Marco Antonio perez - A00351452

Jose Manuel Ossorio - A00130524

Juan David Aguirre - A00352469

**Experimento algoritmos**

1. **Planeación y realización:**
2. **Diseño del experimento**

Nuestro experimento consiste en medir la eficiencia de dos algoritmos de ordenamiento diferentes. Ambos algoritmos pueden ser utilizados para ordenar cualquier tipo de objeto con base en algún criterio, pero en este experimento ordenaremos arreglos de números enteros. Para medir la eficiencia de los algoritmos utilizaremos como unidad experimental el tiempo que tardan en ordenar un arreglo de enteros. Para obtener una medida representativa, se realizarán 12 mediciones con cada tratamiento.

**1.1 Unidad experimental:** el tiempo promedio que tarda el algoritmo en ordenar un arreglo de enteros

**1.2 Variable de respuesta:** nuestra variable de respuesta será el tiempo promedio que tarda el algoritmo en ordenar el arreglo, en donde un menor tiempo significa un mejor desempeño y una mejor calidad de ejecución.

**1.3 Factores controlables:** entre los factores controlables para nuestro experimento se encuentran los siguientes:

* Algoritmo utilizado para ordenar el arreglo
* Lenguaje de programación
* Tamaño del arreglo
* Sistema operativo en donde se ejecuta el algoritmo
* Procesador usado
* Memoria RAM del equipo
* Número de aplicaciones abiertas en el momento de ejecución

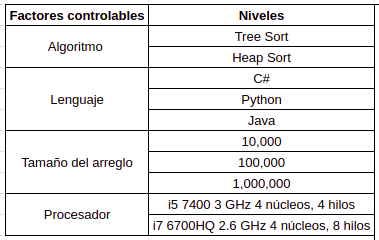
**1.4 Factores no controlables:** entre los factores no controlables para nuestro experimento se encuentran los siguientes:

* Grado de orden del arreglo a ordenar (no es controlable ya que el arreglo es generado aleatoriamente)

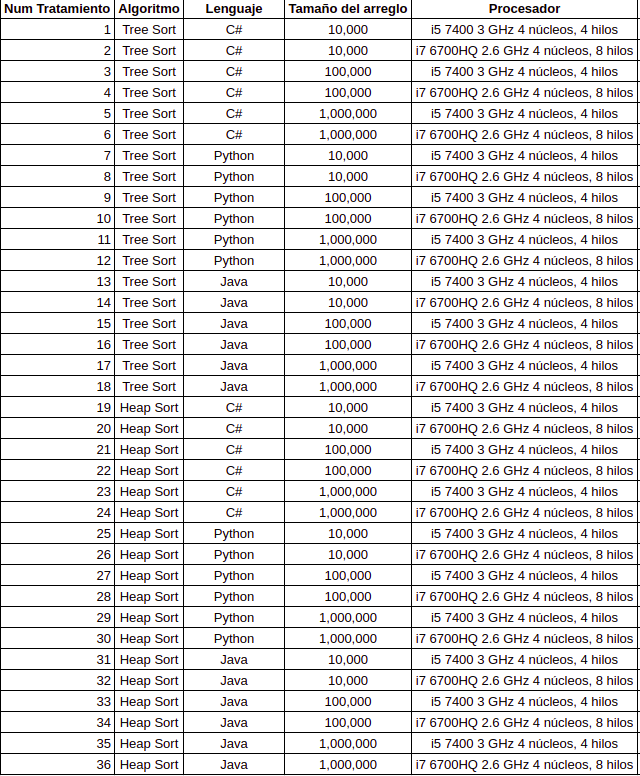
**1.5 Factores estudiados:** los factores estudiados para nuestro experimento serán:

* Algoritmo utilizado para ordenar el arreglo
* Lenguaje de programación
* Tamaño del arreglo
* Procesador usado

