Detalles de implementación *Splay Trees*

Francisco Javier Blázquez Martínez

Resumen

Trabajo realizado en el marco de la asignatura *Métodos algorítmicos en resolución de problemas*. El objetivo de este documento es el de explicar las decisiones de diseño tomadas en el desarrollo del código y aclarar distintos aspectos de las implementaciones en *C++* y *Java* que puedan generar confusión.

1. Introducción

Los árboles binarios de búsqueda autoajustables (*splay trees*) aparecen en 1985 ideados por Robert Tarjan y Daniel Selator. Estos no son sino árboles binarios de búsqueda con una peculiaridad, una operación interna sobre los nodos llamada "*splay*". Esta operación, invisible en todo momento al usuario, recibe un nodo como parámetro y transforma el árbol de partida en un árbol binario de búsqueda equivalente (manteniendo exactamente los elementos iniciales) en el cual la raíz es el nodo seleccionado. Intuitivamente, esta operación se emplea para conseguir que elementos recientemente requeridos se encuentren cerca de la raíz y, por tanto, sea más rápido volver a acceder a ellos. De aquí proviene su apellido "*autoajustable*" pues esta operación de splay consigue que el árbol dé preferencia en alturas menores a elementos más empleados en cualquier secuencia de operaciones.

El principal inconveniente de esta estructura de datos, sin embargo, es que, a diferencia de los árboles AVL o los árboles Red-Black, no se mantiene ningún invariante en la altura del árbol, pudiendo llegar a ser esta del orden del número de elementos. Como ya sabemos, búsqueda, inserción y borrado en un árbol binario son operaciones del orden de la altura del árbol. Esto quiere decir que el coste en el caso peor de estas operaciones en splay trees está en *O(n)*. A pesar de esto, y precisamente el hecho de su publicación por R. Tarjan y D. Selator, se puede demostrar que en cualquier secuencia de inserciones, borrados y búsquedas de elementos estas operaciones tienen un coste amortizado logarítmico. De hecho, se puede aprovechar esta propiedad de los splay trees de que elementos accedidos recientemente son fácilmente accesibles para conseguir una mayor eficiencia bajo ciertas circunstancias (no aleatorias). Aquí nos limitaremos a explicar los detalles de la implementación presentada, para una mayor comprensión recomiendo consultar la bibliografía al final del documento.

2. Implementación

Como ya hemos mencionado, los splay trees son esencialmente árboles de búsqueda binarios y, como tales, tienen infinidad de aplicaciones. Es preciso antes de comenzar la implementación concretar las operaciones que vamos a permitir ejecutar al usuario. En mi caso he optado por aplicar los splay trees para crear una estrucura de datos de conjunto ordenado de elementos de tipo genérico, con un orden que puede ser especificado por el usuario. La interfaz pública de usuario se puede consultar en la tabla inferior.

Operación	**Coste**	**Caso peor**	**Descripción**
splay_tree()	*O(1)*	*O(1)*	Crea un conjunto vacío
splay_tree()	*O(n)*	*O(n)*	Vacía el conjunto y libera la memoria
bool empty()	*O(1)*	*O(1)*	Devuelve true si el conjunto está vacío
bool insert(elem)	*O(log(n))*	*O(n)*	Devuelve true si inserta el elemento
bool erase(elem)	*O(log(n))*	*O(n)*	Devuelve true si elimina el elemento
bool find(elem)	*O(log(n))*	*O(n)*	Devuelve true si *elem* está en el conjunto
void print()	*O(n)*	*O(n)*	Muestra en preorden los elementos y su altura

El diseño se ha realizado así para asemejarse a la clase *set* de la librería estándar de *C++*. De hecho, que detrás de esta interfaz se encuentra la estructura de datos de splay tree es algo invisible al usuario. Una diferencia a destacar con respecto a la clase set de la *C++ STL* es que se incluye el método *print()*. Esto es para poder visualizar en consola la estructura completa del conjunto, cómo se almacenan y cómo varía la posición de los nodos al ejecutar las distintas operaciones, más allá del uso como conjunto de la clase. Cabe destacar también de la implementación las siguientes características:

1. No se permite la inserción repetida de un mismo elemento, esto es, nunca nuestro splay tree subyacente al conjunto implementado contendrá dos nodos con elementos iguales. Para garantizar esto es necesario también que se dé el punto número dos.
2. El usuario puede decidir el orden que rige la estructura siempre que este constituya un orden total sobre el conjunto de posibles elementos. La estructura implementada está preparada para funcionar con un comparador cualquiera pero puede llegar a estados inconsistentes si esta condición no se satisface.
3. A diferencia de otras implementaciones de splay trees, no se contempla la unión de árboles/conjuntos en la estructura implementada. Esto tiene importancia en los métodos de inserción y borrado de elementos.
4. La estructura de splay tree que subyace a la implementación de conjunto puede ser implementada sin añadir atributos adicionales a los nodos del árbol (más allá del elemento y los punteros a ambos hijos). Por una mayor sencillez en la implementación he optado por añadir un atributo, un puntero al nodo padre para poder actualizar sus atributos de forma sencilla en *O(1)*.

3. Algoritmos

3.1. Atributos de la clase

Como ya hemos dicho, hemos optado por añadir como atributo propio de cada nodo un puntero a su nodo padre para simplificar el código. También hemos optado por que tanto la clase nodo (invisible al usuario) como los atributos de la clase sean *protected* y no *private* en vista a posibles modificaciones futuras y extensiones de funcionalidad mediante herencia. Abajo se muestra la declaración de la clase sin métodos.

```
template<typename T, typename Comp = std::less<T>>
class splay_tree
{
  protected:
    struct Node
    {
      T     elem;
      Node* left;
      Node* right;
      Node* parent;

      Node(Node* l, T const& e, Node* r, Node* p)
          :elem(e),left(l),right(r),parent(p) {}
    };

    Node* root;
    Comp  lower;
}; //splay_tree
```

3.2. Búsqueda

Se realiza igual que en el caso de un árbol binario de búsqueda convencional con la salvedad de que se realiza un splay del nodo encontrado o el último nodo visitado si el elemento no se encuentra en el árbol. Recuerdo que *lower* es un comparador que actúa sobre los elementos de tipo T (ver el código añadido abajo). Se aprecia en este método por qué el comparador debe crear un orden total en el conjunto de posibles valores de T. En el bucle de descenso desde la raíz, buscando el elemento recibido como parámetro, se realiza una comparación nodo a nodo. Si el parámetro es menor que el elemento del nodo, la búsqueda continúa por el hijo izquierdo del nodo. Si el parámetro es mayor, continúa por el hijo derecho. Si un elemento no es mayor ni menor que el recibido como parámetro, el método sobreentiende que es igual. Es por esto que el comparador debe establecer un orden total estricto.

```
template<typename T, typename Comp>
bool splay_tree<T, Comp>::find(T const& elem)
{
  Node* father = nullptr; Node* node = root;

  while(node != nullptr)
  {
    father = node;

         if(lower(elem, node->elem))     node = node->left;
    else if(lower(node->elem, elem))     node = node->right;
    else {splay(node); return true;} // Encuentra el elemento
  }

  if(father != nullptr) splay(father);

  return false;
}
```

