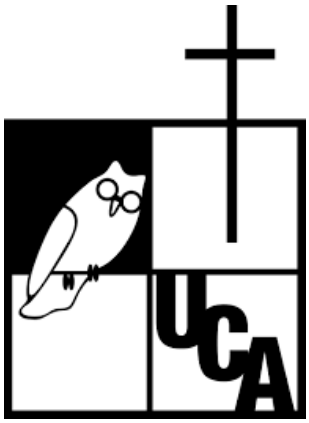


Universidad Centroamericana "José Simeón Cañas"

Facultad de Ingeniería y Arquitectura

Implementación de estructura de datos Heap

Integrantes del equipo

Eduardo Alessandro Rivera Diaz

Kevin Alexander Mejia Hercules

Catedráticos:

Enmanuel Amaya Araujo, MSc.

Ing. Mario Isaac López.

INDICE

Introducción

El presente trabajo se centra en el análisis y diseño de un programa que optimiza la ordenación de salarios de empleados del almacén Salem. Utilizando el algoritmo de heapsort, que destaca por su complejidad temporal de $O(n \log n)$ el programa asegura una ejecución rápida y eficiente, incluso cuando se enfrenta a conjuntos de datos extensos. A lo largo del proyecto, se implementaron diferentes módulos que abarcan desde la carga de datos hasta la visualización de resultados, cada uno diseñado para maximizar la eficiencia y minimizar el uso de recursos, en línea con los requisitos del proyecto.

Además de la implementación técnica, se llevó a cabo un análisis exhaustivo de la complejidad de cada función, lo que permitió comprender mejor el rendimiento del programa y el impacto de las decisiones algorítmicas en la experiencia del usuario final. Este proceso de análisis no solo subraya la importancia de elegir los algoritmos adecuados en la programación, sino que también destaca la necesidad de una planificación meticulosa y la implementación de soluciones adecuadas en el desarrollo de software. A través de este trabajo, se evidencian las prácticas efectivas de ingeniería de software, que permiten abordar desafíos complejos y ofrecer soluciones que cumplen con las expectativas de eficiencia y funcionalidad.

Descripción del programa

El presente programa está diseñado para gestionar y ordenar la información de empleados del almacén Salem. Se cargan los datos de los empleados desde un archivo de texto (nombrado empleados_salem.txt), donde cada línea contiene el nombre del empleado, su puesto de trabajo y su ingreso salarial. Esta información se almacena en una estructura llamada Empleado, que facilita el manejo de los datos.

El principal objetivo del programa es ordenar los salarios de los empleados en orden descendente, lo cual es fundamental para facilitar el cálculo de los bonos de fin de año, comenzando por aquellos con mayores ingresos. Para lograr esto, el programa utiliza el algoritmo de heapsort, que es un método eficiente basado en la estructura de datos conocida como heap, específicamente un max heap, permitiendo extraer el mayor elemento de manera eficiente. Además, el programa presenta un menú simple en la consola, donde el usuario puede elegir opciones, como ordenar los salarios de los empleados. Una vez que se realiza la ordenación, se muestran los datos de los empleados en una tabla formateada, destacando el nombre, el puesto y el ingreso de cada uno. Se ha implementado con un enfoque en la eficiencia tanto en el uso de memoria como en el tiempo de ejecución, considerando que la empresa ha crecido considerablemente y puede tener miles de empleados.

Archivos y funciones del programa

Para brindar solución a la solicitud del almacén Salem, se han creado varios archivos que facilitan la modularización del programa. Cada archivo contiene funciones específicas que contribuyen a la organización y claridad del código, permitiendo un mejor mantenimiento y comprensión del mismo. A continuación, se detallan las funciones de cada archivo:

reader.hpp:

```
1  // Define la estructura para almacenar la informacion del empleado
2  struct Empleado{
3      string nombre, puesto;
4      float ingreso;
5  };
```

