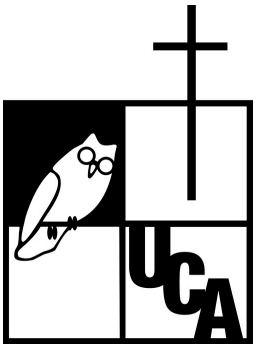
Universidad Centroamericana José Simeón Cañas

Facultad de Ingeniería y Arquitectura
Análisis de Algoritmos



Taller N° 2

Software para almacen Salem 2

Integrantes
Valeria Michelle Barbero Rivera, 00121522
Gabriel Enrique Cortez Joya, 00037021
Carlos Andrés Rodriguez Novoa, 00402720

Profesores
Enmanuel Araujo & Mario Lopez

Antiguo Cuscatlán, 12 de Octubre 2024

● No recursivo

● Recursivo

```
#include <iostream>
#include <string>
#include "BaseEmpleados.h"

using namespace std;

//Basado en que el archivo .h tendrá 1000 registros by default
//Varia en base a los registros agregados y eliminados de la lista
int extraCount = 1000;

void printEmployees(){
    cout << "   ID   ||      Nombre      ||      Salario      ||\n";
    for (int p = extraCount; p >= 0 ; p--)
    {
        if(employees[p].name != ""){
        cout << "   " << p << "         " << employees[p].name << "         " << employees[p].salary
<< "\n";
        }
    }
}

//Función temporal de prueba
void viewEmployee(int id){
    cout << "Nombre: " << employees[id].name << "\n";
    cout << "Salario: " << employees[id].salary << "\n";
}

void orderHeap(Employee emp[], int n, int i) {
    int largest = i;        // Inicializamos la raíz como el valor mas grande
    int left = 2 * i + 1;     // definimos el hijo derecho de la raíz, en caso 0 toma posicion 1
    int right = 2 * i + 2;    // Definimos el hijo izquierdo de la raíz, en caso 0 toma posicion 2

    // Si el hijo izquierdo es el más grande
    if (left < n && emp[left].salary > emp[largest].salary)
        largest = left;

    // si el hijo derecho es el mas grande de todos
    if (right < n && emp[right].salary > emp[largest].salary)
        largest = right;

    // SI una vez comparados la raiz escogida al principio no es la mayor
    if (largest != i) {
        swap(emp[i], emp[largest]);

        // Volvemos a llamar para arreglar en base a los cambios que han ocurrido
        orderHeap(emp, n, largest);
    }
}
```

O(n)

O(1)

O(lgn)

Debido a que la altura del árbol binario es lgn y en el peor caso la función debe recorrer desde la raíz hasta la hoja más profunda del árbol

Cada llamada a orderHeap es O(lg n) por lo explicado anteriormente

```
void doMaxHeap(Employee emp[]) {
    int n = extraCount;

    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        orderHeap(emp, n, i);
    }

    // One by one extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(emp[0], emp[i]);  // Move current root to end
        orderHeap(emp, i, 0);   // Re-heapify the reduced heap
    }

    // Optional: Print the sorted employees
    printEmployees();  // This will now display from largest to smallest salary
}

//Definicion de funciones
void showDescending(){
    /* ¿Como ordenar el listado para mostrar en orden descendiente? Min heap*/
    doMaxHeap(employees);
    //una vez ordenado muestra el listado
}

void addEmployee(string n, float s){
    /*Agregar empleado a la lista del archivo .h*/
    employees[extraCount].name = n;
    employees[extraCount].salary = s;
    extraCount++;
}

void updateEmployee(int id, string n, float s){
    /*Reingresar la informacion del empleado*/
    employees[id].name = n;
    employees[id].salary = s;
}

void removeEmployee(int id){
    /*Eliminar al empleado de la lista*/
    for (int i = id+1 ; i < extraCount; i++)
    {
        employees[i - 1] = employees[i];
    }
    extraCount--;
}
```

