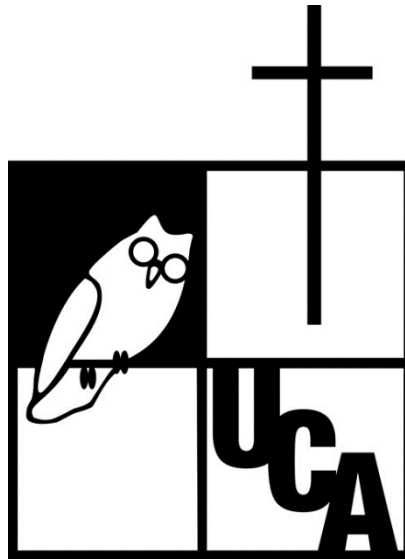


Universidad Centroamericana “José Simeón Cañas”



Análisis de Algoritmos

Taller 2

Grupo #34

Integrantes

ALEXANDRA BEATRIZ AGUILAR GARCIA	00164722	Sec01
DIEGO EDUARDO CASTRO QUINTANILLA	00117322	Sec02
RONALD FRANCISCO RIVAS DIAZ	00131321	Sec02

Fecha de entrega:

Sábado 12 de octubre de 2024.

Índice

Etapa II.....	3
Etapa III.....	3
Etapa IV.....	4
Ensayo.....	8

Etapa II

Es necesario realizar una investigación acerca de la estructura de datos llamada Montículo (Heap), enfocándose en sus propiedades, funcionamiento y las diferentes variantes que existen.

A continuación se presenta el link de la infografía: [Infografía T2 - 34](#)

Etapa III

En el presente informe se abordará la tarea encomendada por el departamento de Recursos Humanos del almacén Salem, que consiste en ordenar los salarios de sus empleados de manera eficiente para la distribución de los bonos de fin de año. Dado el crecimiento acelerado de la empresa y la expansión de su nómina, se requiere un método de ordenamiento rápido y de bajo consumo de memoria. A continuación, se explorarán las posibles soluciones para esta problemática, detallando el enfoque seleccionado y su implementación para garantizar un proceso ágil y confiable.


A continuación se presenta el link del repositorio: [AA022024_Taller2_Grupo#34](#)

Etapa IV

Para realizar un análisis formal, el código será dividido en secciones, lo que permitirá evaluar cada una de sus partes por separado. Posteriormente, se sumarán las magnitudes correspondientes de cada sección con el fin de obtener el valor total de manera precisa.

Análisis del código no recursivo

Incluir librerías y directiva de espacios



```
1  #include "../lista_salarios.h"      //C1 * 1
2  #include <iostream>                  //C2 * 1
3
4  using namespace std;                //C3 * 1
```

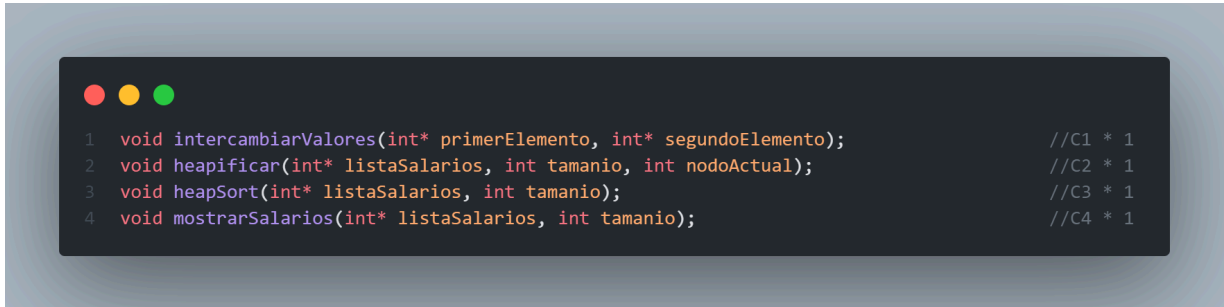
Cada inclusión y uso del namespace son operaciones constante por eso serán $\Theta(1)$ será el resultado final de esas líneas de código.

