contador
                        }
                        else {                     //Si el contador no es
igual al numero de divisiones
                                PI = PI + 2;  //Aumento de la posiscion de 2 en
dos para evitar usar el numero del padre
                                CI++;              //Aumento contador
                        }
                }
                cout<< endl;
                cout<<"Arbol actualizado"<< endl;
                mostrarFormaArbol();
                cout<< endl;
                contadorTurno++;
        }
    } while (arbolInt.getPesoDerecha()!=arbolInt.getPesoIzquierda());//Repetir
hasta que se cumpla
    cout<< endl;
    cout<<"El arbol ha sido equilibrado. Se termino el juego"<< endl;
    if (contadorTurno % 2 == 0) {
        cout<<"HA GANADO LA MAQUINA"<< endl;
    }else{
```

```
                cout<<"HA GANADO EL JUGADOR"<< endl;
        }
        cout<< endl;
}
```

Extra juego difícil:

El funcionamiento es igual al del otro juego pero en este caso se complica ya que la maquina no es aleatoria si no que elije meticulosamente donde y que peso ha de colocar para lograr ganar, la maquina nos da una primera oportunidad que en caso de no aprovechar luego lamentaremos ya que solo es posible ganar en el segundo intento o logrando que la diferencia entre pesos sea una concreta lo cual nos dará otra oportunidad la cual deberemos aprovechar.

```
void equilibrarJuegoAvanzado() {
        //variables para los calculos
        int NI = 2;     //Nivel
        int ND = 2;
        int DI = 1;     //Divisiones o veces a realizar
        int DD = 1;
        int PI = 1;     //Posicion
        int PD = 3;
        int CD = 1;     //Contador veces
        int CI = 1;
        int contadorTurno = 0;
        int diferenciaPesos = 0;
        int pesoMaquina = 0;
        char direccion;
        int pesoBola = 0;
        bool pesado = true;
        bool direccionado = true;
        bool nosDaVentaja = true;

// Podamos el árbol para crear uno nuevo
arbolInt.podar(arbolInt.getRaiz());
// Inicializa los valores del árbol
arbolInt.inicializar();
        //iniciamos el nodo raiz con el valor intermedio y peso 0 para que no cuente en el total
        arbolInt.insertar(intermedio, 0);
        //Mensajes de informacion del juego
        cout<<"Bienvenido al Juego de equilibrar el arbol"<< endl;
        cout<<  endl;
        cout<<"Reglas:"<< endl;
        cout<<"1- El juego consiste ir añadiendo pesos hasta en equilibrar el arbol de tal
manera"<< endl;
cout<<"   que el oponente no pueda terminarlo y tu si."<<endl;
cout<<"2- Solo se pueden insertar caracteres validos en caso contrario volvera a solicitar"<<
endl;
cout<<"3- La maquina seleciona los valores a conciencia en este nivel"<< endl;
cout<<"4- Para ganar el equilibrio tiene que ser exacto"<< endl;
cout<<"5- El jugador que coloque la ultima bola sera el ganador"<< endl;
cout<<"6- Suerte y que dios nos pille confesados"<< endl;
cout<<  endl;

        //realizamos las operaciones hasta que el arbol este equilibrado
        do
        {
                if (contadorTurno % 2 == 0)
                {
                        cout<<"Jugador"<< endl;
                        do
                        {
                                pesado = true;
                                cout<<"Inserte el peso de la bola (1,2,3,5,7,10)"<< endl;
                                cin>> pesoBola;
```

```
                                if (pesoBola==1 || pesoBola==2 || pesoBola==3 || pesoBola==5 ||
pesoBola==7 || pesoBola==10)
                                {
                                        pesado = false;
                                }
                        } while (pesado);
                        do
                        {
                                direccionado = true;
                                cout<<"Seleccione lado Izquierda (I) o Derecha (D)"<< endl;
                                cin>> direccion;
                                if (direccion == 'D' || direccion == 'd' || direccion == 'I' ||
direccion == 'i')
                                {
                                        direccionado = false;
                                }
                        } while (direccionado);
                        if (direccion=='D' || direccion == 'd')
                        {
                                cout<<"Insertamos en el lado derecho ";
                                cout<<"la bola con peso "<< pesoBola << endl;
                                arbolInt.insertar(numeroDerecha(ND, PD), pesoBola);
                                arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
                                cout<<"Actualizamos informacion: "<< endl;
                                cout<<"Peso total del arbol "<< arbolInt.getPesoTotal() << endl;
                                cout<<"Peso total de la derecha "<< arbolInt.getPesoDerecha() <<
endl;
                                cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
                                cout<< endl;
                                if (CD == DD)                   //Se igualan los valores
contador y divisiones y pasamos al prox nivel
                                {
                                        ND = ND * 2;    //Pasamos de nivel
                                        DD = DD * 2;    //Aumento el numero de divisiones
                                        PD = PD + 2;    //Aumento de la posiscion de 2 en dos
para evitar usar el numero del padre
                                        CD = 1;                 //Iniciamos contador
                                }
                                else {                          //Si el contador no es igual al
numero de divisiones
                                        PD = PD + 2;    //Aumento de la posiscion de 2 en dos
para evitar usar el numero del padre
                                        CD++;                   //Aumento contador
                                }
                        }
                        else {
                                cout<<"Insertamos en el lado izquierdo ";
                                cout<<"la bola con peso "<< pesoBola << endl;
                                arbolInt.insertar(numeroIzquierda(NI, PI), pesoBola);
                                arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
                                cout<<"Actualizamos informacion: "<< endl;
                                cout<<"Peso total del arbol "<< arbolInt.getPesoTotal() << endl;
                                cout<<"Peso total de la derecha "<< arbolInt.getPesoDerecha() <<
endl;
                                cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
                                cout<< endl;
                                if (CI == DI)           //Se igualan los valores contador y
divisiones y pasamos al prox nivel
                                {
                                        NI = NI * 2;    //Pasamos de nivel
                                        DI = DI * 2;    //Aumento el numero de divisiones
                                        PI = 1;                 //Inicializamos la posicion
                                        CI = 1;                 //Iniciamos contador
                                }
                                else {                          //Si el contador no es igual al
numero de divisiones
                                        PI = PI + 2;    //Aumento de la posiscion de 2 en dos
para evitar usar el numero del padre
                                        CI++;                   //Aumento contador
                                }
                        }
                        contadorTurno++;
```

