Roberto Portillo Torres

Árboles autoajustables

En este documento se registran dos casos de prueba muy sencillos que muestran el comportamiento del tipo de datos. A continuación se muestran unas gráficas que representan el tiempo de ejecución con un banco de pruebas de mayor tamaño. Finalmente haremos una comparación entre el coste que este tipo tiene teóricamente y el que hemos obtenido en la práctica.

-Prueba 1:

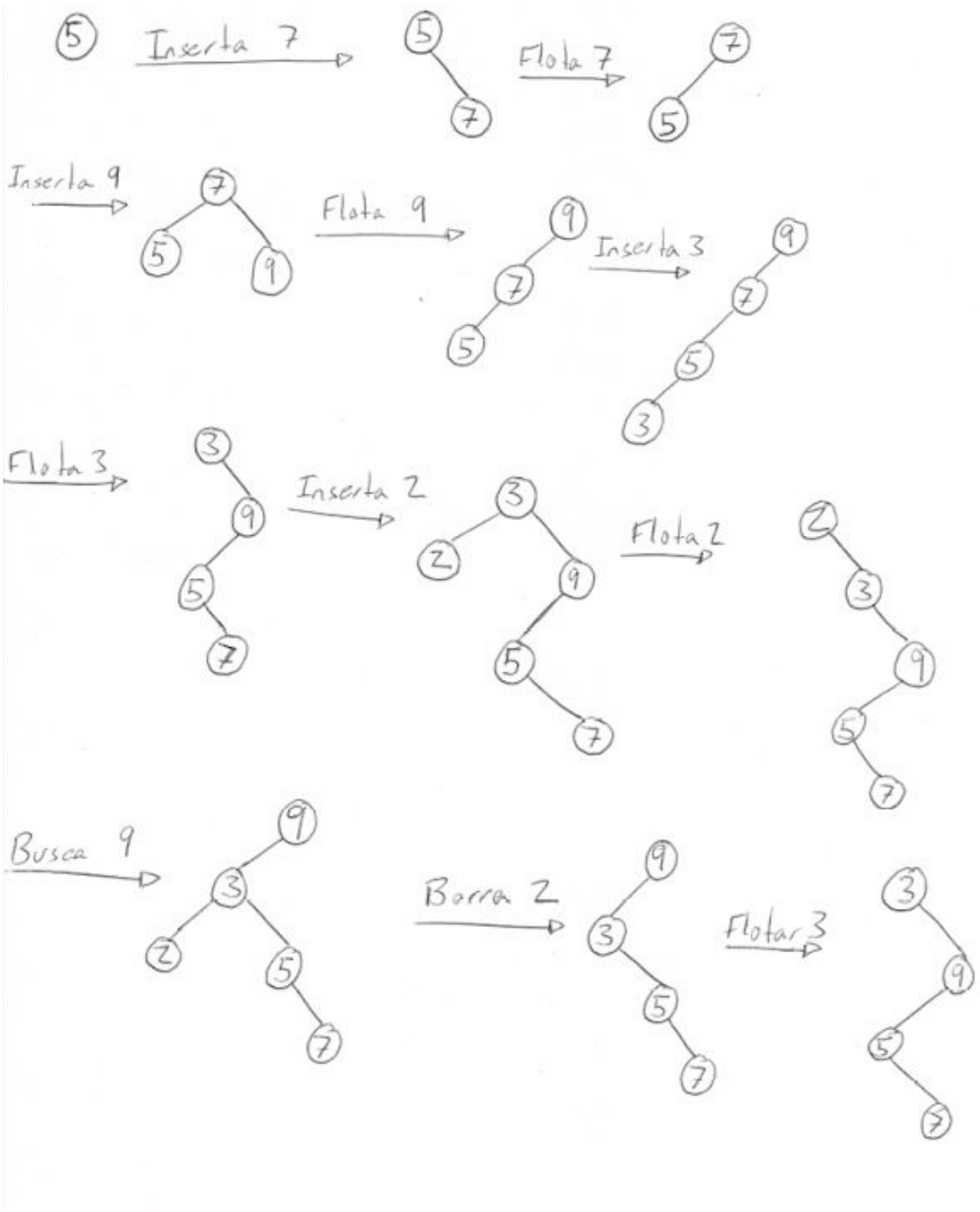
Vamos a insertar cinco elementos en el árbol y a continuación realizar una búsqueda y un borrado prestando atención a las rotaciones. Este es el código probado:

```cpp
#include "autoAdjTree.h"
#include <iostream>
using namespace std;


int main() {

    autoAdjTree<int> arbol = autoAdjTree<int>();

    int nums[] = { 5, 7, 9, 3, 2 };

    for (int i = 0; i < 5; i++) {
        arbol.insert(nums[i]);
        cout << "Insercion del numero: " << nums[i] << '\n';
        arbol.mostrar();
    }

    arbol.buscar(9);
    cout << "Tras buscar el numero 9:\n";
    arbol.mostrar();

    arbol.borrar(2);
    cout << "Tras borrar el numero 2:\n";
    arbol.mostrar();

    system("pause");

    return 0;
}
```