$$T(n) = C1*1 + C2*1 + C3*1$$

$$T(n) = \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$$

$$T(n) = \Theta(1)$$

Declaración de funciones



```
1  void intercambiarValores(int* primerElemento, int* segundoElemento);      //C1 * 1
2  void heapificar(int* listaSalarios, int tamano, int nodoActual);           //C2 * 1
3  void heapSort(int* listaSalarios, int tamano);                             //C3 * 1
4  void mostrarSalarios(int* listaSalarios, int tamano);                      //C4 * 1
```

A la hora de declarar las funciones, todas son linea de codigo constante asi que sera por defecto $\Theta(1)$ el tiempo total de las declaraciones.

$$T(n) = C1*1 + C2*1 + C3*1 + C4*1$$

$$T(n) = \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$$

$$T(n) = \Theta(1)$$

Archivo de cabecera lista_salarios.h

```
1  #ifndef LISTA_SALARIOS_H // C1 * 1
2  #define LISTA_SALARIOS_H // C2 * 1
3
4  const int numSalarios = 1000; // C3 * 1
5
6  int listaSalarios[numSalarios] = { // C4 * 1
7      3466, 1694, 1850, 2452, 2246, 2475, 862, 521, 568, 459, 1688, 1274, 1614,
8      1281, 3069, 1446, 1749, 3574, 1093, 3187, 1244, 2307, 1401, 571, 815, 1295,
9      1257, 3739, 2652, 2973, 2883, 2444, 1671, 1897, 3674, 2179, 715, 3906, 559,
10     898, 2745, 2318, 3820, 830, 3094, 3285, 1852, 1376, 1216, 3901, 1494, 2731,
11     1692, 1430, 2936, 3325, 1986, 2711, 3977, 3654, 1247, 1201, 1730, 2969, 3722,
12     1785, 2869, 3831, 1019, 489, 3840, 2051, 2099, 3229, 2945, 1247, 1227, 2188,
13     .....
14 };
15
16 #endif // C4 * 5
```

Como banco de datos utilizamos un archivo de cabecera con extensión .h que contiene un arreglo con 1000 salarios y al únicamente declararlo el tiempo de ejecución para la creación e instanciación es constante

$$T(n) = C1*1 + C2*1 + C3*1 + C4*1$$

$$T(n) = \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$$

$$T(n) = \Theta(1)$$

Función intercambio de valores

```
1 void intercambiarValores(int* primerElemento, int* segundoElemento) { //C1 * 1
2     int temp = *primerElemento; //C2 * 1
3     *primerElemento = *segundoElemento; //C3 * 1
4     *segundoElemento = temp; //C4 * 1
5 }
```

En la función intercambiarValores únicamente se encarga de intercambiar dos valores por eso todas las líneas de código son constante, significa que al final la suba de los tiempo debería dar $\Theta(1)$.

$$T(n) = C1*1 + C2*1 + C3*1 + C4*1$$

$$T(n) = \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$$

$$T(n) = \Theta(1)$$

Función de mostrar salarios

```
1 void mostrarSalarios(int* listaSalarios, int tamano) { //C1 * 1
2     for (int i = tamano - 1; i >= 0; i--) { //C2 * n
3         cout << (tamano - i) << " - " << *(listaSalarios + i) << endl; //C3 * 1
4     }
5 }
```

La función muestra los salarios en la base de datos y adentro contiene un for, esa función recorre toda la lista completa en un bucle y por ese detalle la complejidad de la función es lineal $\Theta(n)$.

$$T(n) = C1*1 + C2*n + C3*1$$

$$T(n) = \Theta(1) + \Theta(n) = \Theta(n)$$

$$T(n) = \Theta(n)$$

Luego, en base a la parte del código que sea recursiva, se deberá definir la Recurrencia que representa el código completo. En esta parte, se deberá explicar a detalle el razonamiento empleado para definir cada pieza de la Recurrencia definida.

