Universidad Centroamericana José Simeón
Cañas
Facultad de Ingeniería y Arquitectura

Departamento de Electrónica e Informática

Taller 2 Análisis y Algoritmos

Estudiantes:

Alejandro José Orellana Reyes 00125720

Allen Enrique Perdomo Cardona 00189021

Raúl Napoleón Caprile Ramos 00006422

HeapSort

Es un algoritmo de árbol binario donde se organizan los datos de modo que se asigne un padre y dos hijos, en el cual se intercambian datos con los padres de modo que cada hijo posea un dato menor al de su padre, hasta formar un “montículo” o heap donde el dato mayor es el padre de todos. Este dato se intercambia con el último del arreglo (hoja) y ya no se toma en cuenta para el árbol ya que este valor se considera ordenado. Quedando un nuevo árbol más pequeño por ordenar y se repite el proceso una y otra vez, hasta no quedar ningún dato en el árbol.

Función reajustarMonticulo (recursiva)

```
13  void reajustarMonticulo(Empleado* empleados, int n, int i) {
14      int mayor = i;
15      int izquierda = 2 * i + 1;
16      int derecha = 2 * i + 2;
17
18      if (izquierda < n && empleados[izquierda].salario > empleados[mayor].salario)
19          mayor = izquierda;
20
21      if (derecha < n && empleados[derecha].salario > empleados[mayor].salario)
22          mayor = derecha;
23
24      if (mayor != i) {
25          swap(empleados[i], empleados[mayor]);
26          reajustarMonticulo(empleados, n, mayor);
27      }
28  }
```

Línea	Orden de magnitud
14	O(1)
15	O(1)
16	O(1)
18	O(1)
19	O(1)
21	O(1)
22	O(1)
24	O(1)
25	O(1)
26	O(log n)

Esta función recursiva establece el dato actual como el mayor y desde el padre compara si el hijo izquierdo o derecho son mayores, al tratarse de una mera comparación la complejidad de este apartado es O(1), de ser mayor, establece a este hijo como el mayor e intercambia de posición con él y vuelve a llamarse a sí misma **(Recursividad)** para reajustar el siguiente nivel hasta llegar a la raíz, entrando aquí la parte de complejidad O(log n) ya que en el peor de los casos el nodo actual desciende hasta la hoja del subárbol más profundo.

Complejidad: O(log n)

Función ConstruirMonticulo

```
30  // Función para construir el montículo a partir de un arreglo de empleados
31  void construirMonticulo(Empleado* empleados, int n) {
32      for (int i = n / 2 - 1; i >= 0; i--) {
33          reajustarMonticulo(empleados, n, i);
34      }
35  }
```

Línea	Orden de magnitud
32	O(n)
33	O(log n)

Esta función construye el árbol con el valor máximo en la raíz partiendo del arreglo de los empleados a este montículo se le conoce como Max Heap, el árbol se genera ejecutando n/2-1 la función **reajustarMonticulo** que ya se estableció que su magnitud es O(log n) simplificando queda que el orden de magnitud de esta función es de O(n).

Complejidad: O(n)

Función ordenarPorMonticulo

```
38  void ordenarPorMonticulo(Empleado* empleados, int n) {
39      construirMonticulo(empleados, n);
40      for (int i = n - 1; i > 0; i--) {
41          swap(empleados[0], empleados[i]);
42          reajustarMonticulo(empleados, i, 0);
43      }
44  }
```

Línea	Orden de magnitud
39	O(n)
40	O(n)
41	O(n)*O(1)
42	O(n)*O(log n)

Con esta función se ordena todo el arreglo partiendo Max heap dado por **ConstruirMonticulo**, para después entrar en un bucle que se ejecuta n-1 veces donde el dato de la raíz (valor máximo) se intercambia con el último dato del arreglo (hoja) y se reajusta el montículo con la función **reajustarMonticulo** para volver a ordenar el árbol y sacar otro dato mayor en la raíz. Ya que la función **reajustarMonticulo** simplificando podríamos decir que se ejecuta O(n) veces y esta tiene una magnitud O(log n) la magnitud de esta función es O(n log n).