El código está adjunto a este fichero listo para compilar y ejecutar con el nombre de *prueba1.cpp*. Al ejecutarlo mostrará en consola el árbol paso a paso. El formato en el que lo muestra es de izquierda a derecha, a la izquierda quedaría la raíz del árbol y a la derecha las hojas del mismo.

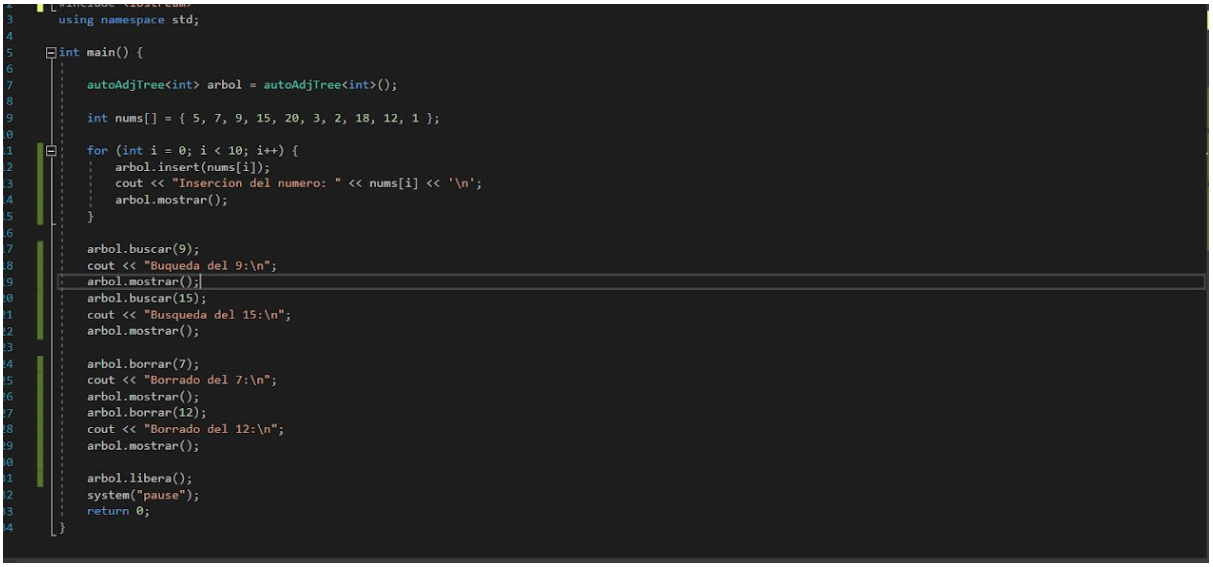
En la siguiente página hay un esquema detallado de cómo se realiza cada operación sobre el árbol y como resulta después de flotar el nodo pertinente en cada ocasión.

1


5
Insere 7
Flota 7
Insere 9
Flota 9
Insere 3
Flota 3
Insere 2
Flota 2
Busca 9
Borra 2
Flotar 3

-Prueba 2:

En este caso vamos a insertar diez elementos, esto llevará al árbol a realizar rotaciones encadenadas y servirá para probar su correcto funcionamiento. Después realizaremos dos búsquedas y dos borrados. El código es el siguiente:



```
using namespace std;

int main() {

    autoAdjTree<int> arbol = autoAdjTree<int>();

    int nums[] = { 5, 7, 9, 15, 20, 3, 2, 18, 12, 1 };

    for (int i = 0; i < 10; i++) {
        arbol.insert(nums[i]);
        cout << "Insercion del numero: " << nums[i] << '\n';
        arbol.mostrar();
    }

    arbol.buscar(9);
    cout << "Buqueda del 9:\n";
    arbol.mostrar();
    arbol.buscar(15);
    cout << "Busqueda del 15:\n";
    arbol.mostrar();

    arbol.borrar(7);
    cout << "Borrado del 7:\n";
    arbol.mostrar();
    arbol.borrar(12);
    cout << "Borrado del 12:\n";
    arbol.mostrar();

    arbol.libera();
    system("pause");
    return 0;
}
```

