Universidad Centroamericana “José Simeón Cañas”
Facultad de Ingeniería y Arquitectura
Análisis de Algoritmos
ciclo 02/2024

Actividad:
Taller 2

Integrantes:

Calvo Solis, Diego	00103322
William Josue Pineda Martinez	00225919
Luis Andrée Hernández Maltez	00012321

Cabecera y declaraciones globales

```
#include <iostream>                                   C1
#include <fstream>                                    C2
using namespace std;                                  C3


const int NumEmployees = 1000;                        C4


struct Employee
{
   int employeeCode;                                  C5
   char firstName[50];                                C6
   char lastName[50];                                 C7
   double salary;                                     C8
};
```

$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8$

$\therefore T(n) = O(1)$

Función Swap

```
void swap(Employee *a, Employee *b)
{
   Employee temp = *a;                                C1
   *a = *b;                                           C2
   *b = temp;                                         C3
}
```

$T(n) = C_1 + C_2 + C_3$

$\therefore T(n) = O(1)$

Función min_heap

```
void min_heap(Employee *workers, int size, int largestnode)
{
   int index = largestnode;                                                    C1
   int leftSon = 2 * largestnode + 1;                                          C2
   int rightSon = 2 * largestnode + 2;                                         C3

   if (leftSon < size && workers[leftSon].salary < workers[index].salary)      C4
   {
      index = leftSon;                                                         C5 * max(1, 0)
   }

   if (rightSon < size && workers[rightSon].salary < workers[index].salary)    C6
   {
      index = rightSon;                                                        C7 * max(1, 0)
   }

   if (index != largestnode)                                                   C8
   {
      swap(&workers[largestnode], &workers[index]);                            O(1) * max(1, 0)
      min_heap(workers, size, index);                                          T(n/2)
   }
}
```

$$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8 + O(1) + T(n/2)$$
$$T(n) = T(n/2) + O(1)$$

por teorema maestro:
$a = 1,\ b = 2,\ d = 0$

$log_b a = log_2 1 = 0$
$\therefore d = log_b a$

por lo que concluimos:
$T(n) = n^d log_2 n$
$\therefore T(n) = log_2 n$

Función insert_heap

```
void insert_heap(Employee *&workers, int &size, Employee newEmployee)
{
   size++;                                                          C1
   workers[size - 1] = newEmployee;                                 C2
   int i = size - 1;                                                C3
   while (i != 0)                                                   C4 * log2n
   {
      int parent = (i - 1) / 2;                                     C5 * log2n
      if (workers[parent].salary > workers[i].salary)               C6 * log2n
      {
         swap(&workers[parent], &workers[i]);                       C7 * log2n * max(0, 1)
         i = parent;                                                C8 * log2n * max(0, 1)
      }
      else
      {
         break;                                                     C9 * log2n * max(0, 1)
      }
   }
}
```

$$T(n) = C_1 + C_2 + C_3 + C_4 log_2 n + C_5 log_2 n + C_6 log_2 n + C_7 log_2 n + C_8 log_2 n + C_9 log_2 n$$

$$T(n) = C_1 + C_2 + C_3 + (C_4 + C_5 + C_6 + C_7 + C_8 + C_9) log_2 n$$

$$T(n) = A + B log_2 n, \ donde\ A\ y\ B\ son\ constantes$$

$$\therefore T(n) = O(log_2 n)$$

Función search_in_heap

```
int search_in_heap(Employee *workers, int size, int employeeCode)
{
   for (int i = 0; i < size; i++)                                   C1 * n
   {
      if (workers[i].employeeCode == employeeCode)                  C2 * (n − 1)
         return i;                                                  C3 * (n − 1) * max(0, 1)
   }
   return -1;                                                       C4
}
```

$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4$

$T(n) = C_1 n + (C_2 + C_3)(n-1) + C_4$

$T(n) = C_1 n + A(n-1) + C_4,\ donde\ A = C_2 + C_3$

$T(n) = C_1 n + An - A + C_4$

$T(n) = (C_1 + A)n + B,\ donde\ B = C_4 - A$

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

Función delete_in_heap

```
void delete_in_heap(Employee *&workers, int &size, int employeeCode)
{
   int i = search_in_heap(workers, size, employeeCode);          C1
   if (i == -1)                                                  C2
   {
      cout << "Empleado no encontrado.\n";                       C3 * max(0, 1)
      return;                                                    C4 * max(0, 1)
   }
   swap(&workers[i], &workers[size - 1]);                        C5 * O(1)
   size--;                                                       C6
   min_heap(workers, size, i);                                   C7 * O(log2n)
}
```

