Análisis de complejidad temporal del InsertionSort.

```
insertionSort() {
        for (int i = 0; i < allClients.size(); i++) {
                Client aux = allClients.get(i);
                int j = i - 1;
                while (j > 0 && allClients.get(j).compareTo(aux) > 0) {
                        allClients.set(j + 1, allClients.get(j));
                        j = j - 1;
                }
        allClients.set(j + 1, aux);
        }
}
```

INSTRUCCIÓN	VECES QUE SE REPITE (Big O)
1. for (int i = 0; i < allClients.size(); i++) {	n
2. Client aux = allClients.get(i);	n-1
3. while i<=A.length and not found	(n*(n-1))/2
4. int j = i - 1;	((n*(n-1))/2)-1
5. allClients.set(j + 1, allClients.get(j));	((n*(n-1))/2)-1
6. j = j - 1;	n
7. allClients.set(j + 1, aux);	n-1
Total	n^2

```
private void sort() {
	for (int i = n / 2 - 1; i >= 0; i--) {
		heapify(theArray, n, i);
	}
	for (int i = n - 1; i > 0; i--) {
		V temp = theArray.get(0);
		theArray.set(0, theArray.get(i));
		theArray.set(i, temp);
		heapify(theArray, i, 0);
	}
}
```

INSTRUCCIÓN	VECES QUE SE REPITE (Big O)
8. for (int i = n / 2 - 1; i >= 0; i--) {	n/2
9. heapify(theArray, n, i);	(n(logn))/2
10. for (int i = n - 1; i > 0; i--)	n
11. V temp = theArray.get(0);	n
12. theArray.set(0, theArray.get(i));	n
13. theArray.set(i, temp);	n
14. heapify(theArray, i, 0);	n(logn)
Total	O(nlogn)

Análisis de complejidad temporal del Direct Hash Sorting.

Ya que la agregación a esta estructura posee un ordenamiento directo aplicado al problema específico se concluye que la complejidad temporal es O(1).

Análisis de complejidad temporal del Natural BTS Sorting.

La inserción de elementos a un árbol binario de búsqueda posee un ordenamiento natural, por lo cual la complejidad del ordenamiento es igual a la de la inserción, la cual es O(log n). Entonces, agregar n elementos a un árbol cualquiera da como resultado una complejidad O(n log n).