Red-Black Hash

TL;DR;
Desarrolle un árbol rojinegro genérico ("emplantillado") que sirva para implementar una tabla hash (arreglo asociativo); cree una visualización del árbol utilizando SVG.

A nuestra compañía de software le interesa profundizar más sobre las habilidades aprendidas en la anterior prueba de programación. Esta vez requerimos evaluar conocimientos más técnicos, es por eso que tenemos a disposición la siguiente prueba de programación.

Suponga que usted tiene una estructura de **Nodo** que "intenta" almacenar cualquier tipo de dato. Es posible que, mediante un **if** preguntemos si el tipo de dato es **int**, **string** u **objetos personalizados**, es decir, objetos de cualquier tipo, como "Perro", "Casa" o "Sanguche". Dicho sea de paso, esa estructura de Nodo funciona para varios tipos de contenedores, tales como Listas Enlazadas, Colas, Pilas, Listas Doblemente Enlazadas y Árboles Binarios.

Usted conoce bien qué es un Árbol de Búsqueda Binaria (Árbol BST), aquel cuyos nodos tienen a lo mucho dos hijos. Se necesita que usted desarrolle un árbol BST con una definición como la siguiente:

```
class ArbolBST(
  raiz
  T_str

  ArbolBST(T)
  insert(dato)
  delete(dato)
  search(dato)
  porNiveles(...)
  preorder(...)
  inorder(...)
```

```
  postorder(...)
  toString()
)
```

Es normal que para sus algoritmos usted requiera conocer cuál tipo de dato se está administrando en ese momento, para tomar un rumbo u otro. La variable *T_str* podría servirle para eso, indistintamente de si está utilizando templates para crear algoritmos genéricos. Sin embargo, podría no ser tan sencillo encontrar cómo obtener el tipo T específicamente en un template. En C++ (11) por ejemplo, podría utilizar algunos macros para obtener el tipo genérico:

```
#include <iostream>

#ifdef _MSC_VER
#define FUNCSTR __FUNCSIG__
#else
#define FUNCSTR __PRETTY_FUNCTION__
#endif

using namespace std;

template <typename T>
void algoritmo(){
    cout << "algoritmo.T: "<< FUNCSTR << endl;
}

int main(){
    algoritmo<int>();
    return 0;
}
```

Sin embargo, hay que evaluar el resultado, ya que el tipo no siempre vendrá "despejado" y listo (en GCC):

```
"algoritmo.T: void algoritmo() [with T = int]"
```

En fin, el árbol tiene un método ***insert***, el cual deberá aceptar **cualquier tipo de dato**, y en diferentes variantes. Recuerde los casos:

```
ArbolBST a;

a.insert(67); // número directo (copiar en arbol, después liberar)

int x = 45;

a.insert(x); // a través de variable (también copiar, después liberar)

int b = 234:

a.insert(&b) // acepta referencias (solo asigna, no libera)
```

Lo anterior aplica también para string y para objetos personalizados.

```
ArbolBST a;

a.insert( new Casa() ); /* objeto directo, es un puntero (asignar, después
liberar) ** al ser anónimo está difícil seguirle el rastro para
liberarlo*/

Casa* x = new Casa();

a.insert(x); // acepta referencias (solo asigna, no libera)

Casa b:

a.insert(&b) // a través de variable (copiar, después liberar)
```

Los métodos de recorridos del árbol deberán devolver una representación textual de los nodos del contenedor. Dicho esto, entonces el método ***toString*** devolverá todos los recorridos en un solo texto, separados con un salto de línea, cada uno debidamente etiquetado.

Su tarea consistirá en programar un **árbol rojinegro**. Un Árbol RB es un tipo especial de Árbol Binario en el cual las operaciones de inserción y eliminación de datos necesita un procedimiento automático de balanceo, de modo que el árbol quede lo más simétricamente distribuido. Esto contribuye a que las operaciones comunes en el árbol, tales como insertar, eliminar, y buscar se lleven a cabo en un tiempo de ejecución O(log n). El balance se mantiene mediante un conjunto de propiedades que tienen que ver con el color de nodo y reglas específicas que ayudan a mantener el árbol aproximadamente balanceado después de cada actualización.

