



Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC3001 – Algoritmos y Estructura de Datos

TDA para Medianas

TAREA 4

Nombre Alumno : Gabriel Iturra Bocaz
Correo : gabrieliturrab@ug.uchile.cl
Profesor : Patricio Poblete
Profesor Auxiliar : Manuel Cáceres
Sergio Peñafiel
Fecha : 23 de Mayo de 2016

Índice

I. Introducción	1
II. Análisis del Problema	2
<i>II.1. Descripción del Problema.....</i>	<i>2</i>
<i>II.2. Estrategia de Solución</i>	<i>3</i>
<i>II.3. Supuestos Importantes.....</i>	<i>3</i>
III. Solución al Problema	4
<i>III.1. Implementación.....</i>	<i>4</i>
<i>III.1.2 TDA MAX_HEAP</i>	<i>4</i>
<i>III.1.3 TDA ColaPrioridadMediana</i>	<i>6</i>
IV. Modos de Uso	8
V. Resultados	9
VI. Discusión.....	10

I. Introducción

El siguiente informe detalla el desarrollo de la Tarea 4, el curso del Algoritmos y Estructuras de Datos (3001). Como cuarta tarea del curso, se pretende que el alumno se interiorice con el uso de Tipo de Datos Abstractos (TDA), implementándolo mediante otro TDA conocido, que son las Colas de Prioridad (Priority Queue), y además de generar su interfaz de uso.

El objetivo es esta tarea es implementar un programa que obtenga la mediana de un rango de datos en tiempo constante, utilizando el supuesto que los datos son ingresados dinámicamente por el usuario, y estos son solo números pertenecientes al conjunto de los enteros positivos.

Para ello, se utilizó dos colas de prioridad, donde una de las colas de prioridad se almacena los elementos menores o iguales a la mediana en un MAX HEAP, y los elementos mayores o iguales en un MIN-HEAP (TDA's implementados con arreglos). Si la diferencia de la cantidad de datos en las colas de prioridad era mayor 1, se debía un hacer una balance para que la diferencia no fuese mayor a 1, en el cual se quitaba un elemento de la cola con mayor cantidad de datos para ser almacenada en la otra.

II. Análisis del Problema

II.1. Descripción del Problema

En muchos problemas de la vida diaria el uso de estadísticos es útil para al momento de tomar decisiones en conjuntos de una gran cantidad de datos. El estadístico más utilizado es la media, sin embargo esta es muy susceptible a errores (como un valor que sea muy grande en comparación al resto), por esto, un mejor estadístico a utilizar es la mediana, donde se elige un representante medio del conjunto.

Para encontrar dicho representante, se implementó un TDA donde los datos son ingresados dinámicamente, es decir, desde entradas del teclado que son leídas de una por línea, y cuando el usuario decide no seguir enviando valores al conjunto de datos, se entrega la mediana de estos.

Usando el TDA descrito, se pedía un programa que calculará los subsidios de viviendas usando la mediana de los datos, donde el subsidio sería otorgado a las familias con menores o iguales ingresos a la mediana de los postulantes (en este caso los datos ingresados por el usuario).

II.2. Estrategia de solución

Para resolver el problema propuesto, como se mencionó anteriormente se utilizó dos Colas de Prioridad, usando específicamente Heap Binarios (Árbol Binario especial que permite su almacenamiento en arreglos, utilizando condiciones especiales que varían de acuerdo al tipo de Heap, máximo o mínimo en las raíces del árbol y sus subárboles).

En una de las colas, se almacenó los valores mayores o iguales a la mediana en un MAX-HEAP, y los elementos menores a ésta en un MIN-HEAP.

Notar que si ambas colas están balanceadas según el número de elementos y la cantidad total de elementos es impar, entonces necesariamente una cola de prioridad tiene 1 elemento más que la otra, y justamente el elemento que está en el tope de esa cola es la mediana. Si la cantidad de elementos es par, entonces los elementos en los topes de ambas pilas son los centrales, por lo tanto la mediana se calcula como el promedio de estos. Luego, para que todo esto funcione el alumno debió decidir a cual (de las colas) insertar los elementos (En este caso fue MIN-HEAP), sin embargo se llega a un punto en que los Heap quedan desbalanceados por lo que luego de insertar, se debe asegurar que la diferencia de los largos sea a lo más 1, si no se cumple se debe eliminar un elemento del Heap con mayor cantidad de datos e insertarlo en el otro. En la figura 1 y 2, se aprecia un ejemplo de los Heap mencionados (Max y Min).