```
                }
                else {
                        cout<<"Maquina"<< endl;
                        //Calculamos la diferencia de los pesos para saber que numero insertar
                        if  (arbolInt.getPesoDerecha()<arbolInt.getPesoIzquierda())
            {
diferenciaPesos=arbolInt.getPesoIzquierda()- arbolInt.getPesoDerecha();
            }else{
diferenciaPesos=arbolInt.getPesoDerecha()-arbolInt.getPesoIzquierda();
            }
                        if(nosDaVentaja){
cout<<"Te voy a dar ventaja"<< endl;
do{
pesoMaquina=pesoAleatorio();
                }while (pesoMaquina==diferenciaPesos);
nosDaVentaja=false;
                        }else {
                        //Obtenemos el valor del peso a insertar
switch (diferenciaPesos)
                {
case 1: pesoMaquina=1; break;
case 2: pesoMaquina=2; break;
case 3: pesoMaquina=3; break;
case 4: pesoMaquina=10; break;
case 5: pesoMaquina=5; break;
case 6: pesoMaquina=2; break;
case 7: pesoMaquina=7; break;
case 8: pesoMaquina=3; break;   //Solo se puede vencer llegando a la diferencia de 8
case 9: pesoMaquina=1; break;
case 10:pesoMaquina=10; break;
case 11:pesoMaquina=2; break;
default:pesoMaquina=1; break;
                }
                        }

                        if (arbolInt.getPesoDerecha()<arbolInt.getPesoIzquierda())
        //Insertamos en la derecha por tener un peso inferior
                        {
                                cout<<"Insertamos en el lado derecho ";
                                arbolInt.insertar(numeroDerecha(ND, PD), pesoMaquina);
                                arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
                                cout<<"Actualizamos informacion: "<< endl;
                                cout<<"Peso total del arbol "<< arbolInt.getPesoTotal() << endl;
                                cout<<"Peso total de la derecha "<< arbolInt.getPesoDerecha() <<
endl;
                                cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
                                cout<< endl;
                                if (CD == DD)                   //Se igualan los valores
contador y divisiones y pasamos al prox nivel
                                {
                                        ND = ND * 2;    //Pasamos de nivel
                                        DD = DD * 2;    //Aumento el numero de divisiones
                                        PD = PD + 2;    //Aumento de la posiscion de 2 en dos
para evitar usar el numero del padre
                                        CD = 1;                 //Iniciamos contador
                                }
                                else {                          //Si el contador no es igual al
numero de divisiones
                                        PD = PD + 2;    //Aumento de la posiscion de 2 en dos
para evitar usar el numero del padre
                                        CD++;                   //Aumento contador
                                }
                        }
                        else {
                                //Si no insertamos en la izquierda
                                cout<<"Insertamos en el lado izquierdo ";
                                arbolInt.insertar(numeroIzquierda(NI, PI), pesoMaquina);
                                arbolInt.calcularAlturaArbol(arbolInt.getRaiz(), 0);
                                cout<<"Actualizamos informacion: "<< endl;
                                cout<<"Peso total del arbol "<< arbolInt.getPesoTotal() << endl;
                                cout<<"Peso total de la derecha "<< arbolInt.getPesoDerecha() <<
endl;
```

```
                                cout<<"Peso total de la izquierda "<<
arbolInt.getPesoIzquierda() << endl;
                                cout<< endl;
                                if (CI == DI)           //Se igualan los valores contador y
divisiones y pasamos al prox nivel
                                {
                                        NI = NI * 2;    //Pasamos de nivel
                                        DI = DI * 2;    //Aumento el numero de divisiones
                                        PI = 1;                 //Inicializamos la posicion
                                        CI = 1;                 //Iniciamos contador
                                }
                                else {                          //Si el contador no es igual al
numero de divisiones
                                        PI = PI + 2;    //Aumento de la posiscion de 2 en dos
para evitar usar el numero del padre
                                        CI++;                   //Aumento contador
                                }
                        }
                        cout<< endl;
                        cout<<"Arbol actualizado"<< endl;
                        mostrarFormaArbol();
                        cout<< endl;
                        contadorTurno++;
                }
        } while (arbolInt.getPesoDerecha()!=arbolInt.getPesoIzquierda());//Repetir hasta que se
cumpla
        cout<< endl;
        cout<<"El arbol ha sido equilibrado. Se termino el juego"<< endl;
        if (contadorTurno % 2 == 0) {
                cout<<"HA GANADO LA MAQUINA"<< endl;
        }else{
                cout<<"HA GANADO EL JUGADOR"<< endl;
        }
        cout<< endl;
}
```

Problemas encontrados y solución adoptada

En esta práctica el principal problema ha sido el entender correctamente el funcionamiento del código inicial proporcionado par a partir de ahí continuar con el programa, hasta lograr que funcionara correctamente, otro problema fue el cómo generar automáticamente el número de la bola que se va a colgar para que respetara el esquema del árbol binario evitando así que todos los números se agolparan en una lado u el otro, se optó por que los números partirán de una base y se vallan dividiendo entre dos para logar que se siguiera el esquema del árbol.