n/2 - 1 → O(n)
O(lg n)
O(n lg n)
O(n)
O(1)
O(1)
O(lg n)
O(n lg n)
O(n)
O(n lg n)
porque depende de doMaxHeap
O(1)
O(1)
O(1)
O(1)
O(1)
O(n - id)
n - 1 → O(n)
O(1)
O(1)
O(n)

```
int main(){

    int option = 0;

    string name = "";
    float salary = 0.00;

    int id = 0;

    do
    {
        //Menu de opciones
        cout << "-------------------------------------------------\n";
        cout << "Seleccione una opcion para continuar:\n";
        cout << "1. Mostrar listado en orden descendiente\n";
        cout << "2. Agregar nuevo empleado\n";
        cout << "3. Modificar empleado existente\n";
        cout << "4. Eliminar empleado\n";
        cout << "0. Terminar\n";
        cin >> option;
        cout << "-------------------------------------------------\n";

        switch (option)
        {
        case 1:
            /* Hacer lo de min heap */
            cout << "Mostrando los empleados en orden descendiente basado en salario\n";
            showDescending();
            break;
        case 2:
            /* Agregar nuevo empleado */
            cout << "Favor ingresar los datos del nuevo empleado\n";
            cout << "Nombre "; cin >> name;
            cout << "Salario "; cin >> salary;
            addEmployee(name, salary);
            cout << "Nuevo empleado ingresado: " << name <<"\n";
            break;
        case 3:
            /* modificar empleado */
            cout << "Ingresar el numero identificador del empleado "; cin >> id;
            cout << "Nuevo nombre "; cin >> name;
            cout << "Nuevo salario "; cin >> salary;
            updateEmployee(id, name, salary);
            cout << "Se ha modificado correctamente el empleado\n";
            break;
        case 4:
            /* Emliminar empleado */
```

```
            cout << "Ingresar el numero identificador del empleado a eliminar "; cin >> id;
            removeEmployee(id);
            cout << "Se ha eliminado correctamente el usuario\n";
            break;

        default:
            break;
        }
    } while (option != 0);

    return 0;
}
```

O(1)
O(n)
O(n)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)

La complejidad de la función main está determinada por las funciones que se ocupan en cada opción, para mostrar empleados en orden descendente, debido a showDescending la complejidad es $O(n \lg n)$, para agregar y modificar empleados, $O(1)$, y eliminar empleados $O(n)$

En el peor caso que sería cuando el usuario elige mostrar los empleados (case 1) y luego elimina un empleado (case 4), la complejidad sería

$$O(n \lg n) + O(n) = O(n \lg n)$$

Resolviendo la Recurrencia

Partiendo de que la estructura es de un montículo (heap) donde un nodo actual (i) se compara con sus hijos, la recurrencia que describe esto es:

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

Teorema Maestro

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

$$a \geq 0 \quad \wedge \quad b > 1 \quad \wedge \quad d \geq 0$$

$$T(n) \begin{cases} O(n^d) & , d > \log_b a \\ O(n^d \log n) & , d = \log_b a \\ O(n^{\log_b a}) & , d < \log_b a \end{cases}$$

$$a = 1 \quad b = 2 \quad d = 0$$

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

$$T(n) = O(n^0 \log n)$$

$$\boxed{T(n) = O(\log n)}$$

Pero debido a que en la función doMaxHeap recorre el array n veces y vuelve a llamar a orderHeap en cada iteración toma $\boxed{T(n) = O(n \log n)}$

Ensayo

En cuanto a los resultados obtenidos por el equipo, se puede concluir que:

Se ha logrado obtener una complejidad de **O(n log n)** lo que representa un rendimiento óptimo en la implementación del código y esto permitió ordenar eficientemente los registros de los empleados en orden descendente de acuerdo a sus salarios, facilitando el manejo de grandes cantidades de datos justo como este caso siendo un total de 1000 empleados.
Añadido a esto, se decidió usar la estructura de datos Max-Heap lo que aseguró que la búsqueda del salario más alto fuera rápida, ya que siempre estaba localizado en la raíz del montículo.

Otro aspecto importante de los resultados es que el sistema mantuvo su eficiencia y rendimiento incluso cuando se realizaron múltiples operaciones de modificación en los registros, como agregar o eliminar empleados. Cada operación fue llevada a cabo sin afectar significativamente la eficiencia del algoritmo. Esto muestra que el uso de Heap no solo fue útil para el ordenamiento, sino también para garantizar que las estructuras subyacentes permanecieran correctamente equilibradas durante las actualizaciones, lo que contribuyó al rendimiento general del sistema.

En resumen, el equipo consiguió implementar un sistema funcional y eficiente, donde la estructura Max-Heap se desempeñó como un componente clave para cumplir con todos los objetivos, logrando un balance adecuado de rendimiento y funcionalidad en las operaciones ejecutadas sobre la lista de empleados.