Análisis del código recursivo

```
1 // Aseguramos que el subarbol con la raíz en el nodoActual cumpla Max-Heap.
2 void heapificar(int* listaSalarios, int tamaño, int nodoActual) {
3     int* nodoMayor = listaSalarios + nodoActual;
4     int* hijoIzquierdo = listaSalarios + 2 * nodoActual + 1;
5     int* hijoDerecho = listaSalarios + 2 * nodoActual + 2;
6
7     if (2 * nodoActual + 1 < tamaño) {
8         if (*hijoIzquierdo > *nodoMayor) {
9             nodoMayor = hijoIzquierdo;
10        }
11    }
12
13    if (2 * nodoActual + 2 < tamaño) {
14        if (*hijoDerecho > *nodoMayor) {
15            nodoMayor = hijoDerecho;
16        }
17    }
18
19    if (nodoMayor != listaSalarios + nodoActual) {
20        intercambiarValores(listaSalarios + nodoActual, nodoMayor);
21        heapificar(listaSalarios, tamaño, nodoMayor - listaSalarios);
22    }
23 }
```

La función donde se realiza la recursión es heapificar ya que toma un nodo en un árbol binario en este caso representado por un arreglo y asegura que el subárbol con la raíz en ese nodo específico cumpla con la propiedad del Max-Heap por lo que se asegura que el nodo actual (nodoActual) sea mayor que sus nodos hijos pero en caso que alguno de sus hijos tenga un valor mayor y por lo tanto no cumpla la propiedad la función realizará una llamada recursiva a sí misma en el nodo hijo afectado ya que es necesario mover un nodo hacia abajo dentro del árbol para restablecer la propiedad Max-Heap.

Cada vez que realizamos la llamada recursiva el tamaño del árbol se reduce a la mitad debido a que en cada nivel de la recursión nos movemos de la raíz hacia una de las hojas del árbol por lo que podemos representar el tamaño del subproblema como $T(\frac{n}{2})$ esto ya que descendemos un nivel en el árbol, el resto de la función se compone de operaciones con valor de ejecución constante $\Theta(1)$ ya que no dependen del tamaño del subárbol funciones como:

- Comparar el valor del nodo raíz con sus hijo izquierdo y derecho
- Comparar para saber cual es mayor
- Realizar un intercambio de valores en caso de ser necesario

Por lo tanto la recurrencia resultante de la función heapificar es

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

En un árbol binario la altura es $\log(n)$ ya que en un árbol que se encuentra balanceado el número de niveles es logarítmico por lo que el peor caso sería $O(\log n)$ y n representa el número de nodos que tiene el árbol.

Para poder aplicar el teorema maestro nuestra recurrencia debe poseer la siguiente estructura:

$$T(n) = a * T\left(\frac{n}{b}\right) + O(n^d)$$

donde:

$$a > 0 \text{ y } b > 1 \text{ y } d \geq 0$$

Aplicando el teorema maestro en la recurrencia de la función recursiva:

$$T(n) = 1 * T\left(\frac{n}{2}\right) + \Theta(1)$$

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

$$a = 1, b = 2, d = 0$$

$$n^{\log b * a} = n^{\log 2 * 1} = n^0 = 1$$

$$f(n) = \Theta(1)$$

Los valores obtenidos corresponden al caso 2 del teorema maestro que establece lo siguiente:

Si $d = \log b * a$ entonces $T(n) = \Theta(n^d \log n)$

Por lo tanto podemos afirmar que la solución de la recurrencia es:

$$T(n) = \Theta(n^0 \log n).$$

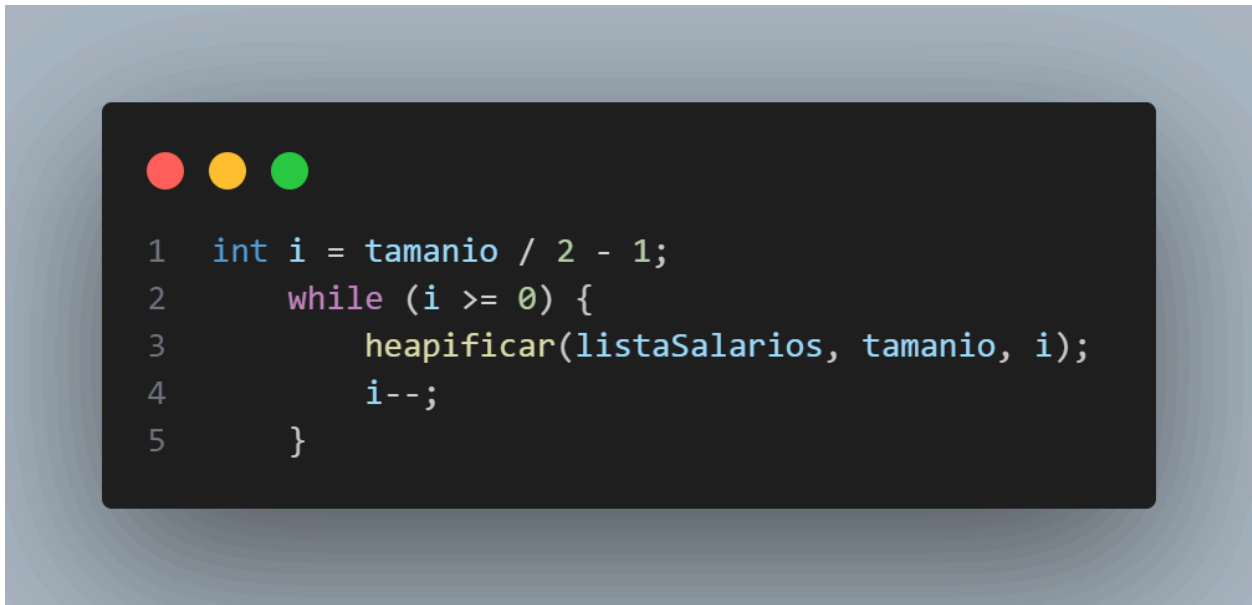
Simplificando:

$$T(n) = \Theta(\log n).$$

La función recursiva heapificar se invoca dentro de la función heapSort por lo que para obtener el tiempo de ejecución del código completo es necesario realizar su respectivo análisis:

La función cuenta con dos secciones principales:

- La conversión del arreglo original de la cabecera .h en un max heap.
- El ordenamiento del heap donde se intercambian valores mediante la función heapificar hasta que el heap tenga un tamaño de 1.



```
1  int i = tamaño / 2 - 1;
2      while (i >= 0) {
3          heapificar(listaSalarios, tamaño, i);
4          i--;
5      }
```

La primera sección de la función es la construcción del max-heap donde el ciclo while recorre hasta el último nodo que no es hoja realizando aproximadamente $\frac{n}{2}$ llamadas a heapificar donde “n” es el tamaño del array.

La suma del tiempo de todas las llamadas que se realizan a heapificar durante la construcción del max heap converge a $\Theta(n)$ ya que la mayoría de las llamadas se realizan en nodos cercanos a las hojas donde el tiempo de ejecución es mucho menor pero contando el peor caso de $\Theta(\log n)$.

```
1  int j = tamaño - 1;
2      while (j > 0) {
3          intercambiarValores(listaSalarios, listaSalarios + j);
4          heapificar(listaSalarios, j, 0);
5          j--;
6      }
```

La segunda sección de la función consiste en el ordenamiento donde en cada iteración el tamaño del heap se va reduciendo en uno mientras se intercambia el elemento máximo con el último elemento del heap actual y posteriormente se aplica heapificar al nodo raíz, el intercambio de valores se realiza mediante la función auxiliar intercambiarValores que posee un tiempo de ejecución constante $\Theta(1)$ y la llamada a heapificar como obtuvimos anteriormente su valor podemos decir que es $\Theta(\log n)$ donde n es el tamaño actual del heap.

Para obtener el tiempo total de ejecución utilizando el teorema maestro lo vamos a contextualizar como un algoritmo de divide y vencerás ya que para poder aplicarlo necesitamos una recurrencia de la forma

$$T(n) = a * T\left(\frac{n}{b}\right) + f(n)$$

Para plantear la recurrencia que represente nuestro algoritmo vamos a analizar su funcionamiento: dentro de la función heapSort dividimos el problema en dos subproblemas, llamamos una función con recursividad y realizamos un proceso de combinación nuestra recurrencia tendría la forma:

$$T(n) = 2 * T\left(\frac{n}{2}\right) + O(n)$$

Donde:

$a = 2$: el problema principal se divide en dos subproblemas

$b = 2$: el tamaño del subproblema va dividiéndose a la mitad

$d = 1$ d es el exponente de $\Theta(n^d) = \Theta(n)$

Identificando caso del teorema maestro

Caso 1: Si $d > \log b * a$ entonces $T(n) = \Theta(n^d)$

Caso 2: Si $d = \log b * a$ entonces $T(n) = \Theta(n^d \log n)$

Caso 3: Si $d < \log b * a$ entonces $T(n) = \Theta(n^{\log b * a})$

1- Calculando $n^{\log b * a}$

$$\log 2 * 2 = 1$$

2- Comparando d con $\log 2 * 2$

$$\log 2 * 2 = d$$

$$1 = 1$$

Se cumple el caso 2 del teorema maestro

3- Aplicando el teorema maestro la solución de nuestra recurrencia es

$$T(n) = \Theta(n^d \log n)$$

$$T(n) = \Theta(n^1 \log n)$$

$$\mathbf{T(n) = \Theta(n \log n)}$$

Calculando el tiempo de ejecución del código completo

Para calcular el tiempo total de ejecución de todo el programa analizamos la función main y utilizamos todos los tiempos de ejecución previamente calculados

```
1  int main() //C1 * 1
2  {
3      try { //C2 * 1
4          system("cls"); //C3 * 1
5          heapSort(listaSalarios, numSalarios); //C4 * nlog n
6          cout << "<-- Listado de salarios ordenados de forma descendente -->\n" << endl; //C5 * 1
7          mostrarSalarios(listaSalarios, numSalarios); //C6 * n
8          cout << "\n<-- Lista de salarios ha sido mostrada por completo -->\n" << endl; //C7 * 1
9      } catch (const exception& e) { //C8 * 1
10         cout << "Ha ocurrido un error: " << e.what() << endl; //C9 * 1
11     }
12     return 0; //C10 * 1
13 }
```

La operación en Main es donde se comienza a complicar más las cosas, la función adentro tiene un try catch que son constante pero tras haber realizado el análisis completo de la función recursiva en la siguiente sección del reporte sabemos que **heapSort** tiene una complejidad de $\Theta(n \log(n))$ bastante eficiente para ordenar grandes cantidades de datos y también encontramos una función lineal $\Theta(N)$ en mostrarSalarios porque ese arreglo recorre la lista N veces, las demás líneas de código dentro de Main son Constantes.

$$T(n) = C1*1 + C2*1 + C3*1 + C4*n\log(n) + C5*1 + C6*n + C7*1 + C8*1 + C9*1 + C10*1$$

$$T(n) = C1*1 + C2*1 + C3*1 + C5*1 + C7*1 + C8*1 + C9*1 + C10*1 + C6*n + C4*(n \log(n))$$

$$T(n) = \Theta(1) + \Theta(n) + \Theta(n \log(n)) = \Theta(n) + \Theta(n \log(n))$$

total:

$$TT(n) = \Theta(1) + \Theta(n \log(n)) + \Theta(n \log(n)) + \Theta(1) + \Theta(\log(n)) + \Theta(n)$$

$$TT(n) = \Theta(1) + \Theta(1) + \Theta(\log(n)) + \Theta(n \log(n)) + \Theta(n \log(n)) + \Theta(n)$$

Por lo tanto el tiempo de ejecución del algoritmo es:

$$TT(n) = \Theta(n \log(n))$$

Ensayo

La implementación de la función de heapificar junto con su análisis, ha proporcionado una comprensión más profunda de la eficiencia del algoritmo Heapsort. Las llamadas recursivas permitieron una gestión eficiente de grandes volúmenes de datos. Esto ha facilitado el análisis y desarrollo de problemas complejos al dividirlos en subproblemas más manejables, reafirmando la importancia de comprender las bases teóricas de los algoritmos y su impacto en el rendimiento.

El teorema maestro confirmó que la función de heapificar y las demás funciones que ayudan a complementar esta misma, que el tiempo de ejecución final sería de $\Theta(\log n)$, esto hace que todo el programa pueda ordenar 1,000 datos en menos de un segundo, esto es posible gracias al proceso de ordenamiento de intercambio de valores que permite que el tiempo total de ejecución sea de $\Theta(n \log n)$.

Gracias a las características destacadas, Almacenes Salem se beneficia enormemente de nuestro programa que emplea un método de ordenamiento rápido, eficiente y óptimo. Este método no solo satisface el requisito de bajo consumo de memoria, sino que, aunque la prueba se realizó con una base de datos de 1,000 registros, el programa puede gestionar volúmenes de datos mucho mayores sin afectar su rendimiento.