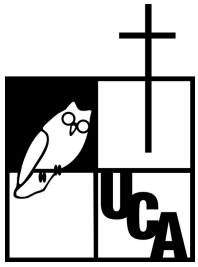
Universidad Centroamericana
José Simeón Cañas



Taller 2

Análisis de Algoritmos

Ing. Mario López

Ing. Enmanuel Araujo

Integrantes:

Graciela Maria Osegueda Hernandez,	00153822
Diego Josué Rosa Pacheco,	00007122
Fernanda Camila Vásquez Meléndez,	00065221

Fecha de entrega:
sábado 12 de octubre del 2024

Orden de magnitud

Código no recursivo

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
O(1)
O(1)
O(1)
```

Análisis de complejidad: Cada iteración tiene un costo O(1), ya que sólo se ejecutan una vez.

```
class MaxHeap {
  int* heap;
  int size;
  int capacity;
```

```
O(1)
O(1)
O(1)
```

Análisis de complejidad: Cada iteración tiene un costo O(1), ya que sólo se ejecutan una vez.

```
public:
  MaxHeap(int cap) : size(0), capacity(cap) {
    heap = new int[cap]; //depende del tamaño de cap
  }
```

```
O(1)
O(cap)
```

Análisis de complejidad: O(1) para las asignaciones, pero la asignación dinámica de memoria es O(cap) ya que se reserva espacio en memoria para cap elementos.

```
  ~MaxHeap() {
    delete[] heap;
  }
```

```
O(1)
```

Análisis de complejidad: Cada iteración tiene un costo O(1), ya que sólo se ejecutan una vez.

```
void insertar(int value) {
  if (size == capacity) {
    cout << "El Heap está lleno" << endl;
    return;
  }
  int i = size;
  heap[size++] = value;

  while (i != 0 && heap[padre(i)] < heap[i]) {
    int temp = heap[i];
    heap[i] = heap[padre(i)];
    heap[padre(i)] = temp;
    i = padre(i);
  }
}
```

```
O(1)
O(1)
O(1)

O(1)
O(1)

O(log n)
en el peor
de los
casos
```

Análisis de complejidad: O(log n), ya que en el peor de los casos, el valor subirá hasta la raíz del heap, lo que implica que recorrer toda la altura del árbol que es log(n).

```
int eliminarMax() {
  if (size <= 0) return -1;
  int maxElemento = heap[0];
  heap[0] = heap[--size];
  reajustarHeap(0);
  return maxElemento;
}
```

```
O(1)
O(1)
O(1)
O(log n)
O(1)
```

Análisis de complejidad: O(log n), debido a que la función reajustarHeap tiene esa complejidad en el peor de los casos.

```
void ordenarDescendente() {
  while (size > 0) {
    cout << eliminarMax() << " ";
  }
  cout << endl;
}
```

```
O(log n)

O(1)
```

Análisis de complejidad: Cada llamada a eliminarMax es O(log n), por lo que eliminar todos los elementos toma O(n log n).

```cpp
int padre(int i) { return (i - 1) / 2; }
int hijoIzquierdo(int i) { return (2 * i) + 1; }
int hijoDerecho(int i) { return (2 * i) + 2; }
```

```
O(1)
O(1)
O(1)
```

Análisis de complejidad: Cada iteración tiene un costo O(1), ya que sólo se ejecutan una vez.

```cpp
int main() {
  MaxHeap heap(200000);

  ifstream archivo("../Software/salarios.txt");
  int salario;

  if (!archivo.is_open()) {
    cout << "Error al abrir el archivo de salarios." << endl;
    return 1;
  }
```

```
O(1)

O(1)
O(1)


O(1)
O(1)
O(1)
```

Análisis de complejidad: Cada iteración tiene un costo O(1), ya que sólo se ejecutan una vez.

```cpp
  while (archivo >> salario) {
    heap.insertar(salario);
  }
  archivo.close();
```

```
O(log n)

O(1)
```

Análisis de complejidad: Cada inserción es O(log n), por lo que si hay n salarios, el bucle tiene complejidad O(n log n).