- **struct Empleado:** Se define una estructura que almacena la información de un empleado, incluyendo su nombre, puesto y salario.

```
1  // Carga los datos de los empleados desde el archivo
2  void cargar_datos(Empleado *datos, const int filas = 1000)
3  {
4      string nombre, puesto, ingresoStr;
5      const int numFilas = filas;
6
7      // Abre el archivo
8      ifstream archivo("empleados_salem.txt");
9
10     if (!archivo.is_open())
11     {
12         cerr << "No se pudo abrir el archivo." << endl;
13         return;
14     }
15
16     int i = 0;
17     while (getline(archivo, nombre, '\t'))
18     {
19         datos[i].nombre = nombre;
20         getline(archivo, puesto, '\t');
21         datos[i].puesto = puesto;
22         getline(archivo, ingresoStr, '\n');
23         datos[i].ingreso = stof(ingresoStr);  // Convertir el ingreso a float
24         ++i;
25     }
26
27     // Cierra el archivo
28     archivo.close();
29 }
```

- **void cargar_datos(Empleado *datos, const int filas = 1000):** Funcion para cargar los datos de los empleados desde un archivo de texto (empleados_salem.txt). Lee cada línea y almacena la información en un arreglo de estructuras Empleado.

heapsort.hpp:

```
1  // Construir un MaxHeap para ordenar
2  void construirMaxHeap(Empleado *array, int tamano){
3      for (int i = tamano / 2 - 1; i >= 0; i--)
4          maxHeapify(array, i, tamano);
5  }
```

- **void construirMaxHeap(Empleado *array, int tamano)**: Función para construir un max heap a partir de un arreglo de empleados, reestructurando el arreglo para cumplir con las propiedades del heap.

```
1  // Funcion de maxHeapify
2  void maxHeapify(Empleado *heap, int i, int tamano)
3  {
4      int mayor = i;
5      int izquierda = 2 * i + 1;
6      int derecha = 2 * i + 2;
7
8      if (izquierda < tamano && heap[izquierda].ingreso > heap[mayor].ingreso)
9          mayor = izquierda;
10
11     if (derecha < tamano && heap[derecha].ingreso > heap[mayor].ingreso)
12         mayor = derecha;
13
14     if (mayor != i)
15     {
16         swap(heap[i], heap[mayor]);
17         maxHeapify(heap, mayor, tamano);
18     }
19 }
```

- **void maxHeapify(Empleado *array, int i, int tamano)**: Función para mantener la propiedad de max heap en un subárbol. Si el nodo en el índice i no cumple con la propiedad del heap, realiza intercambios recursivos para corregirlo.

```
// Funcion para ordenar usando el heap
void ordenarHeap(Empleado *array, int tamano){
    construirMaxHeap(array, tamano);

    for (int i = tamano - 1; i > 0; i--)
    {
        swap(array[0], array[i]); // Mueve el mayor al final del array
        maxHeapify(array, 0, i);  // Ajusta el heap
    }
}
```

- **void ordenarHeap(Empleado *array, int n)**: Función para ordenar un arreglo de empleados utilizando el algoritmo heapsort.

heap.hpp:

```
// Funcion de maxHeapify
void maxHeapify(Empleado *heap, int i, int tamano)
{
    int mayor = i;
    int izquierda = 2 * i + 1;
    int derecha = 2 * i + 2;

    if (izquierda < tamano && heap[izquierda].ingreso > heap[mayor].ingreso)
        mayor = izquierda;

    if (derecha < tamano && heap[derecha].ingreso > heap[mayor].ingreso)
        mayor = derecha;

    if (mayor != i)
    {
        swap(heap[i], heap[mayor]);
        maxHeapify(heap, mayor, tamano);
    }
}
```

- **void maxHeapify(Empleado *heap, int i, int tamano):** Función para mantener la propiedad de max heap en el subárbol con raíz en el índice i. Si el nodo en el índice i es menor que alguno de sus hijos, intercambia el nodo con el hijo mayor y aplica recursivamente la misma lógica en el subárbol afectado.

```
// Inserta un nuevo empleado en el heap
void insertarHeap(Empleado *heap, Empleado nuevo, int &tamano)
{
    tamano++;
    int i = tamano - 1;
    heap[i] = nuevo;

    while (i != 0 && heap[(i - 1) / 2].ingreso < heap[i].ingreso)
    {
        swap(heap[i], heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}
```