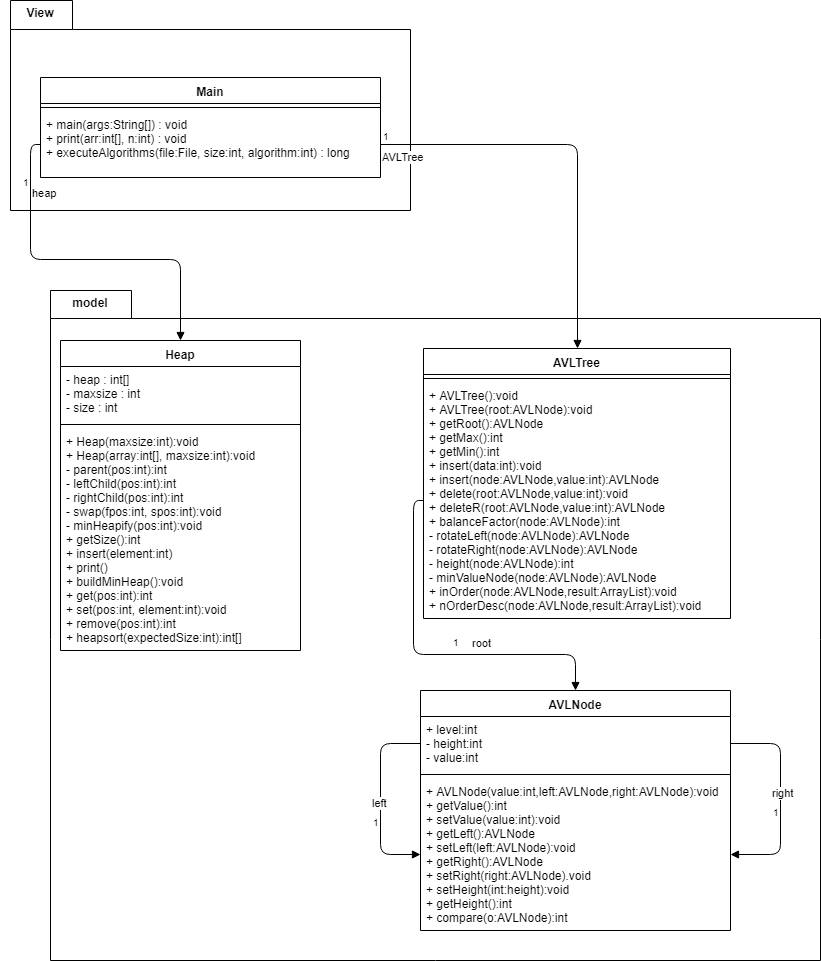
**1.6 Niveles:** a continuación se presenta una tabla con los niveles para cada factor controlable



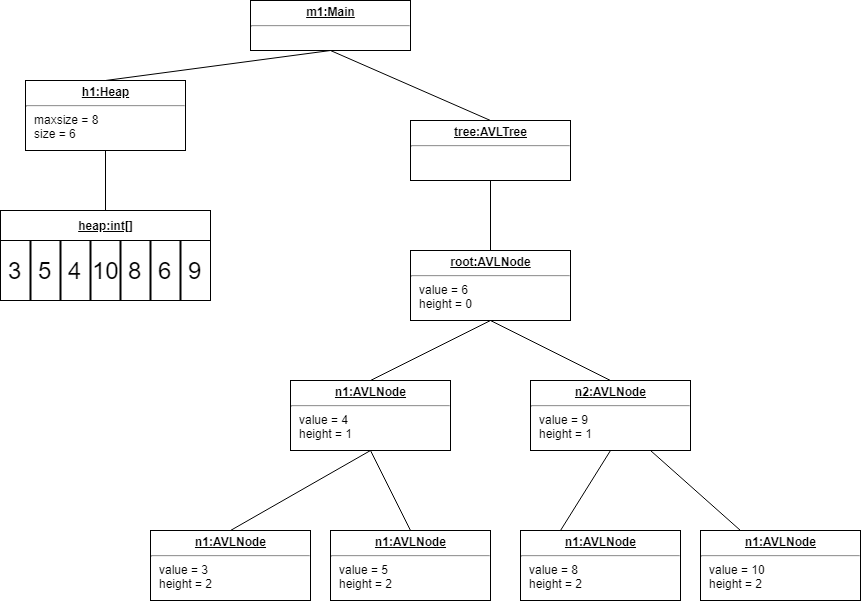
**1.7 Tratamientos:** los tratamientos resultantes de la combinación de los diferentes niveles de los factores controlables son los siguientes