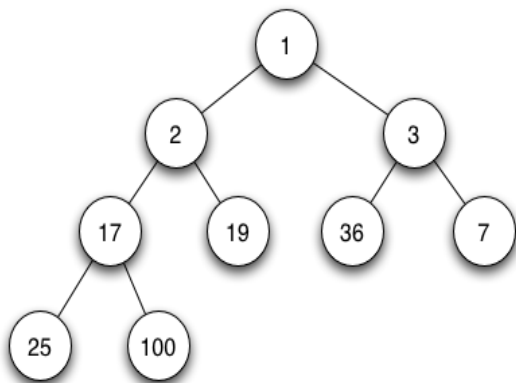


Fig. 1: Min-Heap.

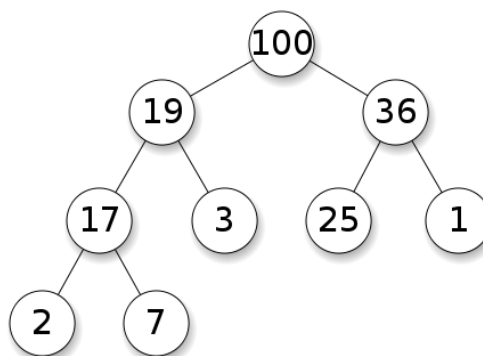


Fig. 2: Max-Heap.

II.3. Supuestos Importantes

Para la implementación y consideración del problema, se pueden destacar:

- 1) El usuario, solo ingresará números reales, y éstos serán pertenecientes al grupo de los reales positivos.
- 2) La implementación de los Heap fue realizada usando dos TDA distintos (uno para cada Heap), dentro del cual se utilizaron arreglos, siguiendo la relación que para un nodo j , su hijos serían $2*j$ y $2*j+1$, siendo j la posición dentro del arreglo.
- 3) Se asignó una capacidad de 101 a cada de arreglo de los heap (donde la posición 0 no tiene elementos), si se intentase insertar un elementos 101 al Heap se cierra el programa, mandando un mensaje de capacidad excedida.

III. Solución al Problema

III.1. Implementación

La implementación del TDA pedido, puede separarse en 3 partes, que son los TDA's y sus respectivos métodos que serán explicados en detalle a continuación:

- MAX_HEAP.
- MIN_HEAP.
- ColaPrioridadMediana (este último es el pedido en la tarea).

Los TDA's MAX_HEAP y MIN_HEAP son análogos en toda su estructura, salvo unas pequeñas diferencias en los métodos insertarElemento(double x) y extraerMax (o Min), esta sección solo se explicará en detalle el TDA MAX_HEAP, y las diferencias con el TDA MIN_HEAP serán explicadas en la siguiente sección.

Como los TDA's MAX_HEAP y MIN_HEAP hacen funcionar a ColaPrioridadMediana, se explicará primero el funcionamiento de éstos.

III.1.2 TDA MAX_HEAP

El TDA MAX_HEAP es un caso particular de un TDA de Cola de Prioridad, cuyo funcionamiento se basa en los Max Heap, donde en la raíz se guardan el elemento mayor de los datos, siguiendo la misma regla para los subárboles, como la implementación los elementos se realiza mediante un arreglo usando la regla descrita en la sección Supuestos Importantes. Esta clase posee 3 atributos, un arreglo de elementos tipo double donde se guardan los datos que se van insertando, un elemento int numElem, que sirve para contar los elementos dentro del arreglo, y un elemento tipo int cap, que toma un valor de 101 que determina la capacidad máxima del arreglo, a continuación se explica sus respectivos constructores y métodos:

- public MAX_HEAP(): este método crea un nuevo objeto MAX_HEAP, en el cual se define un arreglo double con 101 casilleros para almacenar datos, numElem, toma valor 0, ya que inicialmente no hay datos. En la figura 3, se aprecia en el código.

```
public MAX_HEAP() {
    arreglo = new double[cap];
    numElem = 0;
}
```

Figura 3: Constructor MAX_HEAP.