- **void insertarHeap(Empleado *heap, Empleado nuevo, int &tamano)**: Inserta un nuevo empleado en el heap. Aumenta el tamaño del heap y coloca el nuevo empleado al final.

```
// Busca un empleado por su nombre en el heap
int buscarHeap(Empleado *heap, string nombre, int tamano)
{
    for (int i = 0; i < tamano; i++)
    {
        if (heap[i].nombre == nombre)
            return i;
    }
    return -1; // No encontrado
}
```

- **int buscarHeap(Empleado *heap, string nombre, int tamano):** Busca un empleado por su nombre en el heap. Recorre el heap desde la raíz hasta el tamaño actual, comparando el nombre de cada empleado con el nombre buscado. Si encuentra el empleado, devuelve su índice; de lo contrario, devuelve -1 para indicar que no fue encontrado.

```
// Modifica el salario de un empleado en el heap
void modificarIngresoHeap(Empleado *heap, int indice, float nuevoIngreso, int tamano)
{
    heap[indice].ingreso = nuevoIngreso;

    while (indice != 0 && heap[(indice - 1) / 2].ingreso < heap[indice].ingreso)
    {
        swap(heap[indice], heap[(indice - 1) / 2]);
        indice = (indice - 1) / 2;
    }
    maxHeapify(heap, indice, tamano);
}
```

- **void modificarIngresoHeap(Empleado *heap, int indice, float nuevoIngreso, int tamano):** Modifica el salario de un empleado en el heap en el índice especificado. Actualiza el ingreso del empleado y realiza el ajuste necesario hacia arriba (si el nuevo salario es mayor) y hacia abajo (usando maxHeapify), asegurando que se mantenga la propiedad de max heap.

main.cc:

- **int main(int argc, char *argv[]):** Función principal del programa. Carga los datos de empleados, llama a la función para ordenar los salarios y presenta un menú para que el usuario interactúe con el programa. Después de la ordenación, muestra los datos de los empleados en una tabla formateada.

main.cc:

```
1  int main(int argc, char *argv[]){
2
3      Empleado datos[Filas];
4      int tamano = 0; // Para manejar el tamaño dinamico del heap
5
6      cargar_datos(datos, Filas);
7      tamano = Filas; // El tamaño se iguala al numero de filas cargadas
8
9      int opcion = 0;
10     do{
11         cout << "\n\t=============== Almacenes Salem ===============\n" << endl;
12         cout << "1. Ordenar salarios" << endl;
13         cout << "2. Salir" << endl;
14         cout << "\nSelecciona una opcion: ";
15         cin >> opcion;
16
17         switch (opcion)
18         {
19         case 1:
20             // Ordenar los empleados por salario
21             ordenarHeap(datos, tamano);
22
23             // Mostrar los datos ordenados
24             cout << left << setw(25) << "Nombre"
25                  << setw(30) << "Puesto"
26                  << setw(15) << "Ingreso" << endl;
27             cout << "-------------------------------------------------------------" << endl;
28
29             for (int i = tamano - 1; i >= 0; i--)
30             {
31                 cout << left << setw(25) << datos[i].nombre
32                      << setw(30) << datos[i].puesto
33                      << "$ " << setw(14) << fixed << setprecision(2) << datos[i].ingreso << endl;
34             }
35             break;
36
37         case 2:
38             cout << "Saliendo del programa..." << endl;
39             break;
40
41         default:
42             cout << "Opcion invalida....." << endl;
43             break;
44         }
45
46     } while (opcion != 2);
47
48     return 0;
49 }
50
```

Funciones recursivas

1. **Función maxHeapify:** Esta función es recursiva porque se llama a sí misma en la línea donde se realiza el intercambio (swap).

