DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico 2

Diseño e implementación de estructuras

19 de octubre de 2025 — Algoritmos y Estructuras de Datos

BILS

Integrante	LU	Correo electrónico
Picard, Silvano	637/24	s.rene.picard@gmail.com
de Suto Nagy, Bautista	346/24	bdesutonagy@gmail.com
Romero, Ignacio	681/24	ignacion.romero@outlook.es
Avellaneda, Lautaro	41/22	lautaro.avellaneda@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción a la estructura

Para cumplir con las complejidades requeridas, utilizamos principalmente heaps para el manejo de traslados y ciudades, optimizando así la selección y eliminación de los traslados más rentables y antiguos. Los heaps se implementan mediante un arreglo redimensionable (usando la estructura `ArrayList<T>`) que proporciona un acceso $O(1)$ amortizado al agregar elementos al final, según se permite en este contexto. La estructura principal es una clase `BestEffort` que contiene los siguientes atributos privados:

- Dos heaps, `Redituables` y `Antiguos`, que almacenan los traslados de mayor rentabilidad y los más antiguos. Ambos heaps permiten un acceso $O(\log n)$ para inserciones y extracciones, cumpliendo así con las restricciones de complejidad en las operaciones de despacho.
- Un heap adicional, `Ciudades`, para la gestión de las estadísticas de ganancia y pérdida de cada ciudad.
- Dos listas `ArrayList`, `ciudadesMaxGanancia` y `ciudadesMaxPerdida`, que almacenan los identificadores de las ciudades con las mayores ganancias y pérdidas, respectivamente.
- Variables de control como `maxGanancia`, `maxPerdida GananciaTotal` y `despachados`, que facilitan el cálculo de estadísticas globales sin afectar la eficiencia de las operaciones principales.

La elección de estas estructuras está motivada por su capacidad para cumplir con las complejidades temporales deseadas, especialmente en las operaciones de acceso y eliminación en heaps, así como en la gestión de ciudades mediante listas redimensionables.

Especificación del Heap

Este `Heap` es una implementación genérica de un max-heap con modificaciones (1), diseñada para gestionar elementos con un orden específico proporcionado por un comparador (`Comparator<T>`). La estructura mantiene una relación entre los elementos y sus identificadores únicos o *handles*, permitiendo así el acceso constante a los elementos, incluso si sus posiciones cambian dentro del heap.

Se implementa usando:

- `vals` (un `ArrayList<`$Tupla < Integer, T >$`>`): Almacena las tuplas de identificadores (*handles*) y los elementos (`T`). La propiedad del max-heap garantiza que el elemento con mayor valor esté siempre en la raíz (índice 0).

```
1   public ArrayList<Tupla<Integer,T>> vals;
```

- `handles` (un `ArrayList<Integer>`): Permite acceder a la posición de cada elemento en `vals`, dado su identificador (*handle*).

```
1   public ArrayList<Integer> handles;
```

- `index_stack` (un `ArrayList<Integer>`): Una pila o stack donde guardaremos los index disponibles en `handles`. Nos ayuda a cumplir las caracteristicas que pediremos mas adelante y para tener mayor eficiencia en el uso de memoria.

```
1   private ArrayList<Integer> index_stack = new ArrayList<Integer>();
```

Invariante adicional

Ademas de los invariantes de un max-heap, nuestra implementacion cumple el siguiente invariante:

$$(\forall h : Heap)((\forall e : Tupla < int, T >)(e \in h.vals \rightarrow_l h.vals[h.handles[e.first]] = e)) \quad (1)$$

Con este invariante estamos implicando que no hay dos elementos con el mismo handle.

Análisis de Complejidad de los Métodos Públicos

- **Constructor** `Heap(T[] s, Comparator<T>comparator)`
  - **Complejidad**: $O(s)$, donde s es el tamaño del array de entrada.
  - **Justificación**:
    - La creación de `vals` y `handles` requiere $O(s)$ operaciones.
    - El primer bucle agrega cada elemento como una tupla en `vals` y su índice en `handles`, ambas operaciones de $O(1)$.

- El segundo bucle `shiftDown` inicializa el heap, ejecutándose $O(s)$ veces. 1
- El tercer bucle establece los índices en `handles` y toma $O(s)$ tiempo.

- **Método** `int raiz()`
  - **Complejidad**: $O(1)$.
  - **Justificación**: Accede al primer elemento de `vals`, devolviendo el *handle* del elemento raíz (máximo).
- **Método** `int insertar(T elem)`
  - **Complejidad**: $O(\log(v))$.
  - **Justificación**:
    - La inserción en `vals` y `handles` se realiza en $O(1)$.
    - Luego, `shiftUp` ajusta el heap para mantener la propiedad del max-heap, operando en $O(\log(v))$.
- **Método** `void remover(Integer handle)`
  - **Complejidad**: $O(\log(v))$.
  - **Justificación**:
    - Primero, intercambia el elemento a eliminar con el último en $O(1)$ usando `swap`.
    - Elimina el último elemento en $O(1)$.
    - Después, ajusta el heap con `shiftDown` y `shiftUp`, cada uno tomando $O(\log(v))$ en el peor caso.