3.3. Inserción

La inserción es igual que en un árbol binario de búsqueda con la salvedad de que al terminar el proceso se realiza una llamada al método splay con el nodo insertado, de realizarse la inserción, o con aquel nodo que ya contiene la clave a insertar. No incluir la función de unión de árboles descarta la posibilidad de reducir la inserción de elementos a la unión del árbol actual con un árbol con un único nodo, el del elemento a insertar. Sí que se podía optar por una implementación recursiva en vez de iterativa a la hora de descender por los nodos del árbol. En mi caso he optado por una implementación iterativa pues en la implementación recursiva, al llegar a un puntero nulo al final del descenso no se puede saber cuál es su nodo padre si no es añadiendo un atributo más como parámetro. La implementación recursiva alternativa desarrollada puede verse en el fichero *splay_tree.h* como comentario en el código.

```
template<typename T, typename Comp>
bool splay_tree<T, Comp>::insert(T const& elem)
{
  Node* father = nullptr; Node* node = root;

  while(node != nullptr)
  {
    father = node;

         if(lower(elem, node->elem))     node = node->left;
    else if(lower(node->elem, elem))     node = node->right;
```

```
12      else {splay(node); return false;} // El elemento esta
13    }
14
15    // Llegados aqui el elemento no esta, lo insertamos
16    node = new Node(nullptr, elem, nullptr, father);
17
18    // Actualizamos su padre y, si es necesario la raiz
19    if(father != nullptr)
20      {if(lower(elem, father->elem)) father->left = node;
21       else                          father->right= node;}
22    else
23      root = node;
24
25    // Finalmente llevamos el nuevo nodo a la raiz
26    splay(node);  return true;
27  }
```

3.4. Borrado

Al igual que en los métodos anteriores, la filosofía es la misma que en los árboles binarios convencionales. De nuevo la diferencia la marca la llamada al método splay al terminar el borrado. En algunas variantes de splay trees, se opta por flotar primero el nodo a borrar a la raíz y luego borrarlo. En mi caso he optado por primero borrar el elemento (de estar en el árbol) y entonces flotar su nodo padre. El elemento borrado se reemplaza por el menor elemento del conjunto mayor que el del nodo a borrar para mantener la estructura de árbol binario de búsqueda. Si el elemento a borrar no se encuentra en el conjunto se llama al método splay con el último nodo visitado durante la búsqueda del elemento.

```
1   template<typename T, typename Comp>
2   bool splay_tree<T, Comp>::erase(T const& elem)
3   {
4     Node* father = nullptr; Node* node = root;
5
6     while(node)
7     {
8           if(lower(elem, node->elem))
9               {father = node; node = node->left;}
10     else if(lower(node->elem, elem))
11              {father = node; node = node->right;}
12     else break; // Llegamos al nodo a borrar
13    }
14
15    /*   ACLARACIONES:
16    *
```

```
17      *    A partir de este momento node apunta al nodo que queremos
18      *    borrar, vamos a distinguir casos segun su situacion.
19      *
20      * -> Los punteros se pueden interpretar como booleanos. En C++
21      *    un puntero equivale a falso si y solo si es nulo.
22      * -> El unico nodo con padre nulo es la raiz. La condicion
23      *    (father == nullptr) equivale a (node == root).
24      */
25
26        // Si el elemento no esta, flotamos el padre y terminamos
27      if(node == nullptr) {if(father) splay(father); return false;}
28
29      if(!node->left && !node->right)   // CASO 1: Hoja
30      {
31        if(node->parent == nullptr) root = nullptr;
32        else
33        {
34          if(node == node->parent->left)  node->parent->left = nullptr;
35          else                            node->parent->right= nullptr;
36        }
37      }
38      else if(!node->left)              // CASO 2: Solo hijo derecho
39      {
40        if(node->parent == nullptr)
41            {root = node->right; root->parent = nullptr;}
42        else
43        {
44          node->right->parent = node->parent;
45          if(node == node->parent->left)
46            node->parent->left = node->right;
47          else
48            node->parent->right= node->right;
49        }
50      }
51      else if(!node->right)             // CASO 3: Solo hijo izquierdo
52      {
53          if(node->parent == nullptr)
54              {root = node->left;  root->parent = nullptr;}
55          else
56          {
57            node->left->parent = node->parent;
58            if(node == node->parent->left)
59                node->parent->left = node->left;
60            else
61                node->parent->right= node->left;
62          }
63      }
```

```
64    else                                  // CASO 4: Ambos hijos existen
65    {
66      // Hallamos el elemento que reemplace el nodo a borrar
67      Node* minGreater = node->right;
68      while(minGreater->left) minGreater = minGreater->left;
69
70      // Lo desconectamos de su posicion
71      if(minGreater->right)
72      {
73        minGreater->right->parent = minGreater->parent;
74
75        if(minGreater == minGreater->parent->right)
76              minGreater->parent->right = minGreater->right;
77        else
78              minGreater->parent->left  = minGreater->right;
79      }
80      else
81      {
82        if(minGreater == minGreater->parent->right)
83              minGreater->parent->right = nullptr;
84        else
85              minGreater->parent->left  = nullptr;
86      }
87
88      // Lo conectamos en la posicion del nodo a borrar
89      minGreater->left  = node->left;
90      minGreater->right = node->right;
91      minGreater->parent= node->parent;
92
93      if(node->left)   node->left->parent = minGreater;
94      if(node->right)  node->right->parent= minGreater;
95
96      if(node->parent)
97        {if(node == node->parent->left)
98              node->parent->left = minGreater;
99         else
100             node->parent->right= minGreater;}
101     else
102       root = minGreater;
103   }
104
105   delete node;  if(father) splay(father);
106   return true;
107 }
```

