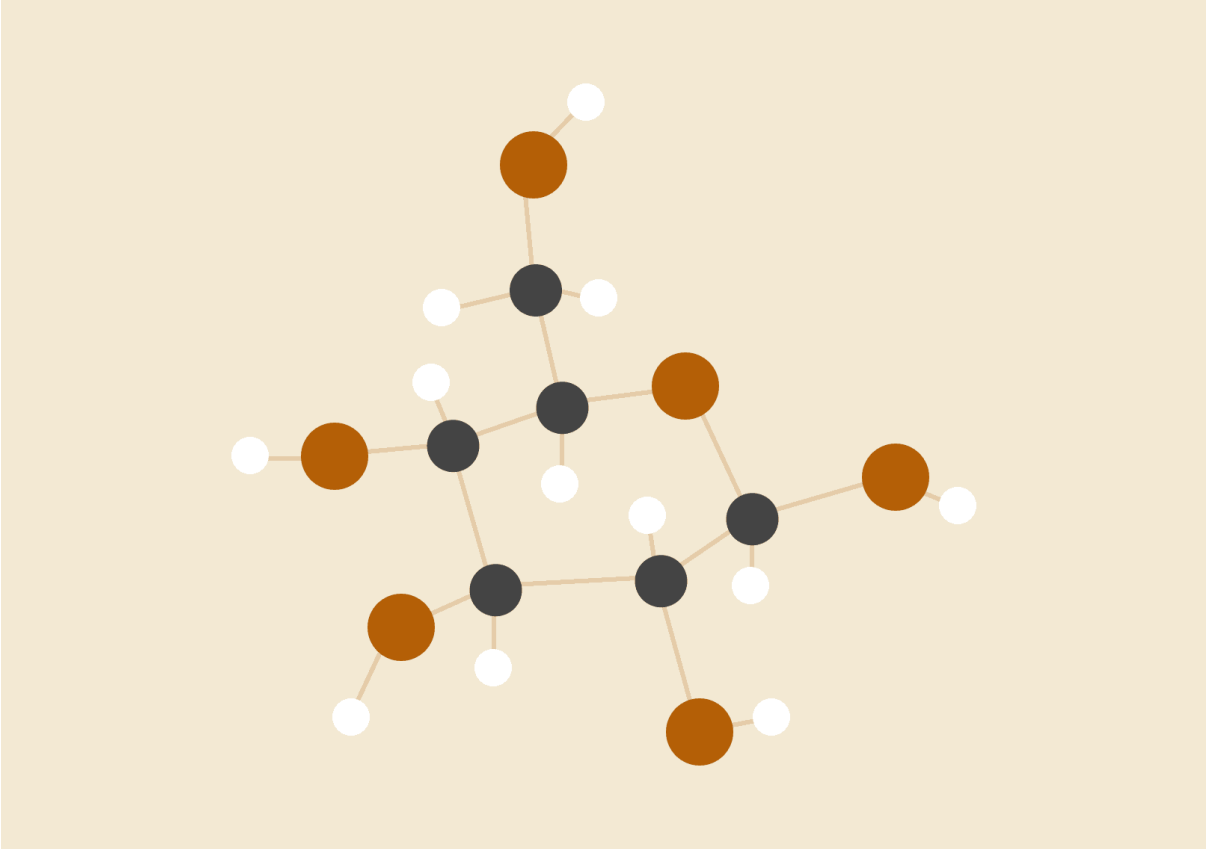
# Algoritmos y Estructuras de Datos

*Heaps y HashMaps*



**Christian Salemme, Pilar García, Lola Díaz, Germán Kleinubing**

06/22/2023
UADE; Ingeniería Informática; Programación II

# Índice

# ***Introducción***

En este informe desarrollaremos sobre dos estructuras de datos llamadas HashMaps y Heaps. Los HashMaps son útiles a la hora de almacenar y recuperar datos de forma rápida a través de claves y valores. En cambio, los Heaps son prácticos para organizar información en forma de árbol binario completo.

En primer lugar, introduciremos la lógica básica de un HashMap, para comprender la misma también debemos adentrarnos en la definición de las funciones de Hashing, buckets y colisiones. Estos son los conceptos más importantes para entender el funcionamiento de esta estructura de datos.