$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 log_2 n$

$T(n) = A + B log_2 n,\ donde\ A\ y\ B\ son\ constantes$

$\therefore T(n) = O(log_2 n)$

Función modify_heap

```
void modify_heap(Employee *&workers, int &size, int employeeCode, double
newSalary)
{
   int index = search_in_heap(workers, size, employeeCode);          C1 * n
   if (index == -1)                                                  C2
   {
      cout << "Empleado no encontrado.\n";                          C3 * max(0, 1)
      return;                                                       C4 * max(0, 1)
   }
   workers[index].salary = newSalary;                               C5
   min_heap(workers, size, index);                                  C6 * log2n
}
```

$T(n) = C_1 n + C_2 + C_3 + C_4 + C_5 + C_6 log_2 n$

$T(n) = An + B + Clog_2 n,\ donde\ A,\ B\ y\ C\ son\ constantes$

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

Función build_min_heap

```
void build_min_heap(Employee *workers, int size)
{
   for (int i = size / 2 - 1; i >= 0; i--)                    C1 * n
   {
      min_heap(workers, size, i);                             C2 * log2n * (n − 1)
   }
}
```

$T(n) = C_1 n + C_2(n-1)log_2 n$

$T(n) = C_1 n + C_2(nlog_2 n - log_2 n)$

$T(n) = C_1 n + C_2 nlog_2 n - C_2 log_2 n$

$T(n) = An + Bnlog_2 n - Blog_2 n,\ donde\ A\ y\ B\ son\ constantes$

$\therefore T(n) = O(nlog_2 n)$

Función heap_sort

```
void heap_sort(Employee *workers, int size)
{
   build_min_heap(workers, size);                    C1 * nlog2n
   for (int i = size - 1; i > 0; i--)                C2 * n
   {
      swap(&workers[0], &workers[i]);                C3 * (n − 1)
      min_heap(workers, i, 0);                       C4 * (n − 1) * log2n
   }
}
```

$$T(n) = C_1 n log_2 n + C_2 n + C_3(n-1) + C_4(n-1)log_2 n$$

$$T(n) = C_1 n log_2 n + C_2 n + C_3 n - C_3 + C_4(n log_2 n - log_2 n)$$

$$T(n) = C_1 n log_2 n + (C_2 + C_3)n - C_3 + C_4 n log_2 n - C_4 log_2 n$$

$$T(n) = (C_2 + C_3)n + (C_1 + C_4)n log_2 n - C_4 log_2 n - C_3$$

$$T(n) = An + Bn log_2 n - C log_2 n - D,\ donde\ A, B, C\ y\ D\ son\ constantes$$

$$\therefore T(n) = O(n log_2 n)$$

Función read_employees

```
int read_employees(Employee *heapTree)
{
   ifstream inputFile("salarios.txt");                                   C1
   if (!inputFile)                                                       C2
   {
      cout << "Hubo un error" << endl;                                  C3 * max(0, 1)
      return -1;                                                        C4 * max(0, 1)
   }


   int size = 0;                                                         C5
   while (size < NumEmployees && inputFile >> heapTree[size].employeeCode >>
heapTree[size].firstName >> heapTree[size].lastName >> heapTree[size].salary) C6n
   {
      if (heapTree[size].employeeCode <= 0 || heapTree[size].salary < 0)  C7 * (n − 1)
      {
         cout << "Error: datos invalidos en la posicion" << size + 1 << endl; C8 * (n − 1)
         return -1;                                                     C9 * (n − 1) * max(0, 1)
      }
      size++;                                                           C10 * (n − 1)
   }
   inputFile.close();                                                    C11

   return size;                                                          C12
}
```

$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + C_6 n + C_7(n-1) + C_8(n-1) + C_9(n-1) + C_{10}(n-1) + C_{11} + C_{12}$

$T(n) = A + C_6 n + C_7 n - C_7 + C_8 n - C_8 + C_9 n - C_9 + C_{10} n - C_{10}$, *donde A es constante*

