Laboratorio N°3
Estructura de datos y algoritmos

Integrantes: Hugo Rojas e Isidora González
Profesor laboratorio: Mauricio Hidalgo
Profesor cátedra: Yerko Ortiz
Sección 04
Mails: hugo.rojas1@mail.udp.cl isidora.gonzalez4@mail.udp.cl

Introducción

En el siguiente informe se dará a conocer la modificación y mejora de un código que fue visto previamente, del cual es posible recordar que una empresa hizo entrega de un código el cual traía incluído clases y métodos, donde algunos de ellos fueron eliminados, modificados u ambos, con el objetivo de poder trabajar con una plataforma de videos musicales. Además, fueron implementados métodos para la búsqueda de videos por título, una función para poder revertir la lista entregada y por último, una función dirigida a la popularidad de los videos donde se retornaba el vídeo con más likes.

En adición a los métodos mencionados anteriormente, se hará uso del código anterior con más métodos y funciones implementados, de los cuales se hará una corrección de los errores y se implementarán nuevos métodos con el objetivo de crear una función con listas de prioridad, la cual se encargará de poder permitir al usuario guardar vídeos para “ver más tarde” según la popularidad.

Esta función busca darle facilidad al usuario, de tal manera que una vez visto el video que fue guardado en la lista, sea eliminado de ella, para así continuar y reproducir el siguiente video. Esta lista también busca tener la opción de ordenarse de manera ascendente y descendente, haciendo posible el cambio de orden para ver los videos más antiguos o más nuevos, dependiendo de los deseos del usuario.

A continuación será posible entender y visualizar todos los cambios que fueron realizados en este nuevo código para cumplir el objetivo deseado, explicando paso a paso por función y método lo que fue hecho de una manera clara. Además se utilizaron nuevos tipos de funciones llamados MaxHeap y MinHeap, los cuales están hechos para encontrar el video con mayor popularidad, menor popularidad y hacer posible el orden de los videos según el árbol binario.

Por cada error mencionado, se encontrará en el mismo punto su solución, para así poder tener un mejor entendimiento del informe y estar al tanto de los cambios realizados durante el trabajo, para así lograr estar al tanto de la problemática y la solución.

Errores en los métodos insert y delete de MaxHeap:

Método Insert:

Errores:

- En este método como primer error es posible notar que el arreglo comienza a guardar valores desde la posición 1, está incorrecto porque debería guardar los datos desde la posición 0 y como no es así, se están guardando 49 datos en vez de 50, que es lo solicitado en el contexto.

```
pq[++size] = video;
swim(size);
```

- A raíz de este mismo error se va a generar otro cuando se llame a la función **"swim"** en este mismo método, ya que en esta función se manda como parámetro la posición en la que se encuentra el arreglo, dado que este parte desde la posición 1, va a generar que en la función **"swim"** se cree una excepción, debido a que en la variable **"parent" tipo int** (la cual sería la (posición/2)), se dejará el valor en 0 y esto llegará a una línea en donde se compara el arreglo [0] con el arreglo [1] y como partimos en el arreglo [1] , se intentará comparar el dato con algo nulo y por consecuencia el código no va a compilar.

```
int parent = (k)/2;
if(k>0 && compare(pq[parent], pq[k] ) < 0 ){
```

- Producto del llamado de la función **"swim",** se somete a análisis cada una de las posiciones del arreglo por cada llamado de la función **"insert"**, dentro de la función **"swim"** se presenta el llamado a la función **"swap"**, la cual realizará el cambio de valores en el arreglo, ordenándolos de mayor a menor. En el código original, a la función **"swap"** se mandan el **valor del arreglo en la posición** que llega a la función y la **posición parent**, el inconveniente se presenta en el **arreglo pq**, que guarda información de tipo video y la función **"swap"** recibe información tipo **int**, lo cual indica un cambio e incompatibilidad.

```
if(k>0 && compare(pq[parent], pq[k] ) < 0 ){
    swap(pq[k], pq[parent]);
    swim(parent);
}
```

Soluciones:

- Se elimina de la posición del arreglo un **“++”** para dejarlo como **“pq[size]”**, para así partir desde la posición 0 y lograr mandar a la función **“swim”** la posición 0 para finalizar con un **“size++”**.

```
pq[size] = video;
swim(size);
size++;
```