En segundo lugar, definiremos el concepto de Heaps, específicamente su aplicación y función. Luego, explicaremos de qué forma operan los métodos “agregar” y “eliminar” necesarios para su implementación. Finalmente nombraremos los costos de estas funciones.

Por último, junto al informe incluiremos un programa con la implementación de los Heaps y el tipo de dato abstracto “Colas con prioridad” vista durante la cursada de la materia.

## Desarrollo

En esta sección se examinará detalladamente las estructuras HashMaps y Heaps. Principalmente se explicará su funcionamiento y cómo están formadas. Posteriormente se detallarán características y conceptos importantes sobre cada una.

## HashMaps

Un HashMap es una estructura de datos no ordenada que sirve para almacenar datos compuestos por una única clave y uno o diversos valores. La estructura en donde se guardan las claves y valores se la conoce como “bucket”.

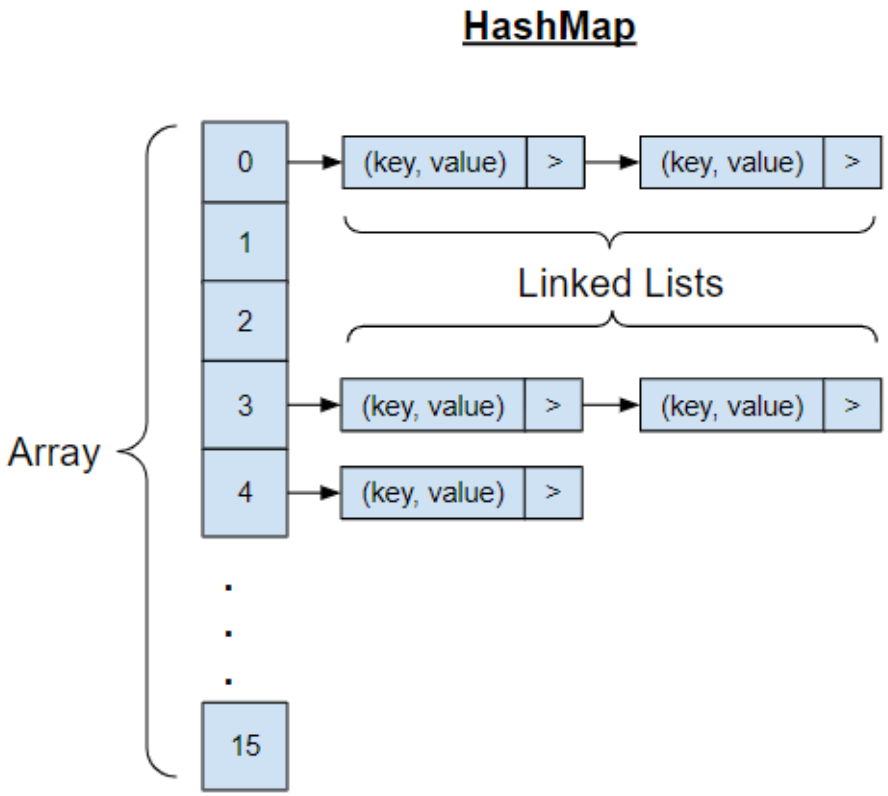
En relación a lo dicho anteriormente, la forma de acceso a los valores es a través de transformar la clave del elemento buscado en un número referente para utilizar como índice de nuestra clave. Dependiendo del caso, este índice puede ser único para cada elemento incluido en el HashMap o no. A esta técnica se le conoce como “función de hash” o “hashing”. Un ejemplo de esto sería obtener el código ASCII de cada carácter de la clave, sumarlos y dividirlos por la cantidad de espacios que el arreglo hashmap contiene. El resultado de esta operación resultaría como el índice del arreglo donde vamos a almacenar nuestra clave.

Por otro lado, una colisión toma lugar dentro de un HashMap cuando se quiere agregar una clave nueva en un índice en el que ya reside una clave, en conjunto con sus respectivos valores. Esto ocurre cuando el valor resultante del hashing es el mismo para dos claves o más.

Para solucionar el problema de la colisión existen varios métodos que podemos implementar. Uno de ellos es agregar la clave en el siguiente espacio libre del HashMap. El cual se va a poder encontrar, mediante una búsqueda lineal que en principio recorre desde el índice calculado hasta el fin de la lista. En caso de no haber encontrado ninguno,

continúa la búsqueda lineal pero esta vez desde el principio. Claramente esta operación tiene O(n).

