GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ESTRUCTURAS DE DATOS BASADAS EN POLÍTICAS (ÁRBOL DE ESTADÍSTICAS DE PEDIDOS)

1. Introducción

Un tipo de estructura de datos que se utiliza comúnmente en informática es la estructura de datos basada en políticas. Una estructura de datos basada en políticas es una estructura de datos que permite al programador definir un conjunto de reglas o políticas que gobiernan el comportamiento de la estructura de datos.

Por ejemplo, suponga que tiene un conjunto de datos que desea almacenar en una estructura de datos. Podría utilizar una estructura de datos estándar como una matriz o una lista vinculada para almacenar los datos. Sin embargo, si desea aplicar ciertas políticas sobre los datos, como permitir que solo se almacenen ciertos valores o garantizar que los datos siempre estén ordenados en un orden particular, es posible que necesite utilizar una estructura de datos basada en políticas.

Las estructuras de datos basadas en políticas, como el árbol de estadísticas de pedidos, son estructuras de datos especializadas que permiten consultar y actualizar datos de manera eficiente en función de políticas o reglas específicas.

2. Conocimientos previos

2.1. Árbol de búsqueda binario equilibrado

Un árbol de búsqueda binario equilibrado, también conocido como árbol AVL (por las iniciales de los inventores Adelson-Velsky y Landis), es una estructura de datos en forma de árbol binario de búsqueda en la que se garantiza que la diferencia de alturas entre los subárboles izquierdo y derecho de cada nodo no sea mayor que 1. Esto asegura que el árbol esté equilibrado y mantiene su eficiencia en términos de tiempo de búsqueda, inserción y eliminación.

2.2. Árboles rojo-negro

Un árbol rojo-negro es otra estructura de datos en forma de árbol binario de búsqueda que se utiliza para mantener el equilibrio y garantizar un rendimiento eficiente en términos de tiempo de búsqueda, inserción y eliminación. Los árboles rojo-negro se caracterizan por tener nodos que son rojos o negros, y cumplen con ciertas reglas para mantener el equilibrio del árbol.

Las reglas fundamentales de un árbol rojo-negro son las siguientes:

1. Cada nodo es rojo o negro.
2. La raíz del árbol siempre es negra.
3. Todos los nodos hoja (nodos nulos) son negros.
4. Si un nodo es rojo, entonces sus hijos deben ser negros.
5. Para cualquier nodo dado, todos los caminos desde ese nodo hasta sus nodos hoja descendientes contienen el mismo número de nodos negros.

Estas reglas garantizan que la altura negra de cualquier camino desde la raíz hasta un nodo hoja sea la misma, lo que mantiene el equilibrio del árbol y asegura un rendimiento eficiente en las operaciones de búsqueda, inserción y eliminación.

Durante las operaciones de inserción y eliminación en un árbol rojo-negro, se aplican rotaciones y cambios de colores en los nodos para mantener las propiedades del árbol. Estas operaciones se realizan de manera similar a los árboles AVL, pero con reglas específicas para los colores de los nodos.

2.3. PBDS

PBDS es una abreviatura que puede referirse a *Policy-Based Data Structures* en el contexto de la biblioteca de estructuras de datos estándar de C++ (STL). Las PBDS son una extensión de la STL que proporciona una forma flexible de definir estructuras de datos con políticas personalizadas para operaciones como inserción, eliminación, búsqueda, etc. Esto permite a los desarrolladores adaptar las estructuras de datos a sus necesidades específicas mediante la especificación de políticas en lugar de implementar estructuras de datos personalizadas desde cero.

En el caso de los árboles rojo-negro, la biblioteca PBDS en C++ ofrece una implementación eficiente y flexible de árboles rojo-negro que se pueden personalizar con políticas específicas según los requisitos del usuario. Esta característica hace que sea más fácil y conveniente trabajar con estructuras de datos complejas como los árboles rojo-negro en C++, ya que se pueden adaptar a diferentes escenarios y requisitos sin tener que reescribir toda la estructura desde cero.