3.5. Rotación de un nodo

Se contemplan rotaciones hacia la izquierda o hacia la derecha en función de si el nodo es reemplazado por su hijo derecho o izquierdo. Estas son iguales a las rotaciones de los árboles AVL u otros árboles binarios balanceados pero, claro está, actualizando también el atributo añadido a los nodos, el nodo padre al que apuntan. Se incluye el código correspondiente a una rotación hacia la izquierda, siendo el código de la rotación hacia la derecha simétrico.

```
template<typename T, typename Comp>
void splay_tree<T, Comp>::rotateRight(Node* node)
{
  if(node == nullptr)
    throw std::domain_error("Right rotation over null node!");

  Node* father    = node->parent;   //Guardamos el padre
  Node* leftChild = node->left;     //Guardamos el hijo izquierdo

  if(leftChild != nullptr)
  {
    node->left        = leftChild->right;
    node->parent      = leftChild;
    leftChild->parent = father;
    leftChild->right  = node;

    if(node->left != nullptr)  node->left->parent = node;
    if(father     != nullptr)
      {if(node == father->left)  father->left = leftChild;
       else                      father->right= leftChild;}
    else
      root = leftChild;
  }
}
```

3.6. Splay

El algoritmo lleva el nodo recibido como parámetro a la raiz manteniendo la estructura de árbol binario de búsqueda. Para esto, mientras el nodo no se sitúe en la raiz, si su nodo padre es la raíz, realiza una única rotación para ocupar la posición de su nodo padre (rotación izquierda o derecha según corresponda). Si por contra el nodo tiene un "nodo abuelo" combina dos rotaciones que llevan el nodo a flotar a dos alturas menos. Al no hacer uso del comparador, tanto este método como las rotaciones que emplea son consistentes a pesar de que el orden inferido del comparador no sea total.

```
template<typename T, typename Comp>
void splay_tree<T, Comp>::splay(Node* node)
{
  if(node == nullptr)
    throw std::domain_error("Splay over null node!");

  while(node->parent)
  {
    if(!node->parent->parent)  // Hijo directo de la raiz
    {
      if(node->parent->left == node)  rotateRight(node->parent);
      else                            rotateLeft(node->parent);
    }
    else                       // El nodo tiene nodo abuelo
    {
      Node* grandParent = node->parent->parent;

      if(grandParent->left       && node==grandParent->left->left)
       {rotateRight(grandParent);
        rotateRight(node->parent);}
      else if(grandParent->right && node==grandParent->right->right)
        {rotateLeft(grandParent);
         rotateLeft(node->parent);}
      else if(grandParent->left  && node==grandParent->left->right)
        {rotateLeft(node->parent);
         rotateRight(grandParent);}
      else if(grandParent->right && node==grandParent->right->left)
        {rotateRight(node->parent);
         rotateLeft(grandParent);}
      else
        throw "Illegal state reached during splay operation!";
    }
  }
}
```