- `public MAX_HEAP(int cap)`: otro constructor de la clase, con la diferencia que el recibe un elemento `cap` tipo `int`, donde ese se le asigna la capacidad al arreglo manualmente.
- Método `public boolean estaVacia()`: retorna `true`, si el objeto ésta vacío, falso sino.
- Método `int public largo()`: devuelve la cantidad de elementos dentro del objeto.
- Método `double getMax()`: devuelve el elemento mayor dentro del conjunto, y 0 si el objeto está vacío.
- Método `void insetarElemento(double x)`: sí el objeto ésta vacío le asigna al elemento `x` al casillero número 1 y `numElem` toma el valor 1, y sí el objeto no está vacío le asigna el casillero siguiente al elemento `x`, pero sí este es mayor a los elementos insertados anteriormente, se entra a un bucle `for` donde los elementos son cambiados y reordenados para no perder la condición del Max Heap, mediante el juego de índices en que el elemento en la posición `j` debe ser mayor a los elementos en las posiciones `2*j` y `2*j+1`, cuyo costo es $O(\log n)$. Sí se inserta un elemento cuanto el objeto está lleno, se despliega un mensaje error, “Capacidad excedida” y se cierra el programa. En la figura 4 se aprecia parte del código de este método.

```
if (estaVacia()){
    arreglo[++numElem] = x;
}
else{
    arreglo[++numElem] = x;
    for (int j = numElem ; j>1 && arreglo[j]>arreglo[j/2]; j/=2){
        double t = arreglo[j];
        arreglo[j] = arreglo[j/2];
        arreglo[j/2] = t;
    }
}
```

Figura 4: Inserción de datos.

- Método `double extraerMax()`: extrae el elemento mayor dentro del conjunto, eliminándolo y retornándolo. Para no perder la condición de Max Heap, el siguiente elemento mayor debe ocupar su lugar lo que implicar un reordenamiento de los elementos. Los índices se van recorriendo a través de un ciclo `while`, desde la posición dos comparando los valores de una posición `k`, con las posiciones `2*k` y `2*k+1`, donde los elementos en las últimas posiciones son mayores al elemento en la posición `k` se intercambian, mediante una variables auxiliar `t`, cuando el elemento en la posición 1 es el mayor se corta el ciclo. En la figura 5, se aprecia parte del código de este método.

```

while (2*j <= numElem){
    int k = 2*j;
    if(k+1<=numElem && arreglo[k+1]>arreglo[k]){
        k++;
    }
    if (arreglo[j]>arreglo[k]){
        break;
    }
    double t = arreglo[j];
    arreglo[j] = arreglo[k];
    arreglo[k] = t;
    j = k;
}

```

Figura 5: Extraer Máximo elemento.

*** El TDA MIN_HEAP es análogo al TDA explicado.

III.1.3 TDA ColaPrioridadMediana.

Este TDA es solicitado formalmente dentro de la tarea, su funcionamiento se realizado mediante los TDA's MAX_HEAP y MIN_HEAP, y es aquí donde se realizar el cálculo de mediana de los datos ingresados, tiene dos parámetros un max_heap de la clase MAX_HEAP y otro min_heap de la clase MIN_HEAP. Los valores menores a la mediana son almacenados en los objetos MAX_HEAP y los mayores en objetos MIN_HEAP, luego a través de los métodos explicados en el punto anterior se definen los métodos de este nuevo TDA. A continuación se explican los constructores y sus respectivos métodos:

- Constructor ColaPrioridadMediana(): se inicializa un nuevo objeto de la clase, en el cual sus parámetros toman nuevas inicializaciones de sus respectivas clases, con objetos vacíos (sin elementos).
- Constructor ColaPrioridadMediana(int cap): misma funcionalidad que el primer constructor, salvo que a cada Heap que se inicializa toma una capacidad de almacenamiento de cap/2.

```

public ColaPrioridadMediana() {
    max_heap = new MAX_HEAP();
    min_heap = new MIN_HEAP();
}

```

Figura 6: Constructor ColaPrioridadMediana.

- Método void insertar(double x): inserta nuevos elementos a los Heap, si un nuevo objeto de nuestra clase está vacío el primer elemento se inserta a min_heap, luego si los próximos elementos insertados son mayores al primer elemento estos se vuelven a insertar dentro de min_heap, utilizando el método de inserción de ese

objeto, en caso contrario se inserta dentro de max_heap. Sin embargo, como bien se dijo, si min_heap posee dos elementos más que max_heap, mediante el método extraerMin() que extrae el elemento menor de min_heap (de la clase MIN_HEAP) se extrae el elemento menor y se inserta dentro de max_heap, en caso contrario, se extrae el elemento mayor de max_heap y se inserta en min_heap utilizando una forma análoga, este método tarda $O(\log n)$, siendo n el número de elementos totales en los heap.

```
public void insertar(double x){
    if ( min_heap.largo() == 0 || x>=min_heap.getMin() ){
        min_heap.insertarElemento(x);
    }
    else{
        max_heap.insertarElemento(x);
    }
    if ( min_heap.largo() - max_heap.largo() > 1){
        max_heap.insertarElemento(min_heap.extraerMin());
    }
    else if ( max_heap.largo() - min_heap.largo() > 1 ){
        min_heap.insertarElemento(max_heap.extraerMax());
    }
}
```

Figura 7: Método de Inserción de elementos.

