GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ESTRUCTURA DE DATOS, COLA CON PRIORIDAD

1. Introducción

Dentro de las estructuras básicas que debe dominar un concursante están la estructura de datos cola con prioridad (Priority Queue). A ellas les vamos a dedicar la siguiente guía de aprendizaje.

2. Conocimientos previos

2.1. Estructuras Dinámicas No Lineales

Estas estructuras se caracterizan por que el acceso y la modificación ya no son constantes, es necesario especificar la complejidad de cada función de ahora en adelante. Entre las estructuras dinámicas no lineales están:

- Árboles
- Grafos
- Cola de Prioridad
- Conjunto
- Diccionario

2.2. Mónticulo (*Heap*)

En computación, un montículo (o heap en inglés) es una estructura de datos del tipo árbol con información perteneciente a un conjunto ordenado. Los montículos máximos tienen la característica de que cada nodo padre tiene un valor mayor que el de cualquiera de sus nodos hijos, mientras que en los montículos mínimos, el valor del nodo padre es siempre menor al de sus nodos hijos.

Un árbol cumple la condición de montículo si satisface dicha condición y además es un árbol binario casi completo. Un árbol binario es completo cuando todos los niveles están llenos, con la excepción del último, que se llena desde la izquierda hacia la derecha.

En un montículo de prioridad, el mayor elemento (o el menor, dependiendo de la relación de orden escogida) está siempre en el nodo raíz. Por esta razón, los montículos son útiles para implementar colas de prioridad. Una ventaja que poseen los montículos es que, por ser árboles completos, se pueden implementar usando arreglos (arrays), lo cual simplifica su codificación y libera al programador del uso de punteros.

3. Desarrollo

Una cola de prioridad es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen, puede llegar a ofrecer la extracción constante de tiempo del elemento más grande (por defecto), a expensas de la inserción

logarítmica, o utilizando una sobrecarga de operadores causaría que el menor elemento aparezca como la parte superior. Trabajar con una cola de prioridad es similar a la gestión de un *Heap*.

Como con cualquier tipo de dato abstracto, una cola de prioridades puede tener múltiples implementaciones, siempre que cumplan las restricciones y el modelo formalizado. A continuación se distinguen implementaciones sencillas frente a otras más especializadas y, también, más habituales para las estructuras de datos que modelen colas de prioridad.

3.1. Implementaciones ingenuas

Hay muchas formas de implementar de forma sencilla, aunque a menudo ineficientemente, una cola de prioridades. A pesar de su ineficiencia, pueden ser muy útiles para observar la analogía con la realidad y comprender el funcionamiento abstracto de una cola de prioridades. Por ejemplo, una implementación posible sería mantener todos los elementos en una lista no ordenada. Cuando se pida el elemento con mayor prioridad, se buscan todos los elementos hasta encontrar el de mayor prioridad (la complejidad de esta estructura tendría, según la notación O grande complejidad computacional de O(1) en inserción, y de O(n) para el desencolado, ya que es la complejidad mínima para buscar en una lista no ordenada).

3.2. Implementación con montículos

Para mejorar el rendimiento, las colas de prioridades suelen implementarse utilizando un montículo como estructura de datos subyacente, y obteniendo así un rendimiento de O ($\log N$) para inserciones y borrados, y de O(N) para la construcción inicial. Existen ciertos tipos especializados de montículos, como los montículos de Fibonacci, que pueden ofrecer mejores cotas asintóticas para algunas operaciones.

En lugar de un montículo, puede utilizarse un árbol binario de búsqueda auto-balanceable, en cuyo caso la inserción y borrado siguen teniendo un coste de O ($\log N$), mientras que la construcción de árboles a partir de secuencias de elementos ya existentes pasaría a tener un coste de O($N \log N$).

Desde el punto de vista de la complejidad computacional, las colas de prioridades son congruentes con los algoritmos de búsqueda.

3.3. C++

Para la utilización de está estructura es necesario añadir en la cabecera del programa **#include <queue>**, gracias a esto ya podemos utilizar la estructura de otro modo no.