4. Tests

Por mayor sencillez a la hora de comprobar el correcto funcionamiento de la clase he incluido dos tests, uno manual y otro automático. Esta sección tiene el objetivo de explicar el funcionamiento de estos para que puedan ser ejecutados por cualquier persona con interés en apreciar los múltiples cambios de estructura de los splay trees en una secuencia de operaciones. Se puede también fácilmente incluir mediciones del tiempo de ejecución en el test automático para convencerse de que el coste amortizado de las operaciones es logarítmico.

Se incluye abajo el código correspondiente para ejecutar un test manual de la clase *splay_tree*. Este test lee por consola un código de operación y ejecuta la operación correspondiente:

- -1 para terminar el test.
- 0 para mostrar el árbol.
- 1 para buscar un elemento (se lee a continuación).
- 2 para insertar un elemento (se lee a continuación).
- 3 para borrar un elemento (se lee a continuación).

```
#include "splay_tree.h"
#include <cstdlib>
#include <time.h>
using namespace std;

int main()
{
  splay_tree<int> arbol;
  int  opcode, aux; cin >> opcode;

  while(opcode != -1)
  {
    cin >> aux;

    try
    {
      if(opcode == 0)     {arbol.print(); cout << endl;}
      else if(opcode == 1) arbol.find(aux);
      else if(opcode == 2) arbol.insert(aux);
      else if(opcode == 3) arbol.erase(aux);
      else if(opcode == 4) {if(arbol.empty()) cout << "VACIO\n";
                            else           cout << "NO VACIO\n";}
    }
    catch (const char* msg)
    {
      cerr << msg << endl;
    }
    catch (exception &e)
    {
      cout << e.what() << endl;
    }

    cin >> opcode;
  }
}
```

Se muestra a continuación el código correspondiente al test automático. Realiza primero un número de inserciones de elementos aleatorios, luego una secuencia de búsquedas de elementos también aleatorios y en el mismo rango que en el caso anterior, y por último una serie de borrados (en el ejemplo 1000). Para terminar muestra los elementos restantes en el árbol. Si sucede algún error durante el test se muestra por consola la excepción recibida.

```
#include "splay_tree.h"
#include <cstdlib>
#include <time.h>
using namespace std;

int main()
{
  splay_tree<int> arbol;
  int upLimit = 1000, nextValue;
  srand(time(NULL));

  try
  {
    for(int i = 0; i < 1000; i++)
    {
      nextValue = rand()%upLimit;
      arbol.insert(nextValue);
    }

    for(int i = 0; i < 1000; i++)
    {
      nextValue = rand()%upLimit;
      arbol.find(nextValue);
    }

    for(int i = 0; i < 4000; i++)
    {
      nextValue = rand()%upLimit;
      arbol.erase(nextValue);
    }
  }
  catch (const char* msg)  {
    cerr << msg << endl;
  }
  catch (exception &e)  {
    cout << e.what() << endl;
  }

  arbol.print();
}
```

Referencias

[1] Ricardo Peña, *Árboles de búsqueda autoajustables (Splay trees)*. Apuntes de la asignatura "Métodos algorítmicos en resolución de problemas", Facultad de informática, Universidad Complutense, Madrid, curso 2015/16.

[2] E. Horowitz, S. Sahni, y D. Mehta. *Fundamentals of Data Structures in C++*. Computer Science Press, 4ª edición, 1997. Capítulo 10.7.

[3] *Splay tree and it's implementation*, versión del 10 de enero de 2019. Disponible en `https://codeforces.com/blog/entry/18462`.

[4] *Splay tree*, versión disponible a 10 de enero de 2019 en `http://en.wikipedia.org/wiki/Splay_tree`.