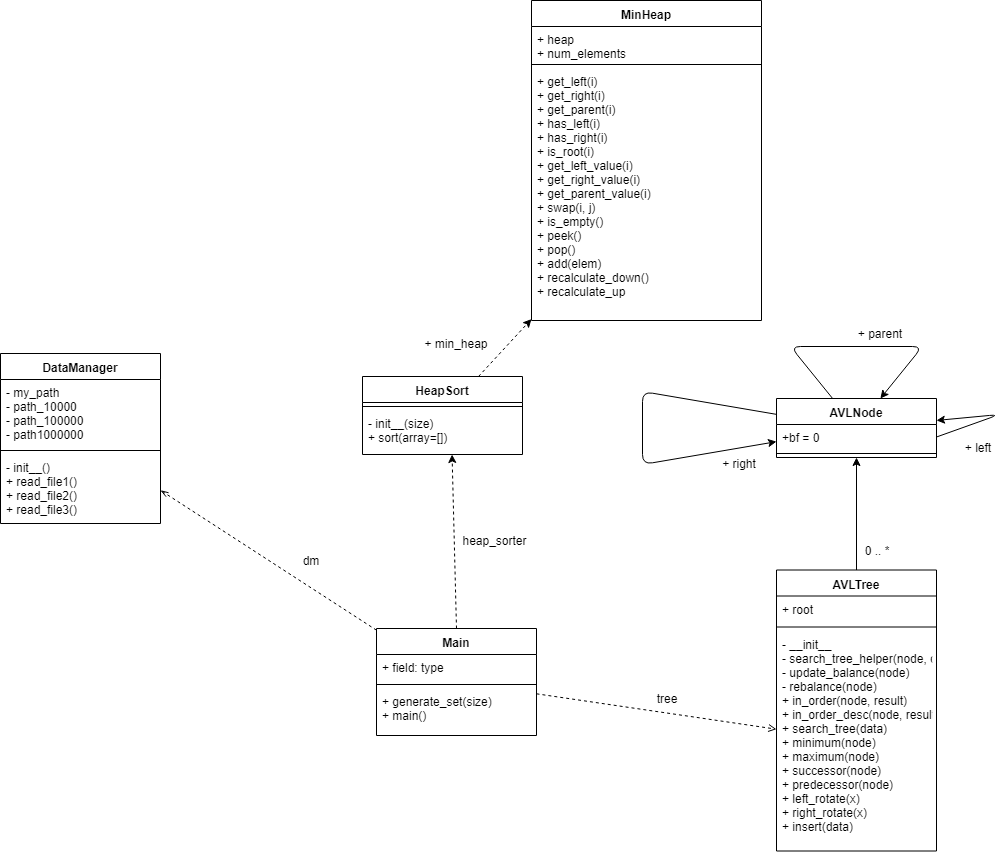
**2. Diseño del programa en Java, C#, Python**