3. Desarrollo

Las estructuras de datos basadas en políticas en C++ son algo similares a los conjuntos. Proporcionan algunas operaciones adicionales, pero considerablemente poderosas, que otorgan al programador las virtudes de alto rendimiento, seguridad semántica y flexibilidad en comparación con las estructuras de datos estándar de la biblioteca estándar de C++.

Un árbol de estadísticas de orden es un tipo de árbol de búsqueda binario equilibrado que mantiene información adicional sobre el orden de los elementos del árbol. Esta información permite la recuperación rápida del k-ésimo elemento más pequeño del árbol, así como otras consultas relacionadas con pedidos.

La implementación de un árbol de estadísticas de pedidos normalmente implica aumentar un árbol de búsqueda binario estándar con campos u operaciones adicionales para mantener la información de las estadísticas de pedidos. Esto se puede hacer utilizando técnicas como árboles rojo-negro o árboles AVL.

3.1. Conjunto ordenado

Los conjuntos ordenados pueden verse como versiones extendidas de conjuntos en la estructura de datos basada en políticas.

3.2. Operaciones

Hay dos operaciones adicionales en las estructuras de datos basadas en políticas, además de las de la biblioteca estándar. Se enumeran a continuación:

- **order_of_key(k):** Devuelve el número de elementos estrictamente menores que k.
- **find_by_order(k):** Devuelve la dirección del elemento en el késimo índice del conjunto mientras se utiliza la indexación basada en cero, es decir, el primer elemento está en el índice cero.

Aclarando las operaciones adicionales que ofrece PBDS:

Sea a un conjunto ordenado en el que se insertan los elementos $2,\ 4,\ 3,\ 7,\ 5$.

Entonces los elementos en un conjunto ordenado a se almacenarían en el siguiente orden:

$$2,\ 3,\ 4,\ 5,\ 7$$

- **find_by_order:** Aquí, *find_by_order(x)* devolverá la dirección del (x+1)-ésimo elemento en el conjunto ordenado a. Por ejemplo, *find_by_order(3)* devolverá la dirección de $'5'$ ya que el cuarto elemento (usando indexación basada en 1) presente en el conjunto es 5.
- **order_of_key:** Aquí, *order_of_key(x)* devolverá el número de elementos estrictamente menor que x. Por ejemplo, *order_of_key(10)* devolverá 5, ya que todas las entradas del conjunto son menores de 10.

Supongamos que tenemos una situación donde los elementos se insertan uno por uno en un arreglo y después de cada inserción, se nos da un rango $[l, r]$ y tenemos que determinar el número de elementos en el arreglo mayor que igual a l y menor igual a r. Inicialmente, el arreglo está vacío.

Aplicando esta estructura con estas funciones podemos resolver el problema anterior fácilmente, es decir, el recuento de elementos entre l y r se puede encontrar mediante:

```
o_set.order_of_key(r+1) - o_set.order_of_key(l)
```

Como el conjunto contiene solo los elementos ÚNICOS, para realizar las operaciones en un arreglo que tiene elementos repetidos, podemos tomar la CLAVE como un par de elementos en lugar de un número entero en el que el primer elemento es nuestro elemento requerido del arreglo y solo el segundo elemento. del par debe ser único para que todo el par sea único.

4. Implementación

Esta estructura solo está presente en el lenguaje de programación C++.

```
//Incluya los siguientes archivos de encabezado en su codigo para usar PBDS:
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
```

```
//Namespace
using namespace __gnu_pbds;
//Plantillas
//definir plantilla cuando todos los elementos son distintos
template <class T> using nombre_tu_estructura_v1 = tree<T, null_type,
less<T>, rb_tree_tag,tree_order_statistics_node_update>;
//definir plantilla cuando tambien se utilizan elementos duplicados
template <class T> using nombre_tu_estructura_v2 = tree<T, null_type,
less_equal<T>, rb_tree_tag,tree_order_statistics_node_update>;

nombre_tu_estructura_v1 <int> B;
nombre_tu_estructura_v1 <pair<int,int> > A;

nombre_tu_estructura_v2 <int> b;
nombre_tu_estructura_v2 <pair<int,int> > a;
```