Funciones no recursivas

1. void cargar_datos(Empleado *datos, const int filas = 2000)
2. void construirMaxHeap(Empleado *array, int tamano);
3. void ordenarHeap(Empleado *array, int tamano)
4. void insertarHeap(Empleado *heap, Empleado nuevo, int &tamano);
5. int buscarHeap(Empleado *heap, string nombre, int tamano);
6. void modificarIngresoHeap(Empleado *heap, int indice, float nuevoIngreso, int tamano);

Análisis de orden de magnitud

Análisis del archivo reader.hpp

Las definiciones de librerías y la definición de una estructura al ser operaciones primitivas tienen un orden de magnitud **O(1).**

```
1  #ifndef READER_HPP
2  #define READER_HPP
3
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7
8  using namespace std;
9
10 // Define la estructura para almacenar la informacion del empleado
11 struct Empleado{
12     string nombre, puesto;
13     float ingreso;
14 };
```

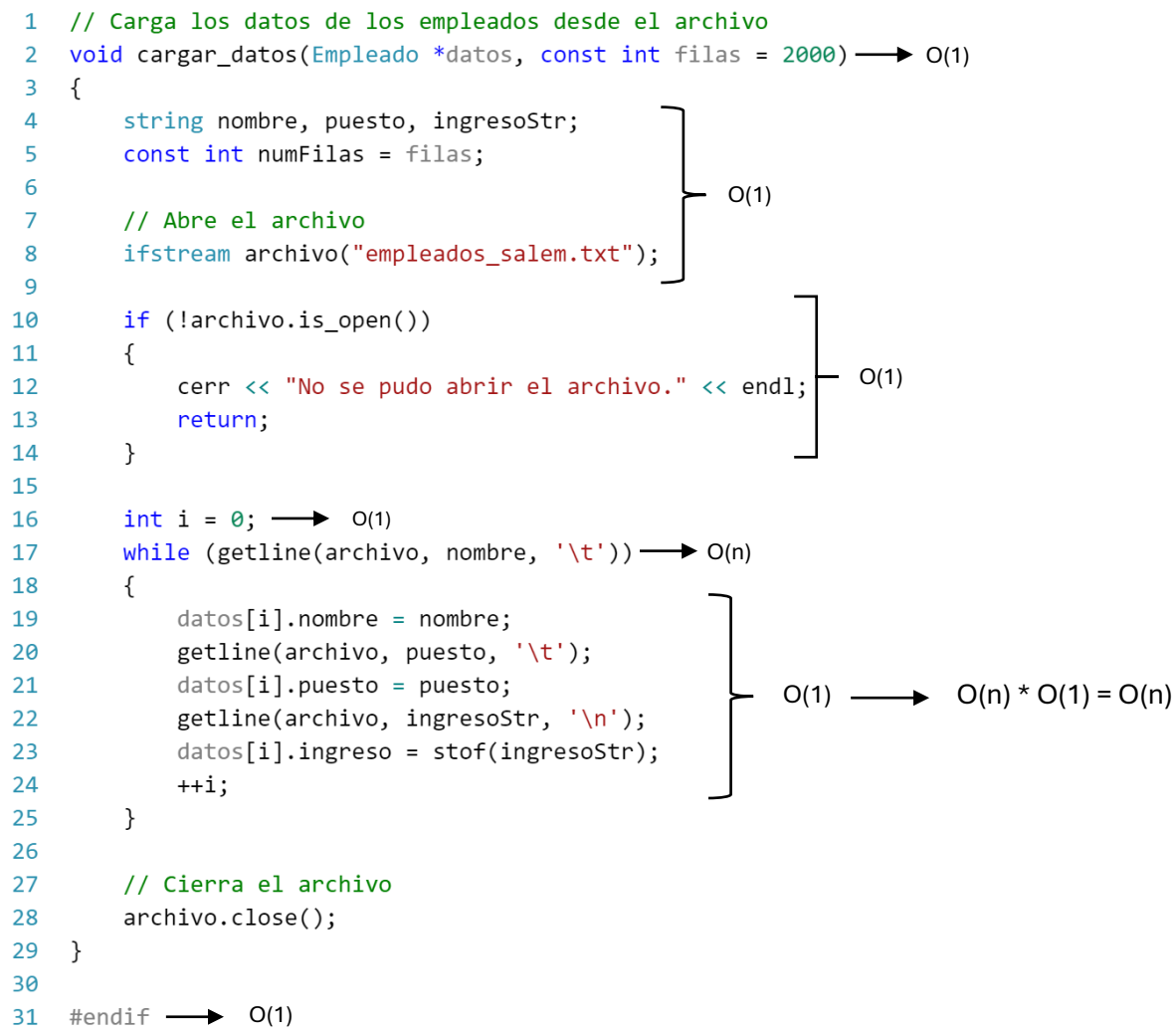
O(1) (líneas 1–6)