$T(n) = A + (C_6 + C_7 + C_8 + C_9 + C_{10})n - (C_6 + C_7 + C_8 + C_9 + C_{10})$

$T(n) = A + Bn - B$, *donde B es constante*

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

Función initialize_employees

```
void initialize_employees(Employee *heapTree, int maxEmployees)
{
   for (int i = 0; i < maxEmployees; ++i)                    C1 * n
   {
      heapTree[i].employeeCode = 0;                          C2 * (n − 1)
      heapTree[i].firstName[0] = '\0';                       C3 * (n − 1)
      heapTree[i].lastName[0] = '\0';                        C4 * (n − 1)
      heapTree[i].salary = 0.0;                              C5 * (n − 1)
   }
}
```

$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5(n-1)$

$T(n) = C_1 n + (C_2 + C_3 + C_4 + C_5)(n-1)$

$T(n) = C_1 n + A(n-1),\ donde\ A\ es\ constante$

$T(n) = C_1 n + An - A$

$T(n) = (C_1 + A)n - A$

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

Función business_logic

```
int business_logic()
{
   Employee heapTree[NumEmployees];                                   C1

   initialize_employees(heapTree, NumEmployees);                      C2

   int size = read_employees(heapTree);                               C3

   if (size <= 0)                                                     C4
   {
      cout << "Archivo abierto pero no se encontraron Datos" << endl;  C5 * max(0, 1)
      return -1;                                                      C6 * max(0, 1)
   }

   heap_sort(heapTree, size);                                         C7 * nlog2n
   cout << "\n******Bienvenido al Sistema de Ordenamiento *****";     C8
   cout << "\nEmpleados ordenados por salario de menor a mayor:\n"   C9
      << endl;

   cout << "Código | Nombre Completo         | Salario" << endl;      C10
   cout << "-----------------------------------------------" << endl; C11

   int cont = 0;                                                      C12
   for (int i = 0; i < size; i++)                                     C13 * n
   {
      cont++;                                                         C14 * (n − 1)
      cout << heapTree[i].employeeCode << "     | "
         << heapTree[i].firstName << " " << heapTree[i].lastName
         << "     | $ " << heapTree[i].salary << "\n";               C15 * (n − 1)
   }

   cout << "\n.............Se Ordenaron: " << cont << " empleados" << endl;  C16
   cout << ".............Hasta Luego:     " << endl;                 C17

   return 0;                                                          C18
}
```

$$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 nlog_2 n + C_8 + C_9 + C_{10} + C_{11} + C_{12} + C_{13} n + C_{14}(n-1) + C_{15}(n-1) + C_{16} + C_{17} + C_{18}$$

$$T(n) = A + C_7 nlog_2 n + C_{13} n + (C_{14} + C_{15})(n-1),\ donde\ A\ es\ constante$$

$$T(n) = A + C_7 nlog_2 n + C_{13} n + B(n-1),\ donde\ B\ es\ constante$$

$$T(n) = A + C_7 nlog_2 n + C_{13} n + Bn - B$$

$$T(n) = A + C_7 nlog_2 n + (C_{13} + B)n - B$$

$$T(n) = Dnlog_2 n + En - F,\ donde\ D,\ E\ y\ F\ son\ constantes$$

$$\therefore T(n) = O(nlog_2 n)$$

Función main

```
int main()
{
   business_logic();          C1 * nlog2n
   return 0;                  C2
}
```

$$T(n) = C_1 nlog_2 n + C_2$$

$$\therefore T(n) = O(nlog_2 n)$$

Por lo tanto se concluye que la complejidad algorítmica de todo el código implementado es $O(nlog_2 n)$

Reflexión final

Al ordenar estos datos con heapsort, destaca la notable diferencia en los salarios entre los empleados. En la lista que se utilizó para el ejemplo, Mateo Rios parece ser uno de las personas con mayor salario, siendo de $7534.70. Este análisis no solo ayuda a identificar a los empleados con mayores ingresos, sino que también proporciona una base sólida para revisar políticas salariales y asegurar una distribución justa teniendo como base una cantidad igual o similar de trabajo.

Los resultados permiten visualizar la estructura salarial y detectar posibles anomalías. Además, se identifican patrones en la distribución salarial, lo que puede guiar decisiones estratégicas para mejorar la equidad y la motivación del personal por una mejora salarial.

Implementar el heapsort en la gestión de salarios no solo facilita el control y orden, sino que también proporciona una herramienta poderosa para tomar decisiones basadas en datos obtenidos de una investigación. Ayuda a los encargados a visualizar de manera clara y eficiente la estructura salarial y a tomar acciones teniendo en cuenta reportes e informes que pueden mejorar la satisfacción y eficacia de los empleados. Esta herramienta es fundamental para garantizar la competencia y el buen manejo en la administración de recursos humanos.