- **T:** T hace referencia al tipo de dato que va almacenar la estructura.
- **null_type:** Es la política mapeada. Aquí es nulo usarlo como un conjunto. Si queremos obtener el mapa pero no el conjunto, como segundo argumento se debe usar el tipo mapeado.
- **less:** Es la base para la comparación de dos funciones.
- **rb_tree_tag:** tipo de árbol utilizado. Generalmente son árboles rojos y negros porque la inserción y eliminación tardan un tiempo $\log(N)$, mientras que otros tardan un tiempo lineal, como *splay_tree*. Hay tres clases base proporcionadas en STL para esto: *rb_tree_tag* (árbol rojo-negro), *splay_tree_tag* (árbol de visualización) y *ov_tree_tag* (árbol de vectores ordenados). Lamentablemente, en las competiciones solo podemos usar árboles rojo-negro para esto porque el árbol splay y el árbol OV usan una operación de división cronometrada lineal que nos impide usarlos.
- **tree_order_statistics_node__update:** Está incluido en *tree_policy.hpp* y contiene varias operaciones para actualizar las variantes de nodos de un contenedor basado en árbol, de modo que podamos realizar un seguimiento de metadatos como el número de nodos en un subárbol. Por defecto está configurado en *null_node_update*, es decir, información adicional no almacenada en los vértices. Además, C++ implementó una política de actualización *tree_order_statistics_node_update*, que, de hecho, lleva a cabo las operaciones necesarias.

```
#include <iostream>
#include <bits/stdc++.h>
//Biblioteca para trabajar con la estructura
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <functional> // for less
using namespace std;
//namespace para trabajar con la estructura
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
```

```
tree_order_statistics_node_update>
ordered_set;

int main(){
   ordered_set p;
   p.insert(5); p.insert(2); p.insert(6); p.insert(4);
   // valor en el tercer indice de la matriz ordenada.
   cout << "El valor en el tercer indice::"<< *p.find_by_order(3) << endl;
   // indice del numero 6
   cout << "El indice del numero 6::" << p.order_of_key(6)<< endl;
   // El numero 7 no esta en el conjunto, pero mostrara el
   // numero de indice si estaba alli en la matriz ordenada.
   cout << "El indice del numero siete ::"<< p.order_of_key(7) << endl;
   return 0;
}
```

Para insertar múltiples copias del mismo elemento en el conjunto ordenado, un enfoque simple es reemplazar la siguiente línea,

```
typedef tree<int , null_type, less<int> , rb_tree_tag ,
   tree_order_statistics_node_update> ordered_set;
```

Por

```
typedef tree<int , null_type ,  less_equal<int> , rb_tree_tag ,
   tree_order_statistics_node_update> ordered_multiset
```

Como se muestra en el siguiente ejemplo:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <functional> // for less
#include <iostream>
using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less_equal<int>, rb_tree_tag,
tree_order_statistics_node_update>
ordered_multiset;


int main(){
   ordered_multiset p;
   p.insert(5); p.insert(5);
   p.insert(5); p.insert(2);
   p.insert(2); p.insert(6);
   p.insert(4);
   for (int i = 0; i < (int)p.size(); i++) {
```