- Debido a que la fórmula estipulada para calcular el **parent** está dada (k-1)/2 se realiza este cambio, si no se llega a cambiar esto, generaría un error en la estructura del arreglo.

```
int parent = (k-1)/2;
```

- Se cambia dentro de los parámetros del **“swap”** el resultado del arreglo sólo en la posición, ya que lo que se busca lograr es el cambio de las posiciones en el caso de que el **“parent”** sea menor que el **“hijo”**.

```
swap(k, parent);
```

Método Delete:

Errores:

- Se cambia **pq[0] = pq[size- -]** debido a que en esta línea lo que obtendremos será que la primera posición será igual a la última, se genera un error al mandar el size como parámetro para la función **“sink”**, porque esta al ser comparada con el **right**, se comparara con nulo y este generará un error.

```
Cannot invoke "Video.getPopularity()" because "v2" is null
```

- La línea en la que se iguala **pq[size+1] = null** se identifica un error, ya que en teoría en la línea anterior se reduce en uno el **"size",** producto de la eliminación de este valor y es posible decir que **pq[size+1]= null** es redundante o tal vez innecesario, debido a que en esa posición no hay nada.

```
pq[size+1] = null;
```

- Se presenta un error en el llamado de la función **"sink"** en donde se manda como parámetro el **size**, esto es erróneo ya que se busca mandar la **posición 0** en donde se encontrará la última posición del arreglo, dado que así podrá reordenar árbol binario.

```
sink(size);
```

- En la función **"sink"** se cambia la fórmula que definirá la variable **left** y **right**, debido a que si se manda la **posición 0** con la fórmula actual se generará un error y eso provocará que el **padre** sea igual que el **greader**.

- Se analiza en la función **"sink"** el error en **compare < 0**, ya que esto permitirá calcular el orden de manera creciente, siendo que lo que se busca es el orden de manera decreciente.

```
if (left < size && compare(pq[left], pq[greater]) < 0) {
    greater = left;
}

if (right < size && compare(pq[right], pq[greater]) < 0)
    greater = right;
}
```

Soluciones:

- Para solucionar el primer error que se presenta en este método se hace un cambio de signo en **pq[0] = pq [size - -]** quedando finalmente como en la imagen a continuación, ya que si se mantiene como estaba primeramente estipulado, el resultado de esta fórmula sería un error en el código que se muestra en el problema.

```
pq[0] = pq[--size];
```

- Se cambia este error por **pq[size] = null**, debido a que se reduce en uno el **size**, ya que al quedar basura en el arreglo, esta se debería eliminar para optimizar el uso de memoria.

```
pq[size] = null;
```

- Para solucionar este problema se mandará como parámetro a la función **"sink"** la posición 0, así teniendo el número más bajo del árbol como cúspide, así se permitirá ordenar nuevamente el árbol, eliminando el valor más alto sin perder ningún valor.

```
sink( k: 0);
```

- Para poder ordenar el árbol de manera decreciente, se cambiará la dirección del resultado de la función **"compare",** de "menor que 0 " a "mayor que 0".

```
if (left < size && compare(pq[left], pq[greater]) > 0) {
    greater = left;
}

if (right < size && compare(pq[right], pq[greater]) > 0)
    greater = right;
}
```

Completación del método deleteMin:

Para la implementación de este método se utilizarán tres variables auxiliares: un bucle **for** para buscar cual es el valor más pequeño, y la función **“sink”** que permitirá reordenar el árbol de manera decreciente.