Otra opción es convertir nuestra lista estática de HashMap en una lista dinámica. En caso de tener un índice repetido simplemente agregamos otro nodo al bucket del índice conectado al previo, resultando así en un bucket compuesto por dos o más nodos, cada uno con su clave y valores. Esta operación en el mejor de los casos tiene costo O(1), sino O(n). De todas formas sigue siendo más eficiente que el método anterior




Cada elemento (key, value) de la lista enlazada es un bucket.

La tercera opción para evitar colisiones, considerada también la más eficiente ya que nunca genera valores repetidos, consiste en modificar el cálculo del valor de la clave. Una modificación posible es la desarrollada a continuación. Primero se multiplica cada uno de los valores ASCII de la clave por su índice en la palabra. Finalmente la sumatoria de todos los valores obtenidos divididos por el tamaño del hashmap es el índice de la clave, y así

aumentamos la complejidad del hashing y de que ocurran nuevas colisiones. El costo de este método es O(1).

| | | | | | | | Resultado | resultado % cant de posiciones |
|---|---|---|---|---|---|---|---|---|
| caractér | m | a | r | c | o | s | | |
| valor | 109*1 | 97*2 | 114*3 | 99*4 | 111*5 | 115*6 | 2286 | 2286 % 13 = 11 |
| caractér | s | o | c | r | a | m | | |
| Valor | 115*1 | 111*2 | 99*3 | 114*4 | 97*5 | 109*6 | 2220 | 2220 % 13 = 10 |

En el ejemplo vemos que si cambiamos el orden de una sola letra, el resultado será distinto, si usaramos simplemente el valor ASCII, la suma de los valores de las letras sin importar el orden en el que se encuentren siempre daría igual.

Es importante considerar que tenemos la opción de agrandar la lista aproximadamente un 30% reduciendo así la posibilidad de que ocurra una colisión. Lo ideal es combinarla con cualquiera de los métodos mencionados anteriormente.

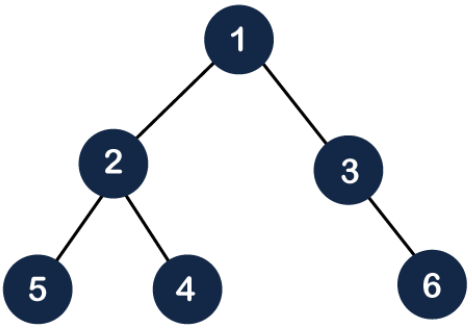
## ***Heaps***

Los Heaps son una estructura de datos que se visualiza como un árbol binario y siempre puede representarse como un arreglo. Además, en esta estructura los árboles tienen la particularidad de que se agregan nodos primero en el sub-árbol izquierdo y luego en el derecho, manteniendo el árbol binario lo más balanceado posible.

Existen dos tipos principales de Heaps, estos son:

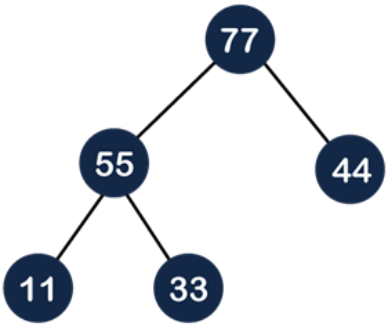
- ***Min Heap:*** El valor del nodo padre siempre tiene que ser menor o igual al valor de sus hijos. De esta forma el nodo de menor valor se encuentra ubicado en la raíz del árbol.

  Ejemplo de Min heap:




- ***Max Heap***: El valor del nodo padre siempre tiene que ser mayor o igual al valor de sus hijos. Al contrario que en los Min Heaps, en la raíz se encontrará el nodo de mayor valor.

  Ejemplo de Max Heap:



Como fue mencionado anteriormente la estructura heap puede ser representada como un arreglo. En principio, para ordenar el arreglo el valor asociado a la raíz se almacena en el primer elemento del arreglo con índice 1, el 0 no se utiliza. Continuando con el resto del árbol, se almacenan los valores de los hijos de la raíz siempre leyéndolos de izquierda a derecha. Quedando así asociado al elemento del arreglo con índice 2 el valor del hijo izquierdo y al elemento con índice 3 el valor del hijo derecho. Posteriormente se repite el proceso con los valores ubicados en la profundidad 3 del árbol y así sucesivamente hasta llegar al final del árbol.