Resumen de Complejidades

Método	Complejidad Temporal
`Heap(T[] s, Comparator<T>comparator)`	$O(s)$
`raiz()`	$O(1)$
`insertar(T elem)`	$O(\log(v))$
`remover(Integer handle)`	$O(\log(v))$

Aclaraciones sobre la estructura

Para poder cumplir las complejidades pedidas en el enunciado necesitamos que nuestra implementacion de heap tenga algunas particularidades. En primer lugar, necesitamos que dado el handle de un elemento en un heap podamos acceder al handle de ese mismo elemento en otro heap en tiempo constante o, a lo sumo, logaritmico. Es de utilidad al, por ejemplo, despachar el traslado mas antiguo, también termine siendo eliminado del heap de redituables. Esto lo logramos del siguiente modo: Como siempre insertamos o eliminamos elementos de los dos heaps en el mismo momento, procuramos que los handles esten dados por el orden en que son ingresados o eliminados los elementos del heap. De esta manera, si queremos insertar y eliminar elementos en el mismo orden, tendremos el mismo handle para éstos.

En segundo lugar, necesitamos poder modificar elementos del heap con el fin de poder modificar el superavit en nuestro heap de ciudades. Con el objetivo de mantener el encapsulamiento, solo permitimos acceder a los valores del heap al removerlos del mismo. Por lo que para modificarlo debemos sacarlo del heap, modificarlo y volverlo a insertar. De esta manera, no permitimos la modificacion de elementos desde el exterior de la estructura, asegurandonos un manejo seguro de la misma. Como insertar y remover son operaciones logaritmicas, podemos modificar el elemento deseado en tiempo logaritmico:

$$O(log(n)) + O(log(n)) = 2 \cdot O(log(n)) = O(log(n)) \text{ pues sus crecimientos asintoticos son los mismos} \tag{2}$$

Ademas necesitamos que, al realizar toda esta operacion el handle del elemento se mantenga. Para esto utilizamos el index_stack (1). Éste procura que el handle de un nuevo elemento sea el del ultimo elemento removido. De este modo se cumple que:

$$(\forall h : Heap)((\forall i : Handle)(i == h.insertar(h.remover(i)))) \tag{3}$$

Por ultimo, los handles de un nuevo elemento se eligen del siguiente modo: Si el `index_stack` está vacio, el handle del nuevo elemento será igual a la longitud de la lista `handles`, en caso contrario el handle sera igual al ultimo elemento de `index_stack`, es decir el handle del ultimo elemento removido. Al hacer esto se desapila ese handle del stack. Cuando se remueve un elemento su handle se apila en el stack. En el código se ve de la siguiente manera:

```
    int handle;
    if(index_stack.isEmpty()){
        handle = handles.size();
    }else{
        handle = handles.remove(handles.size() - 1);
    }
```

Complejidad temporal de Heapify

El constructor del Heap utiliza el algoritmo de *Heapify* para construir un Max-Heap a partir de una lista de elementos. Este proceso tiene una complejidad de $O(n)$, y aquí explicamos por qué.

```
   for (int i = s.length/2 - 1; i >= 0; i--){
            this.shiftDown(i);
    }
```

Algoritmo Heapify

El proceso de *Heapify* consiste en ajustar la propiedad del heap desde los nodos internos hacia la raíz. La cantidad de trabajo en cada nodo depende de su profundidad en el árbol.

Dado un heap con n elementos:

- La profundidad máxima del árbol es $\lfloor \log_2 n \rfloor$.
- Los nodos en la mitad inferior del árbol (las hojas) ya cumplen la propiedad del heap y no requieren trabajo adicional.
- Solo los nodos internos (de índice $\lfloor n/2 \rfloor - 1$ hasta la raíz) necesitan ser ajustados.

El tiempo que toma ajustar un nodo depende de la altura de su subárbol.Para un nodo en el nivel h (siendo la raíz el nivel 0), el costo máximo es proporcional a la altura h, ya que siftDown puede recorrer todo el camino hasta una hoja.

Análisis del Trabajo Total

Para calcular el costo total de *Heapify*, sumamos el costo de *shiftDown* para todos los nodos internos. Este costo disminuye exponencialmente a medida que los nodos están más cerca de las hojas (menos altura por recorrer).

La fórmula para el trabajo total es:

$$T(n) = \sum_{d=0}^{\lfloor \log n \rfloor} (\text{nodos en nivel d}) \cdot (\text{costo por nodo en nivel d})$$

Sustituyendo:

- $\lfloor \log_2 n \rfloor$ (altura): h.
- Número de nodos en el nivel d: 2^d.
- Costo por nodo (altura máxima desde ese nivel): h - d.

$$T(n) = \sum_{d=0}^{h} 2^d \cdot (h - d)$$

Reorganizando los términos:

$$T(n) = h \sum_{d=0}^{h} 2^d - \sum_{d=0}^{h} d \cdot 2^d$$

Remplazamos las sumatorias usando las formulas:

$$T(n) = h(2^{h+1} - 1) - (2 + (h-1) \cdot 2^{h+1})$$

Reorganizamos y cancelamos:

$$T(n) = h \cdot 2^{h+1} - (h-1) \cdot 2^{h+1} - h - 2$$
$$T(n) = 2^{h+1} - h - 2$$

Lo escribimos en términos de n:

$$T(n) = 2^{\log_2 n} - \log_2 n - 2 = n - \log_2 n - 2$$

Conclusión del análisis

El trabajo total para construir el Max-Heap es:

$$T(n) \in O(n)$$

Esto se debe a que la suma del trabajo en todos los niveles del árbol es proporcional a n, y no a $n \log n$, como podría esperarse si ajustáramos individualmente cada elemento.

Conclusión

En conclusión, el trabajo presentado describe un diseño acorde a lo pedido. El uso de heaps y el mantenimiento de invariantes clave garantizan un acceso rápido y seguro a los elementos, así como la posibilidad de manejar operaciones en tiempo logarítmico.

Tratamos de mantener la integridad de los datos sin sacrificar totalmente el costo temporal y de memoria. Por ultimo y no menos importante, priorizamos un manejo seguro y escalable (principalmente en memoria) de los datos con el uso de un stack de indices.