**2.1 Diagrama de clases Java**

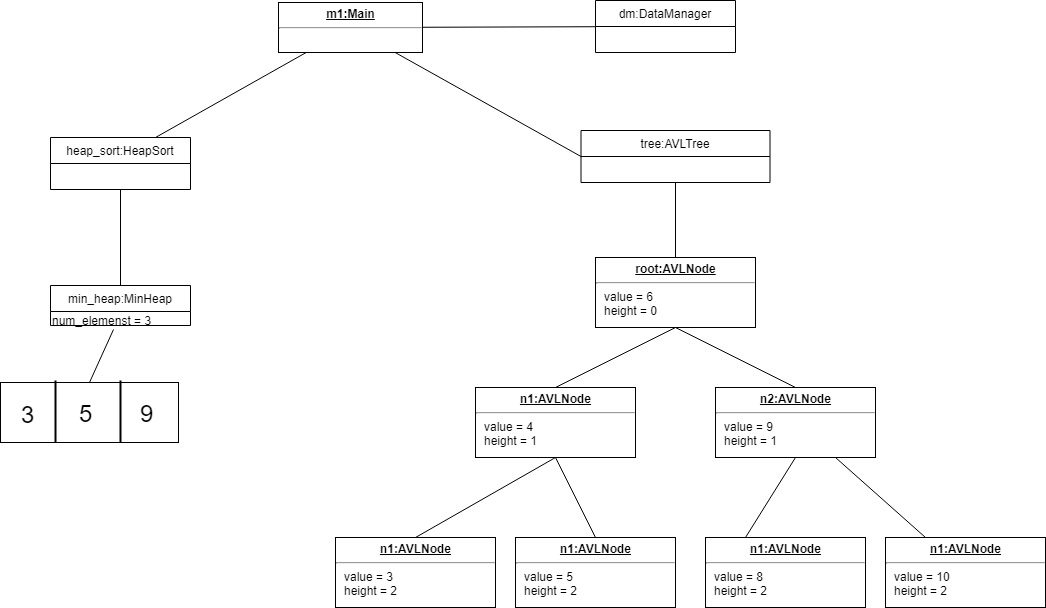
**2.2 Diagrama de objetos Java**

****

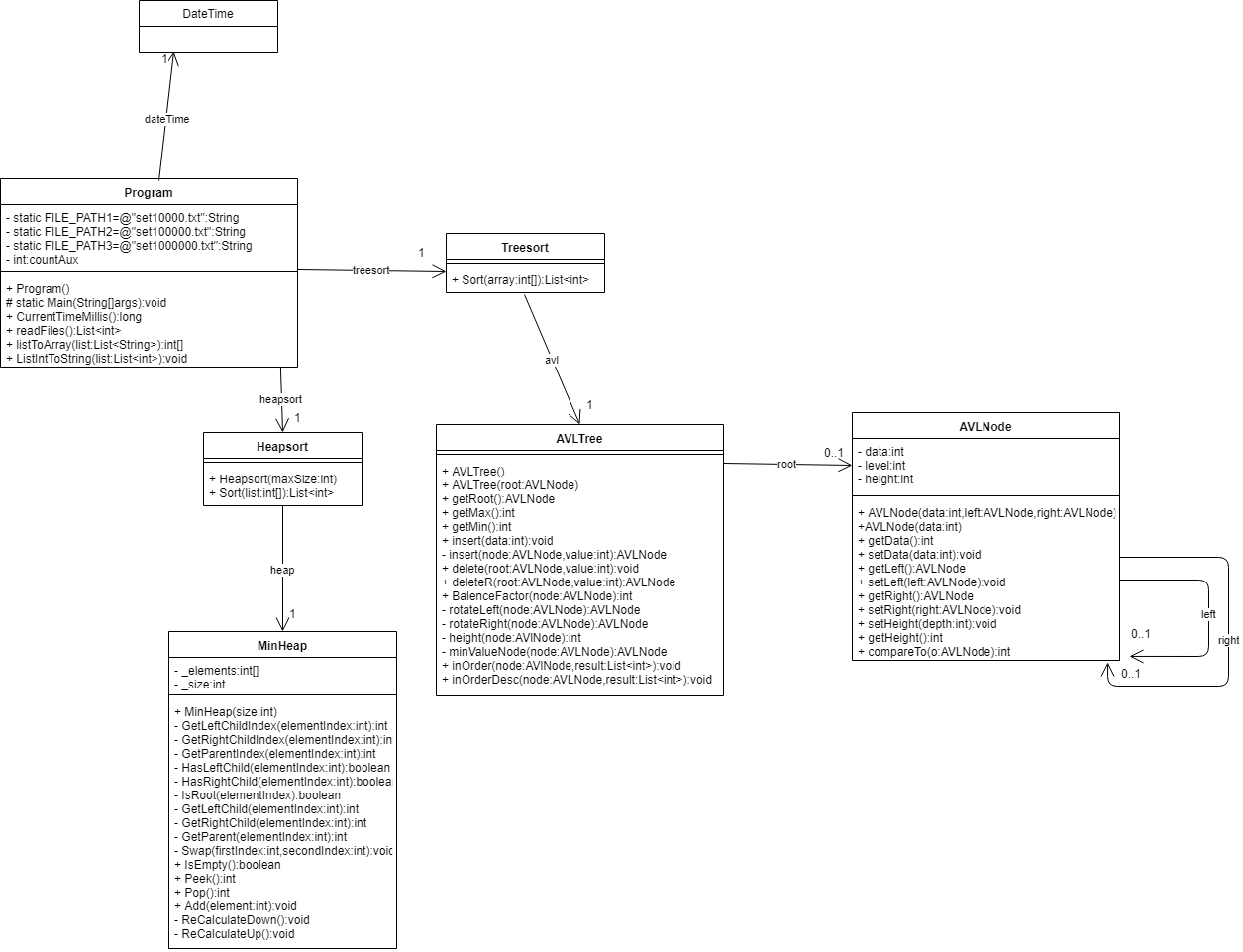
**2.3 Diagrama de clases Python:**



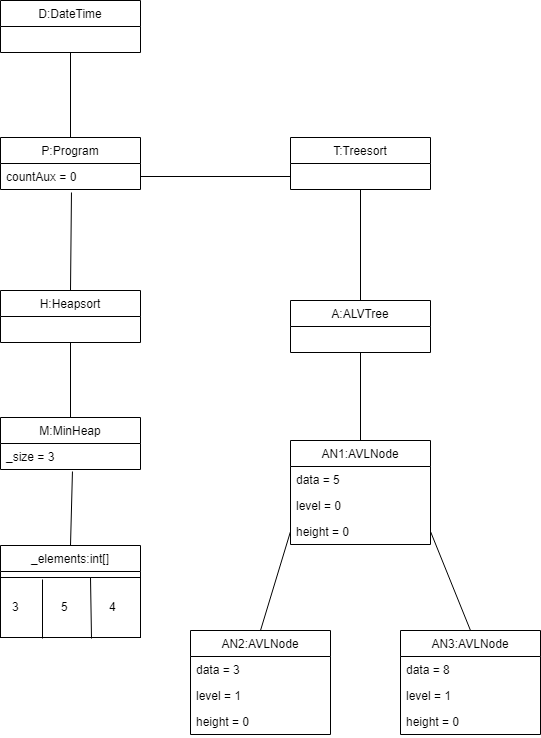
**2.4 Diagrama de objetos Python:**



**2.5 Diagrama de clases en C#:**

****

**2.5 Diagrama de objetos en C#:**

****

**3. Diseño de pruebas unitarias**

Para Heapsort:

|  |  |
| --- | --- |
| número de prueba | 1 |
| método | Sort(), de la clase Heapsort |
| Entrada | El arreglo de enteros el cual quiere se quiere ordenar |
| Salida | Un arreglo con los elementos del arreglo de entrada pero ordenado ascendentemente |

|  |  |
| --- | --- |
| Caso de prueba | 1.1 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Heapsort para un máximo de 10 elementos, donde la lista entrada será con cantidad de elementos par y esos elementos están ya ordenados |
| Entrada | new int([1][2][3][4][5][6][7][8][9][10]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.2 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Heapsort para un máximo de 10 elementos, donde la lista entrada será con cantidad de elementos par y esos elementos están ordenados inversamente |
| Entrada | new int([10][9][8][7][6][5][4][3][2][1]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.3 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Heapsort para un máximo de 10 elementos, donde la lista entrada será con cantidad de elementos par y esos elementos están desordenados |
| Entrada | new int([2][9][1][6][3][7][4][5][8][10]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.4 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Heapsort para un máximo de 9 elementos, donde la lista entrada será con cantidad de elementos impar y esos elementos están ordenados |
| Entrada | new int([1][2][3][4][5][6][7][8][9]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.5 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Heapsort para un máximo de 9 elementos, donde la lista entrada será con cantidad de elementos impar y esos elementos están ordenados inversamente |
| Entrada | new int([9][8][7][6][5][4][3][2][1]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.6 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Heapsort para un máximo de 9 elementos, donde la lista entrada será con cantidad de elementos impar y esos elementos están desordenados |
| Entrada | new int([2][9][1][6][3][7][4][5][8]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