- Método double getMediana(): devuelve el valor de la mediana del conjunto de datos que se encuentre dentro del TDA. Si el número de datos es un número impar, significa que min_heap o max_heap posee un elemento más en comparación al otro, dependiendo de cual sea, mediante los métodos getMax o getMin se devuelve el valor de la mediana, en cambio si el número de elementos del objeto es par, se suma los elementos que se obtienen con los métodos dichos anteriormente y se dividen por 2, cuyo valor es la mediana. Este método tiene un costo $O(1)$. En la figura 7, se ve parte del código.

```
if ( max_heap.largo() == min_heap.largo() ){
    mediana = (min_heap.getMin() + max_heap.getMax())/2.0;
}
else{
    if (min_heap.largo() > max_heap.largo()){
        mediana = min_heap.getMin();
    }
    else{
        mediana = max_heap.getMax();
    }
}
```

Figura 8: Método para obtener la mediana.

IV. Modos de Usos

IV.1 TDA MIN_HEAP

Siendo análogo en MAX_HEAP, tanto en implementación y sus métodos, es importante destacar la diferencia es dos métodos importantes, que son:

- void insertarElemento(double x): al igual que el método de MAX_HEAP, inserta elementos dentro del objeto, pero al insertar el segundo o los posteriores elementos estos son ordenados de forma que en una posición j , el elemento sea menor en comparación a los elementos en las posiciones $2*j$ y $2*j+1$, este método también tiene un costo $O(\log n)$.
- double extraerMin(): misma funcionalidad que extraerMax(), pero extraer el valor mínimo del conjunto, y reordena los elementos de forma que no se pierda la condición de Min Heap, y también tiene un costo $O(\log n)$.

****El resto de métodos poseen la misma funcionalidad, salvo getMin() que es análogo a getMax().*

IV.2 Cálculo de Subsidios.

Usando el TDA pedido, se creó el programa que pide al usuario los números para ingreso familiar por la entrada estándar, si usuario ingresa vacío se despliega un mensaje que retorna la mediana de los datos, con el monto máximo de beneficiarios del subsidios. Para programar esto, se definió un objeto `m` de la clase `ColaPrioridadMediana`, un objeto `scn` de la clase `Scanner` (para pedir datos al usuario) y un String `s` nulo, luego utilizando el bucle `while`, se pide al usuario que ingrese los datos necesarios partiendo de los supuestos planteados en las secciones anteriores. En la variable `s`, se guardan los valores que ingresa el usuario en forma de String, si son números, estos son parseados de String a double y son insertados en el objeto `m`, finalmente cuando el usuario ingresa vacío y pulsa “Enter”, se rompe el invariante del ciclo, donde la variable `s` es distinta de vacío (“”), y se despliega el texto máximo de valor de subsidio, todo dentro de la función `void main(String[] args)`. Parte del código se puede apreciar en la figura 9.

```
do {
    System.out.println(">>");
    s = scn.nextLine();
    if (s.equals("")==false){
        subsidio=Double.parseDouble(s);
        if (subsidio<0){
            System.out.print("Solo puede ingresar valores positivos.");
            System.exit(-1);
        }
        m.insertar(subsidio);
    }
}

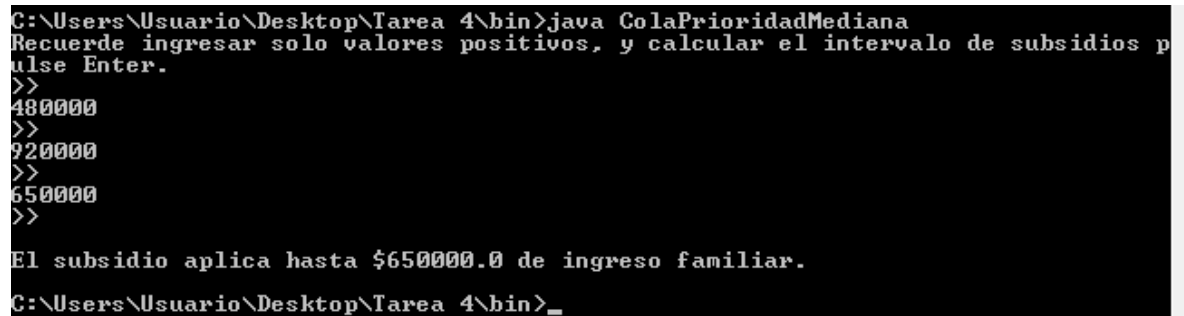
while(s.equals("")==false);
scn.close();
System.out.println("El subsidio aplica hasta $" +m.getMediana()+" de ingreso familiar.");
,
```