Aplicando lo mencionado a un ejemplo, un arreglo del ejemplo del Max Heap quedaría de la siguiente manera:

| Índices | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Valores | 77 | 55 | 44 | 11 | 33 |

También, como en toda estructura estática se crean dos variables utilizadas como puntero, una que llamaremos tamaño e indicará la cantidad de valores que tenemos en el heap. Y la segunda será una variable que hará referencia a la cantidad máxima de valores que va a poder tener el heap.

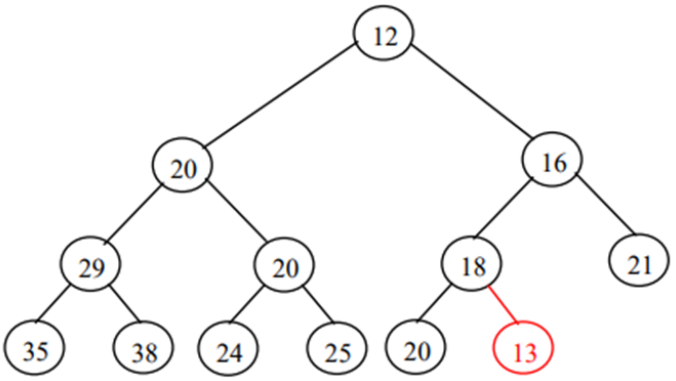
Teniendo en cuenta todo esto, para averiguar los índices de los elementos en un heap podemos establecer las siguientes propiedades:

- Si un nodo con índice *i* tiene hijo izquierdo, entonces el índice de ese hijo izquierdo será: 2**i*
- Si un nodo con índice *i* tiene hijo derecho, entonces el índice de ese hijo derecho será: 2**i* +1
- Si un nodo con índice *i* no es la raíz del árbol, entonces el índice de su padre será: *i*/2 (división de enteros)
- El nodo con índice *i* existe siempre y cuando *i* sea mayor o igual a 1 e *i* sea menor a la variable tamaño.

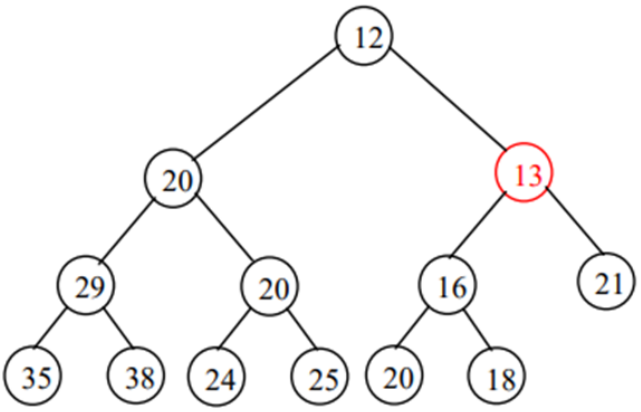
***Inserción de un elemento en un heap:***

En esta estructura de datos los elementos se agregan siempre al final, es decir en la posición n-ésima, luego revisando la propiedad del heap ya sea un min heap o al contrario un max heap, se compara al elemento recién agregado con su padre y en caso de necesitarlo, el elemento intercambia de posición con su padre. En caso de que el elemento tenga una nueva posición y esta no sea la de raíz del árbol, se repite el proceso de comparación con su padre e intercambio si es necesario.

Para entender esto mejor vamos a ejemplificar con una simple imagen.




La figura muestra que se desea agregar el elemento con valor 13, para esto se lo coloca en la última posición. A continuación se le realiza la respectiva comparación con su padre teniendo en cuenta de cumplir con la organización de este árbol, es decir que se cumpla con las propiedades de un min heap. Al realizar esto, se aprecia que el valor del elemento recién agregado es menor que su padre, por lo tanto se les intercambiara la posición. Como el elemento tiene una nueva posición, se le debe volver aplicar el proceso de comparación, en este caso se aprecia que el valor del padre del elemento sigue siendo mayor que el valor del propio elemento, esto significa que hay que volver a realizar un intercambio de posición. Para finalizar, como ocupa un nuevo lugar en el árbol, se compara una vez más con su padre, en este caso ya es la raíz, pero a diferencia de las veces anteriores, ya no es mayor y no hay que realizar un intercambio. De esta manera la figura ejemplificada quedaría de la siguiente manera:




De esta manera el árbol ya estaría correctamente ordenado y cumpliría con las propiedades de un min heap.