Complejidad: O(n log n)

Función imprimirResultados

```
47  void imprimirResultados(Empleado* empleados, int n) {
48
49      int anchoNombre = 20;
50      int anchoSalario = 10;
51
52      cout << "+--------------------+------------+" << endl;
53
54
55      cout << "| " << "Nombre" << string(anchoNombre - 6, ' ')
56           << " | " << "Salario" << string(anchoSalario - 7, ' ') << " |" << endl;
57
58      cout << "+--------------------+------------+" << endl;
59
60      for (int i = n - 1; i >= 0; i--) {
61          cout << "| " << empleados[i].nombre
62               << string(anchoNombre - empleados[i].nombre.length(), ' ') << " | ";
63
64          cout << empleados[i].salario
65               << string(anchoSalario - to_string(empleados[i].salario).length(), ' ')
66               << " |" << endl;
67      }
68      cout << "+--------------------+------------+" << endl;
69  }
```

Línea	Orden de magnitud
49	O(1)

50	O(1)
52	O(1)
55	O(1)
56	O(1)
58	O(1)
60	O(n)
61	O(n)*O(1)
62	O(n)*O(1)
64	O(n)*O(1)
65	O(n)*O(1)
66	O(n)*O(1)
68	O(1)

Función que se encarga de imprimir cada dato del árbol con un formato que lo hace más legible, recorre todo el árbol con un for y muestra el dato con un y el nombre del empleado con cout, este bucle tiene un peso de O(n) ya que recorre todos los datos.

Complejidad: O(n)

Función Main

```
71  int main() {
72      int capacidad = 100;
73      int n = 0;
74      Empleado* empleados = new Empleado[capacidad];
75
76      ifstream archivo("salarios.txt"); // Archivo de entrada
77
78      if (!archivo) {
79          cerr << "No se pudo abrir el archivo de salarios." << endl;
80          return 1;
81      }
82
83      string nombre;
84      int salario;
85
86      while (archivo >> nombre >> salario) {
87          if (n == capacidad) {
88              capacidad *= 2;
89              Empleado* nuevoArray = new Empleado[capacidad];
90              for (int i = 0; i < n; i++) {
91                  nuevoArray[i] = empleados[i];
92              }
93              delete[] empleados;
94              empleados = nuevoArray;
95          }
96
97          empleados[n].nombre = nombre;
98          empleados[n].salario = salario;
99          n++;
100     }
101
102     archivo.close();
103
104     // Ordenar los empleados por sus salarios
105     ordenarPorMonticulo(empleados, n);
106
107
108     imprimirResultados(empleados, n);
109
110     delete[] empleados;
111
112     return 0;
113 }
```

Línea	Orden de magnitud
72	O(1)
73	O(1)
74	O(1)
76	O(1)

78	O(1)
79	O(1)
80	O(1)
83	O(1)
84	O(1)
86	O(n)
87	O(n)
88	O(n)
89	O(n)
90	O(n)
91	O(n)
93	O(n)
94	O(n)
97	O(1)
98	O(1)
99	O(1)
102	O(1)
105	O(n log n)
108	O(n)
110	O(1)

En el main se insertan los datos desde un archivo .txt previamente consultado en el que están todos los empleados y sus respectivos salarios en desorden. Para después ser almacenados en un arreglo de la estructura “Empleado” donde haciendo uso de la función **ordenarPorMonticulo** serán ordenados y después visualizados a nivel de consola usando la función **imprimirResultados.** A sabiendas que se tienen que consultar y agregar todos los datos y hacer uso de las funciones previamente mencionadas, esta función posee una complejidad **O(n log n).**

Complejidad: O(n log n)

Recursividad

La función recursiva se llama en la línea 26 **reajustarMonticulo**, utiliza recursión descendente, donde la llamada recursiva se realiza primero para resolver subproblemas más pequeños. La recurrencia ocurre en la mitad de los hijos por lo tanto la ecuación quedaría:

$T(n) = T(\frac{n}{2}) + O(1)$

Aplicando el teorema maestro

$T(n) = aT(\frac{n}{b}) + O(n^d)$

a=1 (es una llamada recursiva)

b=2 (se divide el problema a la mitad)

d=0 (fuera de recursión el trabajo es > O(1), una constante)

Comparando:

Si $a < b^d$, la complejidad es $O(n^d)$

Si $a = b^d$, la complejidad es $O(n^d log\ n)$

Si $a > b^d$, la complejidad es $O(n^{log_b a})$

Tenemos:

a = 1, b = 2, d = 0

Por lo tanto $a = b^d$, dando como resultado una complejidad de $O(n^d log\ n)$, resultando en

T(n) = $O(log\ n)$

Reflexión

Como equipo observamos que este tipo de algoritmo es muy versátil, entendible y tiene la ventaja que todo el ordenamiento lo hace desde el propio arreglo sin usar espacio extra, aunque no es el mejor tipo de árbol binario, su facilidad de implementación lo hace una buena opción para el ordenamiento de muchos datos, ordenando los 1000 datos de la muestra en apenas 756 ms (el tiempo es un promedio y puede variar drásticamente según el equipo) esto debido a su orden de magnitud es de **O(n log n).**