COLAS CON PRIORIDADES VARIABLES

Modificación de las colas de prioridad para añadir una operación que permita cambiar la prioridad de un elemento que ya se encuentra en la cola con un coste logarítmico también

UNIVERSIDAD COMPLUTENSE MADRID

ALBERTO VERDEJO

Colas de prioridad con prioridades variables

Como ocurre en algunos algoritmos sobre grafos.

- En ocasiones queremos poder referirnos a elementos que ya están en la cola de prioridad para cambiarles su prioridad.

Y que el elemento ocupe su nueva posición dentro de la cola atendiendo a la nueva prioridad.

- Lo más sencillo es asociar un número distinto a cada elemento y utilizar ese *índice* para referirnos a él.

Asociar a cada elemento un índice, casi como ya habíamos hecho.

- Suponemos que el número de elementos a los que nos vamos a querer referir es fijo, N, y que los elementos están identificados con los índices de 0 a $N - 1$.

Como ocurre en los algoritmos de grafos. donde tenemos unos vértices, podemos numerar los elementos.

IndexPQ

Lo llamamos así porque utilizamos índices para referirnos a los elementos.

La clase `IndexPQ<T>` cuenta con las siguientes operaciones:

- crear una cola de prioridad vacía, `IndexPQ(int N)`
- insertar un elemento, `void push(int e, T const& p)`
- modificar la prioridad de un elemento, `void update(int e, T const& p)`
- consultar el elemento más prioritario, `Par const& top() const`
- eliminar el primer elemento, `void pop()`
- determinar si la cola de prioridad es vacía, `bool empty() const`
- consultar el número de elementos de la cola, `int size() const`

Recibe el número de elementos que pueden formar parte de la cola. Aunque la cola sea vacía, sabemos que vamos a poder meter como mucho N elementos.

índice

prioridad-. El tipo T del parámetro se refiere a las prioridades.

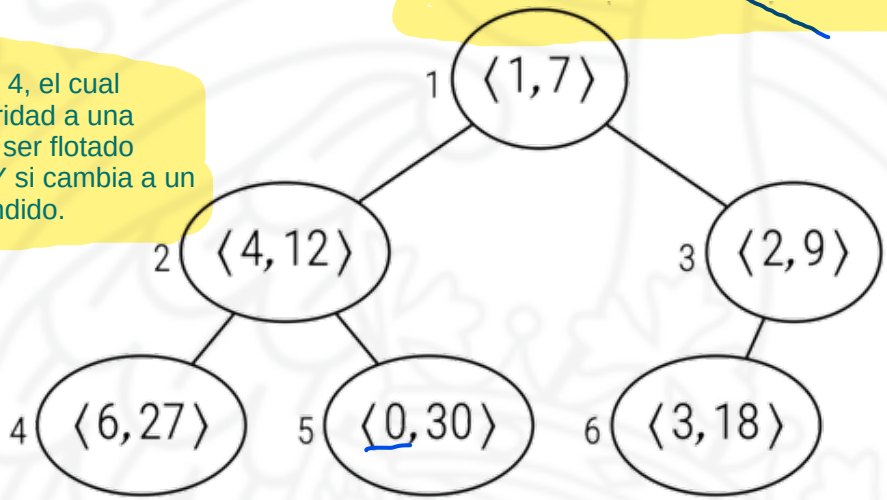
decimos cual es (índice) y cual es su nueva prioridad.

Devuelve un par formado por el identificador del elemento, y la prioridad.

Representación

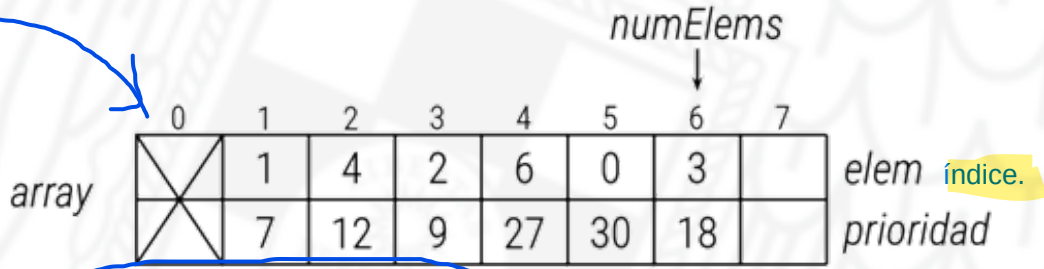
Par <id, prio>

Si queremos cambiar la prioridad del elemento 4, el cual tiene una prioridad de 12 si cambiamos la prioridad a una menor, por ejemplo 5, el elemento tendría que ser flotado porque es un montículo de mínimos y el 5<7. Y si cambia a un número mayor, el elemento tendrá que ser hundido.