- **priority_queue::top():** En una cola de prioridad solo podemos acceder a la raíz del *heap* con esta función. Es como obtener el primer elemento de la cola. Está función si es constante ya que siempre se sabe dónde está la raíz del *heap*.
- **priority_queue::empty():** Está función retorna verdad si la cola de prioridad está vacía y retorna falso en otro caso.

- **priority_queue::size():** Está función retorna cuantos elementos tiene la cola de prioridad.
- **priority_queue::push(valor):** Está función inserta un elemento (valor) a la cola de prioridad, valor debe ser un tipo de dato aceptado por la cola de prioridad. Está función tiene complejidad logarítmica O($\log_2 N$).
- **priority_queue::pop():** Está función elimina el valor que esta en el tope o frente de la cola de prioridad. Está función tiene complejidad logarítmica O($\log_2 N$).

3.4. Java

Para la utilización de está estructura es necesario añadir en la cabecera del programa **import java.util.***, gracias a esto ya podemos utilizar la estructura de otro modo no.

- **add(E elemento):** Agrega un elemento a la cola de prioridad.
- **clear():** Borra la cola de prioridad.
- **addAll(coleccion):** Agrega toda una colección.
- **remove():** Elimina el elemento más pequeño.
- **isEmpty():** Devuelve verdadera si está vacía.
- **contains(E elemento):** Devuelve verdadero si la cola contiene el elemento.
- **size():** Devuelve el número de elementos.

4. Implementación

4.1. C++

```
#include <iostream>
#include <queue>
using namespace std ;
int main (){
   //Creacion de una cola de prioridad
   priority_queue <int> pq ;
   //De esta manera la prioridad sera inversa al valor del elemento.
   priority_queue < int , vector < int > , greater < int > > pqg ;
   //Adicionamos a la cola de prioridad los elementos
   pq.push(100); pq.push(10); pq.push(20); pq.push (100);
   pq.push(70); pq.push(2000);
   if( pq.empty() ){ cout<<" La cola de prioridad esta vacia\n"; }
   else{
      cout<< " La cola de prioridad lleva " ;
      cout<< pq.size() <<" elementos\n" ;
   }
   //Salida : La cola de prioridad tiene 6 elementos
   while (!pq.empty()){
```

```
        int a = pq.top();//O(1) el acceso es constante
        pq.pop(); // O(log n) la eliminacion no es constante
        cout<<a<< "\n" ;
    }//Salida: 2000 100 100 70 20 10
    return 0;
}
```

4.2. Java

```
import java.util.*;
public class Main{
    public static void main(String[] args) {
        PriorityQueue<String> cp = new PriorityQueue<String>();
        cp.add("Juan"); cp.add("Maria"); cp.add("Jose");
        cp.add("Laura");
        System.out.println("Sacando elementos...");
        String datos;
        while (!cp.isEmpty()) {
            datos = cp.remove();
            System.out.println(datos);
        }
    }
}
```

5. Aplicaciones

Al igual que la cola está estructura puede ser útil en ejercicios de simulación donde el orden y además una determinada prioridad sea importante. Uno de las aplicaciones de esta estructura es en el algoritmo de Dijkstra donde la utilización de esta estructura reduce considerablemente la complejidad del algoritmo.

En la gestión de procesos en un sistema operativo. Los procesos no se ejecutan uno tras otro en base a su orden de llegada. Algunos procesos deben tener prioridad (por su mayor importancia, por su menor duración, etc.) sobre otros.

Implementación de algoritmos voraces, los cuales proporcionan soluciones globales a problemas basandose en decisiones tomadas sólo con información local. La determinación de la mejor opción local suele basarse en una cola de prioridad.

6. Complejidad

La complejidad de las operaciones de la cola con prioridad van a depender en gran medida de como fue implementada dicha estructura. En la mayoría de los lenguajes o al menos en C++ y Java su implementación se basa en una árbol de mónticulo tambien conocido como *Heap* por lo

que las operaciones de adicionar o eliminar un elemento tienen una complejidad de O($\log N$) en el caso de obtener el primer elemento o tope de la estructura la complejidad es O(1).

7. Ejercicios propuestos

A continuación una lista de ejercicios que se puede resolver usando esta estructura de datos:

- UVA - 01203 - Argus
- UVA - 11997 - K Smallest Sums
- UVA - 13190 - Rockabye Tobby