Propiedades clave de un Árbol RB

1. Color de nodo: Cada nodo está coloreado ya sea rojo o negro.
2. Propiedad de la raíz: El nodo raíz siempre es negro.
3. Propiedad del Rojo: Los nodos rojos no pueden tener hijos rojos.
4. Propiedad del Negro: Cada camino desde un nodo hasta una de sus hojas descendientes tiene el mismo número de nodos negros (altura negra).
5. Nodos hoja: Todas las hojas (sean NULL o con sentinela) son considerados de color negro.

Operaciones básicas

Insertion: Cuando un nodo se agrega es inicialmente coloreado rojo. Si viola las propiedades RB, se necesita hacer rotaciones y re-coloramiento para restaurar las propiedades.
Deletion: Eliminar un nodo puede provocar violaciones de las propiedades, las cuales también se arreglan mediante rotaciones y re-coloramiento, similares a los ajustes por inserción.
Search: La búsqueda estándar de un árbol de búsqueda binaria, se ve beneficiada del mantenimiento del balance.

La implementación de su Árbol RB no será a la libre, deberá cumplir con los siguientes requerimientos:

1. El ArbolRB es una derivación de un ArbolBST, es decir, el ArbolRB hereda el comportamiento de un ArbolBST.
2. Debe extender el comportamiento original mediante atributos y métodos propios de los árboles rojinegros, tales como los métodos de **rotación**.
3. Además del comportamiento original de un ArbolBST, el ArbolRB deberá proveer una funcionalidad extra que permita obtener una representación del árbol en formato SVG (*Scalable Vector Graphics*) para ser visualizado en gráficamente con todo y sus colores.
4. Al igual que la base, el ArbolRB deberá ser completamente genérico ("emplantillado"), es decir, deberá poder administrar cualquier tipo de dato primitivo (int, float, double, char, etc), así como tipo string y además tipos arbitrarios (clases personalizadas hechas por usted).

Para desarrollar su trabajo recuerde todos los detalles mencionados en la clase (presencial):

- Convenciones de implementación. Esto es lo que está obligado el programador a respetar si quiere utilizar nuestro árbol, por ejemplo, puede ser que los objetos que queramos insertar en un árbol necesiten implementar o heredar de una clase base, o quizás, puede ser que este detalle se lo ocultemos al programador y por debajo nosotros lo hacemos.
- Uso de herencia, polimorfismo y abstracción.
- Sobrecarga de métodos para el polimorfismo. Esto es especialmente importante para proveer esas "capas" de abstracción al usuario y que nosotros por debajo "acomodemos" las cosas.
- Sobrecarga de operadores también para el polimorfismo, pero además para uniformar estéticamente las implementaciones de algoritmos y que el código se vea elegante.
- Manejo de clases concretas y abstractas, métodos virtuales y virtuales puros.
- Utilización de templates para abstraer el tipo de dato de los algoritmos genéricos.

Se le proporcionará un código de ejemplo para que a partir de ahí **tome ideas para su solución**. Recuerde que no es un código definitivo, no está completo. De ese código es indispensable que tome el código referente a la generación de la visualización SVG. Eso ya está listo y funcionando en ese ejemplo, usted nada más tendría que adaptarlo a su solución. Por ejemplo, usted podría tomar directamente el código de los métodos encargados del SVG y colocárselo al ArbolRB, olvidándonos de la clase ArbolSVG, o bien, podría hacer que su ArbolBST ya tenga esa funcionalidad y que los árboles derivados la reutilicen, o bien, podría hacer que el ArbolRB herede del ArbolSVG (el cual ya es un ArbolBST). Como puede ver, ideas hay más de una ....

La visualización en SVG nos permitirá visualizar si el trabajo que estamos haciendo con el árbol rojinegro está correcto. Este tipo de archivos los podemos abrir con un navegador web. Recuerde que ya ese código se le proporcionó, usted nada más deberá de adaptarlo.