Para Treesort:

|  |  |
| --- | --- |
| número de prueba | 1 |
| método | Sort(), de la clase Treesort |
| Entrada | El arreglo de enteros el cual quiere se quiere ordenar |
| Salida | Un arreglo con los elementos del arreglo de entrada pero ordenado ascendentemente |

|  |  |
| --- | --- |
| Caso de prueba | 1.1 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Treesort para un máximo de 10 elementos, donde la lista entrada será con cantidad de elementos par y esos elementos están ya ordenados |
| Entrada | new int([1][2][3][4][5][6][7][8][9][10]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.2 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Treesort para un máximo de 10 elementos, donde la lista entrada será con cantidad de elementos par y esos elementos están ordenados inversamente |
| Entrada | new int([10][9][8][7][6][5][4][3][2][1]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.3 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Treesort para un máximo de 10 elementos, donde la lista entrada será con cantidad de elementos par y esos elementos están desordenados |
| Entrada | new int([2][9][1][6][3][7][4][5][8][10]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.4 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Treesort para un máximo de 9 elementos, donde la lista entrada será con cantidad de elementos impar y esos elementos están ordenados |
| Entrada | new int([1][2][3][4][5][6][7][8][9]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.5 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Treesort para un máximo de 9 elementos, donde la lista entrada será con cantidad de elementos impar y esos elementos están ordenados inversamente |
| Entrada | new int([9][8][7][6][5][4][3][2][1]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

|  |  |
| --- | --- |
| Caso de prueba | 1.6 |
| Objetivo | Probar el buen funcionamiento del método y la correctitud de la salida |
| Escenario | Un Treesort para un máximo de 9 elementos, donde la lista entrada será con cantidad de elementos impar y esos elementos están desordenados |
| Entrada | new int([2][9][1][6][3][7][4][5][8]); |
| Salida | El Assert.isTrue después de revisar con una lista de resultado esperado retorna true |

**4. Análisis de complejidad de los algoritmos**

**4.1 Heapsort**

|  |  |
| --- | --- |
| **Línea de código** | **Costo** |
| Heapsort() |  |
| for i = heap.length hacia 2 | n-1 |
| cambiar heap[1] con heap[i] | 1 |
| heap.size = heap.size - 1 | 1 |
| MinHeapify(1) | logn |

MinHeapify es una operación cuya complejidad asintótica es de *log n* pues siempre recorrerá un porción del montículo cuya máxima longitud siempre será log n. El anterior pseudocódigo asume que el montículo ya ha sido construído. Por ende, tenemos como resultado:

En notación asintótica:

**4.2 Treesort**

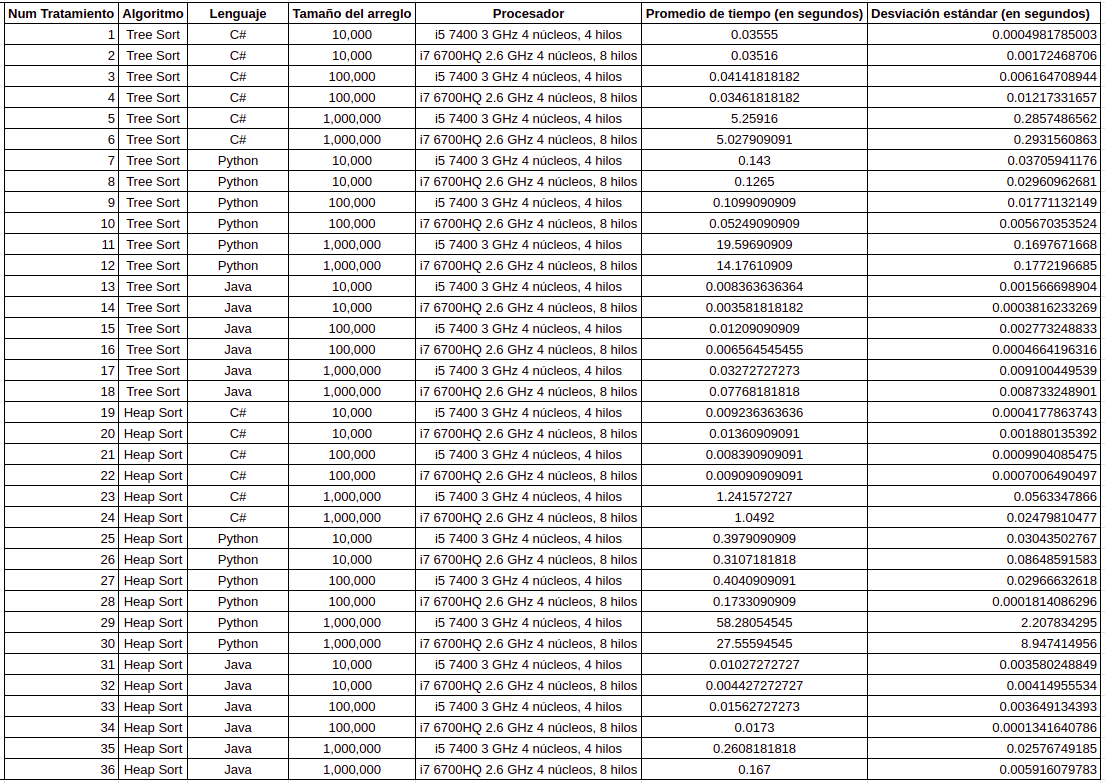
|  |  |
| --- | --- |
| **Línea de código** | **Costo** |
| TreeSort(A) |  |
| desde i = 0 hacia A.length - 1 | n |
| insertar(A[i]) | log n |

El anterior pseudocódigo usa la función insertar de los árboles AVL cuya complejidad asintótica es de O(*log n)* puesto que para insertar un elemento nuevo, debe recorrer un segmento del árbol. En este caso nunca será O(*n)* puesto que el árbol es autobalanceado. Por ende, tenemos:

El resultado del ordenamiento se puede obtener modificado levemente el recorrido *in-orden* lo que conlleva un costo temporal adicional pero no es tenido en cuenta pues no aporta al ordenamiento del arreglo de valores.

**II. Resultados y análisis:**

Luego de realizar nuestro experimento con los tratamientos mencionados, obtuvimos los siguientes resultados en tiempo de ejecución para cada tratamiento:



Los datos de cada ejecucion por cada tratamiento se encuentran en el documento tipo hoja de calculo anexado.

1. **Análisis estadístico**

Del experimento realizado con respecto a los algoritmos Treesort y Heapsort bajo estudio se puede observar que con respecto a la variable de interés, el tiempo, la mayoría tienen un comportamiento estable en las pruebas realizadas, dando como resultado que no haya datos atípicos en nuestra muestra. Además, la desviación estándar no genera intervalos en los que se contienen una gran mayoría de datos muy lejanos a los tiempos originales, esto puede llevar a considerar los algoritmos como temporalmente constantes.

Al comparar ambos algoritmos con la entrada grande (1.000.000 de enteros) en cada uno de los lenguajes de programación en estudio, se obtuvieron los siguientes resultados:

* Para C#:
  + Treesort:

|  |  |
| --- | --- |
| Procesador | Tiempo de ejecución en segundos |
| i5 7400 3 GHz 4 núcleos, 4 hilos | 5.25916 |
| i7 6700HQ 2.6 GHz 4 núcleos, 8 hilos | 5.027909091 |

* Heapsort:

|  |  |
| --- | --- |
| Procesador | Tiempo de ejecución en segundos |
| i5 7400 3 GHz 4 núcleos, 4 hilos | 1.241572727 |
| i7 6700HQ 2.6 GHz 4 núcleos, 8 hilos | 1.0492 |