Figura 9:Uso del TDA.

V. Resultados.

En la siguiente sección se presentan distintos ejemplos del uso del programa:

- En la figura 10, puede verse el ejemplo del enunciado (desde el cmd):



```
C:\Users\Usuario\Desktop\Tarea 4\bin>java ColaPrioridadMediana
Recuerde ingresar solo valores positivos, y calcular el intervalo de subsidios p
ulse Enter.
>>
480000
>>
920000
>>
650000
>>
El subsidio aplica hasta $650000.0 de ingreso familiar.
C:\Users\Usuario\Desktop\Tarea 4\bin>_
```

Fig. 10.

- En la figura 11, se ve un ejemplo con una mayor cantidad de datos.

```

>>
7000
>>
9000
>>
10000
>>
4000
>>
5000
>>

El subsidio aplica hasta $7000.0 de ingreso familiar.

```

Fig. 11

- En la figura 12, se ve un ejemplo cuando se sobrepasa la capacidad de los arreglos en las distintas clases.

```

-----
>>
60000
>>
4000
>>
100000
|Capacidad Excedida

```

Fig. 12.

VI. Discusión

VI.1. Complejidad Métodos del TDA.

Como los el TDA ColaPrioridadMediana, está hecho en base a los TDA's MAX_HEAP y MIN_HEAP, sus dos métodos principales están implementados en base a los métodos de estos dos heap, analizaremos los dos métodos:

- void insertar(double x): este método inserta elementos al objeto de la clase, el primer elemento si inserta a min_heap, si los elementos posteriores que se van insertando son mayores estos van a min_heap, sin embargo puede que al insertando se rompa la condición de min heap, pero afortunadamente con el método insertarElemento(double x) a medida que se insertan los elementos se van reordenando, esto tiene un costo de $O(\log n)$ (ésta en el apunte), en caso contrario si los elementos que se agregan son menores al primero, estos van a max_heap que se insertar con el método análogo también de costo $O(\log n)$. En caso de que ir al agregando elementos los heap queden una diferencia mayor a 1 (en cuanto a la

cantidad de elementos) se debe extraer un elemento e insertarlo dentro del otro, usando los métodos `extraerMin()` o `extraerMax()`, cuyos costos también son $O(\log n)$, lo que finalmente implica que los elementos quedan balanceados, ya sea con una diferencia de un elemento entre ambos o la misma cantidad, teniendo costo $O(2\log n)$ que es lo mismo que $O(\log n)$.

- `double getMediana()`: debido a cómo quedan los elementos dentro de los heap, si `max_heap` tiene un elemento más que `min_heap` se debe retornar el primer elemento del heap, este caso el mayor, lo mismo pasa si `min_heap` tiene un elemento más que `max_heap` se retorna el primer elemento del arreglo, el costo de esto es $O(1)$. Si los dos heap tienen la misma cantidad de elementos se retorna el primer elemento del arreglo de `max_heap` y `min_heap` se suman y dividen por 2, lo que también tiene un costo de $O(1)$. Como es un caso o el otro, se deduce que este método posee un costo $O(1)$.

VI.2. Fuerza Bruta

Si los elementos se van agregando por orden de entrada al arreglo, y luego son ordenados por algún algoritmo eficiente, en el peor caso esto puede tener un costo $O(n \log n)$ como en caso del Quicksort, Heapsort, entre otros. Luego de estar ordenando los elementos podemos decir que la mediana se ubica en la posición $n/2$, siendo n la cantidad de elementos, lo que tiene un costo $O(1)$, sin embargo, el costo de esto sigue siendo $O(n \log n)$. Lo que claramente es peor en términos de complejidad que la solución sugerida en la tarea, tiene un costo $O(\log n)$.