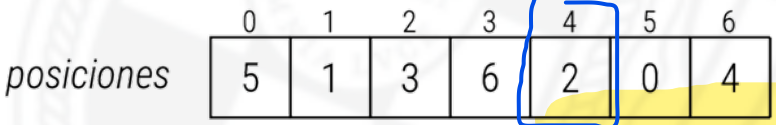
Elementos del 0....6 pero el 5 NO se encuentra en la cola de prioridad.

Acordarnos de que empieza desde el uno



índice.

Así lo encontramos en tiempo constante y podemos flotarlo o hundirlo con coste logarítmico.



posiciones[array[i].elem]=i

El 4 está en la posición dos que es el indice que sale debajo del círculo.

Implementación

IndexPQ.h

```
// T es el tipo de las prioridades
// Comparator dice cuándo un valor de tipo T es más prioritario que otro
template <typename T, typename Comparator = std::less<T>>
class IndexPQ {
public:
   // registro para las parejas < elem, prioridad >
   struct Par { int elem; T prioridad; };

private:
   // vector que contiene los datos (pares < elem, prio >)
   std::vector<Par> array;     // primer elemento en la posición 1

   // vector que contiene las posiciones en array de los elementos
   std::vector<int> posiciones;   // un 0 indica que el elemento no está

   // Objeto función que sabe comparar prioridades.
   // antes(a,b) es cierto si a es más prioritario que b
   Comparator antes;
```

Implementación

IndexPQ.h

```
public:
    IndexPQ(int N, Comparator c = Comparator()) :
            array(1), posiciones(N, 0), antes(c) {}

    Par const& top() const {
        if (size() == 0)
            throw std::domain_error(
                    "No se puede consultar el primero de una cola vacia");
        else
            return array[1];
    }
```

Tamaño N con todas las posiciones 0.

Devuelve el elemento de la posición 1.
El cual es un Par.

Implementación

IndexPQ.h

```
public:
    void push(int e, T const& p) {
        if (posiciones.at(e) != 0)
            throw std::invalid_argument(
                                "No se pueden insertar elementos repetidos.");
        else {
            array.push_back({e, p});
            posiciones[e] = size();
            flotar(size());
        }
    }
```

comprueba si e es un índice válido.
Solo podemos añadir entre 0...N-1

Comprobamos que no esté.

añadimos el par al montículo

Por ahora esta en la posición size()

Floramos el elemento de la posición size

Implementación

IndexPQ.h

Igual que las colas de prioridad pero comparamos prioridades.

```
private:
 void flotar(int i) {
   Par parmov = array[i];
   int hueco = i;
   while (hueco != 1 && antes(parmov.prioridad, array[hueco/2].prioridad)) {
      array[hueco] = array[hueco/2];
      posiciones[array[hueco].elem] = hueco;
      hueco /= 2;
   }
   array[hueco] = parmov;
   posiciones[array[hueco].elem] = hueco;
 }
```

Implementación

IndexPQ.h

```
public:
    void pop() {
        if (size() == 0)
            throw std::domain_error(
                    "No se puede eliminar el primero de una cola vacía.");
        else {
            posiciones[array[1].elem] = 0; // para indicar que no está
            if (size() > 1) {
                array[1] = array.back(); array.pop_back();
                posiciones[array[1].elem] = 1;
                hundir(1);
            } else
                array.pop_back();
        }
    }
```

tenemos que decir que ese elemento está en la posición 1.

Implementación

IndexPQ.h

```
private:
 void hundir(int i) {
    Par parmov = array[i];
    int hueco = i;
    int hijo = 2*hueco; // hijo izquierdo, si existe
    while (hijo <= size())  {
       // cambiar al hijo derecho de i si existe y va antes que el izquierdo
       if (hijo < size() && antes(array[hijo + 1].prioridad, array[hijo].prioridad))
          ++hijo;
       // flotar el hijo si va antes que el elemento hundiéndose
       if (antes(array[hijo].prioridad, parmov.prioridad)) {
          array[hueco] = array[hijo];
          posiciones[array[hueco].elem] = hueco;
          hueco = hijo; hijo = 2*hueco;
       }
       else break;
    }
    array[hueco] = parmov; posiciones[array[hueco].elem] = hueco;
   }
```

Coste log(N)

Implementación

Nueva operación que cambia la prioridad del elemento e.

IndexPQ.h

```
public:
    void update(int e, T const& p) {
        int i = posiciones.at(e);
        if (i == 0) // el elemento e se inserta por primera vez
            push(e, p);
        else {
            array[i].prioridad = p;
            if (i != 1 && antes(array[i].prioridad, array[i/2].prioridad))
                flotar(i);
            else // puede hacer falta hundir a e
                hundir(i);
        }
    }
```

Comprobamos que el elemento sea válido

Si el elemento NO estaba, update se comporta como un push.

Si SI estaba, directamente cambiamos la prioridad del elemento de la posición i

Vemos si hace falta flotar o hundir.

Si el elemento NO está en la raíz y su prioridad es menor que la del padre, entonces hay que FLOTARLO

SI NO LO HUNDIMOS