* Para Java:
  + Treesort:

|  |  |
| --- | --- |
| Procesador | Tiempo de ejecución en segundos |
| i5 7400 3 GHz 4 núcleos, 4 hilos | 0.03272727273 |
| i7 6700HQ 2.6 GHz 4 núcleos, 8 hilos | 0.07768181818 |

* + Heapsort:

|  |  |
| --- | --- |
| Procesador | Tiempo de ejecución en segundos |
| i5 7400 3 GHz 4 núcleos, 4 hilos | 0.2608181818 |
| i7 6700HQ 2.6 GHz 4 núcleos, 8 hilos | 0.167 |

* Para Python:
  + Treesort:

|  |  |
| --- | --- |
| Procesador | Tiempo de ejecución en segundos |
| i5 7400 3 GHz 4 núcleos, 4 hilos | 19.59690909 |
| i7 6700HQ 2.6 GHz 4 núcleos, 8 hilos | 14.17610909 |

* + Heapsort:

|  |  |
| --- | --- |
| Procesador | Tiempo de ejecución en segundos |
| i5 7400 3 GHz 4 núcleos, 4 hilos | 58.28054545 |
| i7 6700HQ 2.6 GHz 4 núcleos, 8 hilos | 27.55594545 |

**III Interpretación**

De los resultados anteriores podemos comparar la rapidez de ambos algoritmos en cada lenguaje de programación para las entradas grandes y los dos procesadores diferentes. En el caso de C#, se observa que Heapsort es mucho más rápido en ambos procesadores. Sin embargo, para el caso de Java, Treesort presenta un menor tiempo promedio de ejecución que Heapsort en ambos procesadores. Por ultimo, vemos que en Python Treesort es tambien mas rapido que Heapsort con los dos procesadores.

De lo anterior podemos concluir que dependiendo del lenguaje de programación que se vaya a utilizar para el desarrollo del programa, puede ser más eficiente usar Heapsort, como en el caso de C#, o Treesort, como en los casos de Python y Java.

Además, cuando se están ordenando bajos volúmenes de datos, sin importar el lenguaje o procesador, ambos algoritmos tienden a ejecutarse en tiempos muy similares. Las diferencias no son muy notables para conjuntos de enteros del orden 10^1, 10^2 y 10^3.

Por otra parte, resulta evidente que las ejecuciones en Python suelen ser más lentas en comparación con los otros dos lenguajes. Esto lleva a considerar a C# o Java como mejores opciones, con mayor razón si se manejan grandes cantidades de datos.

Finalmente, es claro que no sólo los factores descritos afectan el proceso. Por ejemplo, el procesador i7 estaba corriendo en Ubuntu, cuyo consumo de memoria RAM es menor al de Windows (S.O. en el que corre el i5). Estos factores idealmente deberían de controlarse pero acarrean costos que por la naturaleza del estudio no pueden satisfacerse. Si, por ejemplo, se quisiera medir el efecto del tamaño de la memoria RAM, se debería considerar nuevas tarjetas madres y procesadores que aumentan los gastos.

**IV Conclusión**

Con el anterior estudio podemos concluir lo siguiente:

* El que dos algoritmos compartan la misma complejidad temporal asintótica no significa que sus tiempos de ejecución reales sean los mismos. Generalmente durante el cálculo de la complejidad temporal de los algoritmos se toman en consideración los términos mayores dejando a un lado los otros. Durante una ejecución real, los términos “extras” cobran importancia. El programador que desee implementar alguno de los dos algoritmos analizados deberá considerar cuál de los dos se acomoda más a sus necesidades.
* El programador debe seleccionar entre los lenguajes cuál es el más adecuado para implementar los algoritmos. Por ejemplo, si se desea realizar programas en un área en la que Python sea objetivamente más óptimo, puede compensar la posible lentitud. Si, en cambio, la velocidad es de mayor importancia, Java y C# serían mejores opciones.
* Si se tiene la posibilidad económica de tener en cuenta tarjetas madres, memoria RAM, etc. y sus permutaciones, los resultados serán considerablemente más representativos de una situación del mundo real.