El código está también adjunto a este documento listo para ejecutar con el nombre *prueba2.cpp*. En su ejecución mostrará por pantalla el árbol en cada momento bajo el mismo formato que la primera prueba.

A continuación observamos un resumen de la salida del programa:

<u>Tras la inserción de los 10 números:</u>

---------20
------18
---------15
---12
------------9
------------------7
---------------5
---------3
------2
1

<u>Búsqueda del 9</u>:

---------20
------18
---------15
---12
9
------------7
---------5
------3
---------2
---1

Búsqueda del 15:

------20
---18
15
------12
---9
----------7
----------5
---------3
----------2
------1

Borrado del 7:

----------20
---------18
------15
---------12
---9
------5
---------3
----------2
----------1

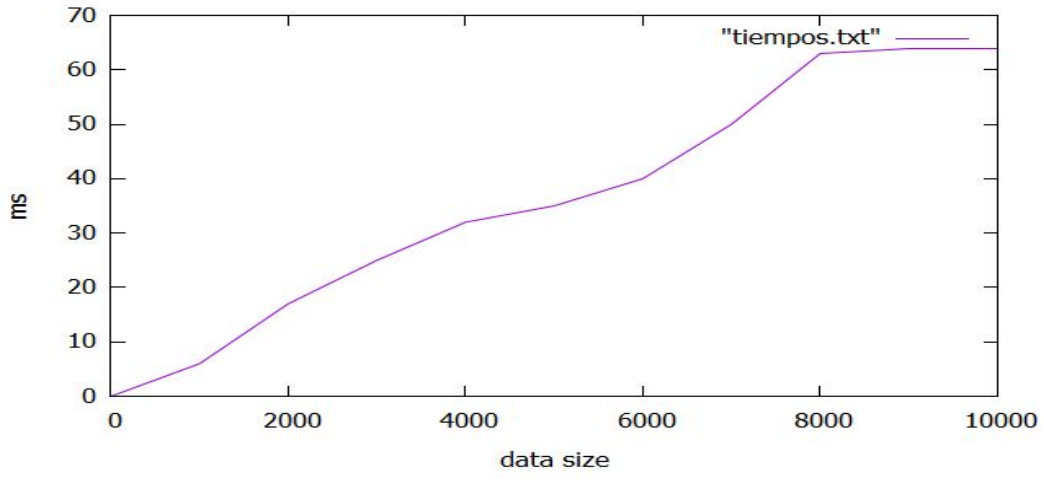
Borrado del 12:

------20
---18
15
---9
------5
---------3
----------2
----------1

-Banco de pruebas:

También está adjunto y listo para ejecutar el fichero *datos.cpp*. Este realiza pruebas más pesadas sobre el tipo de datos. En concreto realiza N inserciones, N búsquedas y N borrados sobre un árbol dando a N valores de 1000 a 10.000.

Este programa no tiene ningún tipo de salida relacionado con el estado del árbol pero si vuelca los tiempos de ejecución de las operaciones para cada tamaño en el fichero *tiempos.txt*. A continuación mostramos estos datos en un gráfica:



-Conclusiones:

Sleator y Tarjan afirmaban que el coste amortizado de las operaciones insertar, borrar y buscar era logarítmico respecto al cardinal del árbol. Observando los resultados del banco de pruebas podemos afirmar que esto se cumple en la práctica dado que a medida que aumenta el tamaño de los datos y el número de operaciones, el crecimiento del tiempo de ejecución es cada vez menor.

-Notas:

Queda adjuntado junto a este documento los siguientes ficheros:

- *autoAdjTree.h* contiene el tipo de árbol autoajustable (*Splay Tree*).
- *datos.cpp* contiene el banco de pruebas
- *numeros.txt* contiene números aleatorios empleados en el banco de pruebas
- *prueba1.cpp* contiene la primera prueba previamente explicada
- *prueba2.cpp* contiene la segunda prueba previamente explicada