Realizar una inserción de esta manera tiene un costo de $O(\log_2(n))$.

***Remover un elemento en un heap:***

Primero y principal debemos tener en cuenta que en esta estructura de datos el elemento a remover siempre es el que se encuentra en la raíz del árbol, debido a esto el primer paso que debemos realizar consiste en comparar los valores de sus hijos y reemplazar a la raíz por uno de ellos dependiendo de las propiedades que hemos decidido aplicarle a nuestro heap, ya sea un min heap o un max heap. Una vez que la raíz tiene valor nuevo, debemos repetir el proceso de comparación y reemplazamiento en la posición del hijo de la raíz el cual tenga el valor que hemos tomado como nuevo valor de la raíz. Estos procesos serán repetidos hasta llegar a una hoja del árbol, una vez pasa esto se elimina el valor utilizado y en caso de tener elementos relacionados con índices posteriores, los elementos posteriores se desplazan un índice a la izquierda, logrando así reordenar el árbol para cumplir con la propiedad del heap. Debido a esta última parte, el costo de la operación es lineal, o dicho de otra manera O(n).

Finalmente, luego de haber investigado sobre los Heaps, creamos un programa donde implementamos los métodos de esta estructura de datos para poder visualizar mejor el funcionamiento de cada uno de sus métodos. En síntesis, decidimos guardar los valores en

una cola pero siguiendo el formato de orden de un árbol binario completo. Luego, implementamos a nuestro programa las Colas con prioridad, en el cual lo único que tuvimos que cambiar del programa sin esta nueva implementación fue crear una clase que se llame “Heap". Dentro de ella se encuentran los arreglos de valores y prioridades. Además, en los métodos de “Agregar” y “Eliminar" ahora para poder realizar estas acciones tuvimos que considerar primero las prioridades de cada valor en lugar del valor en sí.

Estas son las interfaces que usamos para implementar una cola con prioridad utilizando heaps.

```
package Interfaces;
public interface HeapTDA {
	void InicializarHeap();
	void Agregar(int x, int v);
	void Eliminar();
	int HijoIzq(int pos);
	int HijoDer(int pos);
	int Padre(int pos);
	boolean HeapVacio();
	ColasTDA prioridad();
	ColasTDA elementos();
}
```

Para ver sus respectivas implementaciones y desarrollo dirigirse al código adjunto a este documento.

## ***Conclusión***

En conclusión, a lo largo de este trabajo investigamos acerca de dos nuevas estructuras de datos y descubrimos que se caracterizan por conceder acceso rápido a la hora de buscar valores en comparación a las estructuras que vimos a lo largo del curso, tales como diccionarios o listas enlazadas. Estos métodos resultaron en una disminución y mejora en los costos, pasando en su mayoría de ser O(n) a O(1) o O(log2(n)).

Por un lado, los HashMaps nos permiten organizar datos de manera eficiente utilizando una única clave para cada valor o valores. También proporciona técnicas que agilizan la recuperación y búsqueda de datos, como por ejemplo la función hash que genera valores de índice en base a la clave del bucket que deseamos almacenar.

Por otro lado, la aplicación e investigación de los Heaps implementados junto a las Colas con prioridad nos proporcionaron una visión más completa del funcionamiento de ordenamiento de los valores y las prioridades con las que contaba cada uno. Especialmente esta estructura nos permite acceder al elemento o elementos de mayor prioridad sin tener que recorrer toda la lista de datos.

# Bibliografía


https://www.youtube.com/watch?v=9tZsDJ3JBUA
https://www.javatpoint.com/java-map#:~:text=A%20map%20contains%20values%20on,the%20basis%20of%20a%20key.
https://www.javatpoint.com/working-of-HashMap-in-java
https://www.javatpoint.com/java-integer .
http://profesores.elo.utfsm.cl/~tarredondo/info/datos-algoritmos/c8.pdf
https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture4.pdf