```
int opcion;                                                    O(1)
do {
  cout << "\n--------- Menu  ---------" << endl;               O(1)
  cout << "1. Ordenar y mostrar salarios en orden descendente"
    << endl;                                                   O(1)
  cout << "2. Salir" << endl;                                  O(1)
  cout << "Seleccione una opcion: ";                           O(1)
  cin >> opcion;                                               O(1)
```

Análisis de complejidad: Cada iteración tiene un costo O(1), ya que sólo se ejecutan una vez.

```
switch (opcion) {
  case 1:
    cout << "Salarios en orden descendente:" << endl;          O(1)
    heap.ordenarDescendente();                                 O(log n)
    break;                                                     O(1)
  case 2:
    cout << "Saliendo del programa." << endl;                  O(1)
    break;                                                     O(1)
  default:
    cout << "Opcion no valida. Por favor, intente de nuevo."   O(1)
      <<  endl;
    break;                                                     O(1)
  }
} while (opcion != 2);
  return 0;                                                    O(1)
}
```

Análisis de complejidad: O(1) para el manejo de la opción, pero debido al heap.ordenarDescendente() es O(n log n).

Código recursivo

```
void reajustarHeap(int i) {
  int mayor = i;                                          O(1)
  int izquierdo = hijoIzquierdo(i);                       O(1)
  int derecho = hijoDerecho(i);                           O(1)

  if (izquierdo < size && heap[izquierdo] > heap[mayor])  O(1)
    mayor = izquierdo;
  if (derecho < size && heap[derecho] > heap[mayor])      O(1)
    mayor = derecho;

  if (mayor != i) {                                       O(log n)
    int temp = heap[i];                                   O(1)
    heap[i] = heap[mayor];                                O(1)
    heap[mayor] = temp;                                   O(1)
    reajustarHeap(mayor);                                 O(log n)
  }
}
```

Análisis de complejidad: En el peor caso, la función recursiva se llamará a sí misma hasta que el elemento llegue a una hoja, lo cual implica realizar el ajuste desde la raíz hasta el nivel más profundo del heap. El heap es un árbol binario, por lo tanto su altura es proporcional a log n.

Recurrencia:

$$T(n) = T(n/2) + O(1)$$

Se envían n/2 de datos cada vez ya que representan las dos mitades (la mitad izquierda o derecha del heap) en las que se divide el heap, y la complejidad de ajustar un nivel es constante por eso es O(1).

Teorema maestro:

$$T(n) = T(n/2) + O(1)$$

En nuestro caso:

a = 1,solo se hace una llamada recursiva.

b = 2, debido a la división del heap.

d = 0, debido a que todo lo demás es constante.

Al sustituir en el Teorema Maestro, nos damos cuenta que corresponde es el caso 2:

$$d = log_b(a)$$

$$0 = log_2(1)$$

$$0 = 0$$

Por lo tanto, queda:

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

Por lo tanto, la complejidad de reajustarHeap es O(log n).

Complejidad Total

Por lo tanto podemos concluir que nuestro código es:

$$T(n) = T[n\ (log(n))] + O(1)$$

Análisis de los resultados

En base a la implementación de la solución propuesta para la problemática presentada por el Almacén de Salem, se identifican resultados positivos; ya que, se logró de forma exitosa el objetivo principal, que es: ordenar y presentar los salarios en orden descendente. Esto fue posible gracias a la implementación de estructuras de datos avanzadas que facilitan la resolución óptima del problema.

Se identificó que el uso de la estructura de datos MaxHeap es certera; debido a que, permite organizar una gran cantidad de información en orden descendente de forma eficiente. Ofreciendo una complejidad de O(log n) para la eficiencia del algoritmo implementado.

Como conclusión, es importante destacar el impacto positivo que esta solución de software puede tener para las operaciones del almacén. Disponer de la información salarial ordenada adecuadamente no solo mejora la visualización y análisis de datos; sino que, adicionalmente, es crucial para la toma de decisiones estratégicas como: ajuste salariales o distribución de recursos disponibles.