Análisis de Complejidad

Estructura de Datos

UNIVERSIDAD DE BURGOS

José Luis Garrido Labrador

José Miguel Ramírez Sanz

Índice de contenido

1. Método put

```
@Override
public V put(R row, C column, V value) {
	if(mapa.get(row)==null){
		mapa.put(row, new HashMap<C,V>());
	}
	elementos++;
	V supp = mapa.get(row).get(column);
	if(supp != null){
		elementos--;
	}
	return mapa.get(row).put(column, value);
}
```

Para este método consideramos para put de hashMap una complejidad media de O(log n) ya que no sabemos si tendrá que redimensionar a priori si nunca tuviese que redimensionar fuese de O(1) y si se redimensionaría siempre sería de O(n) porque tendría que recorrerse todo el mapa. Luego el método get tiene complejidad O(1) y cuando aplicamos put sobre el mapa interno tenemos nuevamente una complejidad de O(log m) (m porque los elementos de el mapa interno no tiene por que ser igual al del mapa externo). Por tanto al desconocer los valores de n y m en vez de aplicar la regla de la suma quedándonos con el valor máximo nos quedamos con la suma de la complejidad tanto de O(log m) como de O(log n) y la suma de los logaritmos es el logaritmo del producto. Por tanto nuestro método put para HashMapTable sería de O(log nm).

2. Método remove

```
@Override
public V remove(R row, C column) {
	V element = mapa.get(row).get(column);
	if(element!=null){
		elementos--;
		mapa.get(row).remove(column);
	}
	return element;
}
```

Nuevamente tenemos que get tiene complejidad de O(1) para los mapas hash y remove como vimos en la teoría de las tablas hash tiene también una complejidad de O(1) ya que no borra si no que marca el bloque como borrado. Por tanto tenemos en total una complejidad que es el sumatorio de 4 complejidades de O(1) y utilizando la regla de la suma nos quedamos con O(1) como complejidad de este método remove para HashMapTable.

3. Método get

```
@Override
public V get(Object row, Object column) {
	Map<C,V> mapaInterno = mapa.get(row);
	if(mapaInterno==null)
		return null;
	else
		return mapaInterno.get(column);
}
```

Para el método get de HashMapTable tenemos dos llamadas a los métodos get de dos HashTable que tienen complejidad de O(1), aplicando la regla de la suma nuestro método tiene una complejidad de O(1).

4. Método containsKeys

```
@Override
public boolean containsKeys(Object row, Object column) {
	if(mapa.containsKey(row))
		return mapa.get(row).containsKey(column);
	else
		return false;
}
```

Para este método utilizamos el método containsKey sobre un mapa hash, que a diferencia de otras implementaciones de Map este tiene una complejidad aproximada de O(1). Por tanto, continuando con la aproximación este método tendría una complejidad también aproximada de O(1) ya que como hemos dicho en apartados anteriores el método get sobre HashMap tiene también complejidad de O(1).

5. Método containsValue

```
@Override
	public boolean containsValue(V value) {
		for(Map.Entry<R, Map<C,V>> e: mapa.entrySet()){
			for(Map.Entry<C, V> v: e.getValue().entrySet()){
				if(v.getValue().equals(value)){
					return true;
				}
			}
		}
		return false;
	}
```

En este método tenemos un primer bucle con n iteraciones (O(n)) y luego un segundo bucle anidado con m iteraciones (valor máximo teniendo en cuenta la fila que tenga una mayor cantidad de columnas). Por tanto solamente con los dos bucles tenemos una complejidad de O(nm). Por otro lado la función getValue sobre un Entry y equals solamente tiene una complejidad de O(1). En conclusión el método containsValue de HashMapTable tiene una complejidad de O(nm).

6. Método row

```
@Override
public Map<C, V> row(R rowKey) {
	if(mapa.containsKey(rowKey))
		return mapa.get(rowKey);
	else
		return new HashMap<>();
}
```

En este caso tenemos dos complejidades ya analizadas y para evitar redundancias en el peor caso este método tiene una complejidad de O(1).

7. Método column

```
@Override
public Map<R, V> column(C columnKey) {
	Map<R,V> supp = new HashMap<>();
	for(Map.Entry<R, Map<C,V>> e : mapa.entrySet()){
		for(Map.Entry<C, V> v: e.getValue().entrySet()){
			if(v.getKey().equals(columnKey)){
				supp.put(e.getKey(), v.getValue());
			}
		}
	}
	return supp;
}
```

Para este método tenemos dos bucles anidados, uno que recorre todos los elementos de mapa (O(n)) y otro que recorre los elementos del mapa interno (O(m)) por tanto solo con el bucle tenemos O(nm). Luego teniendo en cuenta el peor caso en el que se recogen todos los datos (nm si tenemos en cuenta el peor caso de que todos los submapas tienen la cantidad de elementos mayor) el put sobre un mapa hash tendría una complejidad de log nm. Por tanto como cota superior tenemos que la complejidad de este método sería de O(nm log nm).

8. Métodos de la subclase Agrupación

```
private class Agrupacion implements Table.Cell<R, C, V>{

        private R rowa;
        private C columna;
        private V valuea;

        public Agrupacion(R row, C column, V value){
                rowa = row;
                columna = column;
                valuea = value;
        }

        @Override
        public R getRowKey() {
                return rowa;
        }

        @Override
        public C getColumnKey() {
                return columna;
        }

        @Override
        public V getValue() {
                return valuea;
        }

        @Override
        public V setValue(V value) {
                V anterior = put(rowa, columna, value);
                valuea=value;
                return anterior;
        }

}
```

Todos los métodos de esta subclase serían de O(1) salvo setValue que para mentener la integridad de los datos hace uso de un método put de la tabla y como vimos en el análisis de ese método tendría una complejidad de O(log nm).

9. Método cellSet

```
@Override
public Collection<es.ubu.lsi.edat.pract08.Table.Cell<R, C, V>> cellSet() {
        ArrayList<es.ubu.lsi.edat.pract08.Table.Cell<R, C, V>> salida = new
ArrayList<>(size());
        Agrupacion supp = null;
        for(Map.Entry<R, Map<C,V>> e : mapa.entrySet()){
                for(Map.Entry<C, V> v: e.getValue().entrySet()){
                        supp = new Agrupacion(e.getKey(),v.getKey(),v.getValue());
                        salida.add(supp);
                }
        }
        return Collections.unmodifiableCollection(salida);
}
```

Aquí tenemos dos bucles for que como hemos dicho en anteriores análisis tienen complejidad de O(nm). Luego el método add sobre un ArrayList para este caso es de O(1) ya que al inicializar de antemano este objeto con la dimensión actual de la tabla no necesita haber una redimención. Por tanto la complejidad de este metodo es unicamente de O(nm).

10. Métodos size, isEmpty y clear

```
@Override
public int size() {
        return elementos;
}

@Override
public boolean isEmpty() {
        return elementos==0;
}

@Override
public void clear() {
        mapa = new HashMap<>();
        elementos=0;
}
```

Para estos tres métodos la complejidad es de O(1) ya que los tres solo tienen acciones de esa complejidad sin anidar.