La **primera variable auxiliar** de nombre **Max** estará igualada a la cúspide del árbol, el cual es el **valor máximo del árbol ya ordenado**, esta misma variable junto al bucle ayudará a conseguir el valor más pequeño.
La **segunda variable auxiliar** de nombre **posición**, guardará la ubicación del valor más pequeño del arreglo.
La **tercera variable auxiliar** se creará luego de la obtención del valor mínimo y la posición, para así ser declarada como “**minVideo”**, esta variable será la encargada de almacenar el valor del arreglo en la posición mínima, para luego de ser reordenada del árbol pueda ser retornada.
Para reordenar el árbol se utilizará la misma sintaxis que en la función “**delete”** ya modificada, en donde se igualará el arreglo en la posición del valor mínimo por **- -size**, para así reducir el tamaño debido a la liberación de un cupo.
El llamado a la función **“sink”** desde la posición del valor mínimo es para ordenar los datos desde la posición que se busca eliminar el dato hacia el arreglo restante, terminado esto se realizará la igualación del arreglo en la posición **size** para eliminar la basura y con la función ya realizada, se terminará con el **retorno** de la variable auxiliar.
La función quedaría de esta manera:

```
public Video deleteMin(){
    if(size == 0){
        throw new IllegalStateException("La lista está vacía");
    }
    float Max = pq[0].getPopularity();
    int posicion = 0;
    for(int i = 0; i < size ; i++){
        if(pq[i].getPopularity() < Max){
            Max = pq[i].getPopularity();
            posicion = i;
        }
    }
    Video minVideo = pq[posicion];
    pq[posicion] = pq[--size];
    sink(posicion);
    pq[size] = null;
    return minVideo;
}
```

Implementaciones en clase Client:

En esta sección se solicitará un cambio en el orden de la lista de espera, para pasar de tener una lista de orden decreciente a creciente al gusto del usuario, además de la completación de distintas funciones faltantes en este código.

Implementaciones:

Como primera implementación del código tenemos el **"Case 5"** del menú principal. Para este caso, se cargarán los datos de la lista **"Ver más tarde"** a la clase **pqMax** para así ordenarlos de manera **descendente** y a la clase creada en base a este informe **pqMin**, la cual ordena los datos de manera **ascendente**, en donde **"pqMin"** seria un **MinHeap** y **"pqMax"** sería **MaxHeap**.

```
pqMax.insert(actualVideo);
pqMin.insert(actualVideo);
```

En segundo lugar está el **"Case 6"** del menú principal, en donde se solicita la completación de este caso, ya que al hacerlo permitirá al usuario desplegar un menú extra el cual sería la **lista de reproducción "Ver más tarde"**. Se agrega el llamado a la función **"seeLaterMenu"** la cual recibe de manera estándar el **bufferedReader** y la variable **pqMax** de tipo MaxHeap, también se le agrega al llamado de la función **"seeLaterMenu"** el parámetro **Br** que corresponde al **bufferedReader, pqMax y pqMin** y al recibimiento de la función se le agregara **pqMin**.

```
seeLaterMenu(br, pqMax,pqMin);
```

En el **"Case 1"** del menú de lista de espera se solicita completar el caso, para que en la lista de espera se pase al siguiente video. Para realizar esto se implementa el llamado a la función **"getNext()"** la cual permite acceder al siguiente video de la lista **pqMax**.

```
actual = actual.getNext();
```

En el **"Case 2"** del menú de lista de espera se solicita el cambio de orden de la lista de espera, para esto se hará uso de la función **"Bool"** ya creada en el código de nombre **maxOrder**, la cual está estipulada en **verdadero** y lo primero a realizar será dejar esta variable en **falso** para mejor entendimiento, luego se realizará un **if**, que en el caso de que **MaxOrder sea verdadero** se **extraerán datos** de la lista **pqMax** y si **MaxOrder es falso**, **se obtendrán** los datos de la lista **pqMin**, la cual estaría **invertida en base a pqMax**.

```
case "2" -> {
    System.out.println("Cambiando orden...");
    maxOrder= !maxOrder;
    if (maxOrder) {
        actual = pqMax.getTop();
    } else {
        actual = pqMin.getTop();
    }
}
```

Conclusiones:

En general la búsqueda y entendimiento del árbol binario fue lo que más tiempo tomó, habían bastantes problemas sobre entender su funcionamiento y el orden de este, por lo que durante la realización del código se hicieron esquemas, gráficos y dibujos que facilitaron la comprensión correcta de este. Una vez que se entendió MaxHeap y MinHeap la realización y corrección de los métodos fue más sencilla de llevar a cabo, en ciertas ocasiones un pequeño error de sintaxis podía cambiar todo el resultado, por lo que se hicieron varia pruebas con distintas sintaxis para poder llegar a utilizar la forma correcta que entregará el resultado deseado.

Referencias:

Graficadora de árbol binario:

- https://visualgo.net/en/heap

Videos que permitieron el correcto entendimiento del código:

- (246) Heap sort en 4 minutos - YouTube
- (246) Data Structures: Heaps - YouTube
- (246) Heap - Build Max Heap - YouTube