La Tabla Hash (Arreglo Asociativo)

A grandes rasgos, un arreglo asociativo es lo mismo que un arreglo normal, pero sus índices no son expresados en términos de posiciones numéricas, sino en términos de claves tipo **string**. Por ejemplo, un arreglo normal sería así:

```
int arreglo[ 5 ] = {10, 20, 30, 40, 37};
int medio = arreglo[ 2 ];
arreglo[ 4 ] = 50;
```

Mientras que una versión asociativa sería así:

```
int arreglo[ 5 ] = {10, 20, 30, 40, 37};
int medio = arreglo[ "tercero" ];
arreglo[ "quinto" ] = 50;
```

A los elementos que componen esta estructura de datos se les conoce como pares clave-valor (*key-value pair*).

Comúnmente, una tabla hash tiene un arreglo donde almacena los datos (*values*) y una función especial que le llaman "función hash", la cual dada una clave (*key* , típicamente string), calcula un índice entero válido en el arreglo. En cada casilla del arreglo se encuentra almacenado un valor asociado a una clave, pero las claves no son almacenadas en ninguna parte, sino que solamente sirven para calcular el índice (numérico) real del arreglo.

Hay casos en los que dos o más claves diferentes podrían producir el mismo índice con la función hash, llamados conflictos, y se debe tener una estrategia para administrarlos. Por ejemplo, una estrategia sería que las casillas del arreglo no contengan un valor directamente, sino que contengan un contenedor donde se puedan guardar varios valores.

Para nuestra "tabla hash" (entre comillas) no hará falta crear un arreglo de valores y una función hash, si no que únicamente bastaría con tener un árbol, un ArbolRB, el cual ya es una estructura optimizada. En este caso, como no hay función hash, hay que almacenar las claves y sus valores directamente en el árbol. Luego, en la inserción de datos, en vez de invocar una función hash para saber un índice de un arreglo, utilizamos la búsqueda normal (*search*) del árbol para encontrar la clave y tener acceso al valor.

Como la "tabla hash" debe comportarse como un arreglo asociativo, entonces usted deberá encontrar los mecanismos del lenguaje que le permitan sobreescribir lo que sucede con los símbolos [y], es decir, sobrecarga de operadores. Al final su tabla hash debería poder utilizarse de la siguiente manera:

```
RBHash<int> myHash;
// si no existe la clave, se crea
myHash[ "Brasil" ] = 5;
myHash[ "Italia" ] = 4;
myHash[ "Alemania" ] = 4;
myHash[ "Argentina" ] = 3;
myHash[ "Francia" ] = 2;
```

```
int mundialesBrasil = myHash[ "Brasil" ];
int mundialesArgentinal = myHash[ "Argentina" ];
```

```
RBHash<string> myStrHash;
myStrHash[ "nombre" ] = "Pepe";
myStrHash[ "provincia" ] = "Cartago";
```

```
string nom = myStrHash[ "nombre" ];
string pro = myStrHash[ "provicia" ];
```

```
RBHash<???> myObjHash;
myObjHash[ "clavo" ] = new Clavo( );
myObjHash[ "martillo" ] = new Martillo( );
myObjHash[ "tablilla" ] = new Tablilla( );
```

```
Clavo clavo = myHash[ "clavo" ];
```

Además ...

1. Todo es a consola (línea de comandos), sin bibliotecas externas.
2. Bajo ningún motivo usar el STL (*Standard Template Library*) , por ejemplo list<...>, vector<...>, array, etc.; nada que utilice boquillas <...>, ni estructuras de datos ya definidas en el lenguaje. Estas boquillas solamente aplican para los contenedores que usted mismo(a) crea manualmente.

Esta tarea debe realizarse y entregarse de forma **grupal, máximo 2 estudiantes**.

Entregables:

1. Código fuente.
2. Comandos de compilación.

Los estudiantes deberán trabajar en equipo y presentar evidencia de esto mediante la plataforma GitHub (commits). Deben entregar sus proyectos en la fecha indicada, sin excepción y sin posibilidad de prórrogas. Deberán comprimir su carpeta con la aplicación y subirla a Mediación Virtual.

Trabajos copiados o hechos en su totalidad con inteligencia artificial serán calificados con nota de cero, además de que serán etiquetados para futuras evaluaciones. Por favor no exponerse, si necesita ayuda estamos para servirle.