```
      cout << "El elemento presente en el indice " << i << " es ";
      // Elemento de impresion presente en el indice i
      cout << *p.find_by_order(i) << ' ';
      cout << '\n';
   }
   return 0;
}
```

Sin embargo, el enfoque anterior no se recomienda porque la estructura de datos basados en políticas de G++ no está diseñada para almacenar elementos en un orden estricto y débil, por lo que usar *less_equal* con ella puede provocar un comportamiento indefinido, como que *std::find* proporcione resultados incorrectos y búsquedas más largas. veces. Otro enfoque es almacenar los elementos como pares donde el primer elemento es el valor del elemento y el segundo elemento del par es el índice en el que se encontró en la matriz. El código fuente a continuación demuestra cómo utilizar este enfoque junto con otros métodos disponibles en el conjunto ordenado.

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <functional> // for less
#include <iostream>
using namespace __gnu_pbds;
using namespace std;

typedef tree<pair<int, int>, null_type,
less<pair<int, int> >, rb_tree_tag,
tree_order_statistics_node_update>
ordered_multiset;

int main(){
   ordered_multiset p;

   // Durante la insercion, utilice {VAL, IDX}
   p.insert({ 5, 0 }); p.insert({ 5, 1 });
   p.insert({ 5, 2 }); p.insert({ 2, 3 });
   p.insert({ 2, 4 }); p.insert({ 6, 5 });
   p.insert({ 4, 6 });

   for (int i = 0; i < (int)p.size(); i++) {
      cout << "El elemento presente en el indice " << i << " es ";
      // Elemento de impresion presente en el indice i
      // Utilice .first para obtener el valor real
      cout << p.find_by_order(i)->first << ' ';
      cout << '\n';
   }
   cout << "\nDeleting element of 2\n\n";
   // Siempre que busque, utilice {VAL, 0} para lower_bound
   //                                {VAL, IDX} para buscar
   //                                {VAL, INT_MAX} para upper_bound
```

```
    p.erase(p.lower_bound({ 2, 0 }));
    for (int i = 0; i < (int)p.size(); i++) {
        cout << "El elemento presente en el indice " << i << " es ";
        cout << p.find_by_order(i)->first << ' ';
        cout << '\n';
    }
    return 0;
}
```

5. Aplicaciones

Uno de los beneficios de utilizar una estructura de datos basada en políticas es que le permite separar el almacenamiento de datos de la manipulación de datos. Esto puede hacer que su código sea más modular y más fácil de mantener.

El árbol de estadísticas de orden es útil en escenarios en los que necesita encontrar rápidamente el k-ésimo elemento más pequeño en un conjunto de elementos o mantener un orden de elementos con inserciones y eliminaciones eficientes.

Las estructuras de datos basadas en políticas, como el árbol de estadísticas de pedidos, tienen diversas aplicaciones en diferentes áreas, incluyendo:

1. **Bases de datos:** En bases de datos relacionales, las *Policy-based Data Structures* pueden ser utilizadas para optimizar consultas que requieren recuperar elementos en un orden específico, como el k-ésimo elemento más pequeño o más grande.
2. **Algoritmos de búsqueda:** En algoritmos de búsqueda y ordenamiento, el *Order Statistics Tree* puede ser utilizado para encontrar rápidamente elementos en un orden específico, lo que es útil en algoritmos de búsqueda binaria y otros algoritmos de búsqueda eficientes.
3. **Estadísticas y análisis de datos:** En aplicaciones que requieren cálculos estadísticos o análisis de grandes conjuntos de datos, las estructuras basadas en políticas pueden ser utilizadas para mantener información sobre el orden de los datos y facilitar la recuperación de estadísticas específicas, como la mediana o percentiles.
4. **Sistemas de información geográfica (GIS):** En sistemas de información geográfica, las *Policy-based Data Structures* pueden ser empleadas para organizar y consultar datos espaciales en un orden específico, por ejemplo, para encontrar rápidamente los puntos más cercanos a una ubicación dada.
5. **Juegos y simulaciones:** En aplicaciones de juegos y simulaciones, las estructuras de datos basadas en políticas pueden ser utilizadas para gestionar y acceder a elementos en un orden específico, como los jugadores con la puntuación más alta o los eventos en un juego en un orden cronológico.

En resumen, las estructuras de datos basadas en políticas, como el árbol de estadísticas de pedidos, son herramientas versátiles que pueden ser aplicadas en una amplia variedad de contextos

donde se requiere un acceso eficiente ya sea en forma de consulta o actualización de los datos en función de políticas o reglas específicas.

6. Complejidad

Al ser una estructura implementadas basada en conjunto (*set*) sus operaciones tienen una complejidad logarítmica y lo mismo pasa con las dos operaciones analizadas de esta estructura.

7. Ejercicios

A continuación una lista de ejercicios ue se pueden resolver aplicando esta estructura de datos:

- CSES - Josephus Problem II
- Timus Online Judge - 1028. Stars
- Timus Online Judge - 1090. In the Army Now
- Timus Online Judge - 1521. War Games 2
- Timus Online Judge - 1439. Battle with You-Know-Who
- Kattis - Galactic Collegiate Programming Contest