O(1) (línea 8)

O(1) (línea 11)

O(1) (líneas 12–13)

O(1) * O(1) = O(1)



Para este caso iniciaremos nuestro análisis de adentro hacia afuera, la declaración de variables tienen un orden de magnitud de *O(1)*, El cuerpo del if al ser una operación primitiva, mostrar en pantalla, tiene un orden de magnitud *O(1)* sentencia if al ser operación primitiva su orden de magnitud es *O(1)* por lo tanto al ser operaciones anidadas el orden de magnitud del if es *O(1) * O(1) = O(1)*. Dentro del while tenemos operaciones de obtener datos que dentro del while solo se ejecutan una vez, por lo tanto su orden de magnitud es *O(1)* mientras que el while se ejecutará de manera lineal ya que se detendrá hasta haber obtenido la última línea del archivo, por lo tanto su orden de magnitud es *O(n)* operaciones anidadas se multiplican, por lo tanto *O(1) * O(n) = O(n)*. cerrar el archivo tiene un orden de magnitud de *O(1)*.

El orden de magnitud de la función cargar_datos seria: *O(1)+O(1)+O(1)+O(n)+O(1) =* ***O(n)***

Cerrar la directiva del preprocesador del procesador tiene un orden de magnitud de *O(1)* ya que es una operación primitiva.

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

Por lo que nuestro archivo reader.hpp tiene un orden de magnitud **O(n).**

Análisis del archivo heap.hpp

```
1  #ifndef HEAP_HPP                                                                  } O(1)
2  #define HEAP_HPP                                                                  }
3
4  #include "reader.hpp"                                                             ---> O(1)
5
6  // Declaraciones de funciones                                                     }
7  void maxHeapify(Empleado *heap, int i, int tamano);                               }
8  void insertarHeap(Empleado *heap, Empleado nuevo, int &tamano);                   } O(1)
9  int buscarHeap(Empleado *heap, string nombre, int tamano);                        }
10 void modificarIngresoHeap(Empleado *heap, int indice, float nuevoIngreso, int tamano); }
11
12 // Inserta un nuevo empleado en el heap
13 void insertarHeap(Empleado *heap, Empleado nuevo, int &tamano)
14 {
15     tamano++;                                                                     }
16     int i = tamano - 1;                                                           } O(1)
17     heap[i] = nuevo;                                                              }
18
19     while (i != 0 && heap[(i - 1) / 2].ingreso < heap[i].ingreso)                 ---> O(log n)
20     {
21         swap(heap[i], heap[(i - 1) / 2]);                                         } O(1)
22         i = (i - 1) / 2;                                                          }
23     }
24 }
```

Resolución de la función recursiva maxHeapify

```
1  // Funcion de maxHeapify
2  void maxHeapify(Empleado *heap, int i, int tamano) ⟶ O(1)
3  {
4      int mayor = i;
5      int izquierda = 2 * i + 1;  } O(1)
6      int derecha = 2 * i + 2;
7
8      if (izquierda < tamano && heap[izquierda].ingreso > heap[mayor].ingreso) ⟶ O(1)
9          mayor = izquierda; ⟶ O(1)
10
11     if (derecha < tamano && heap[derecha].ingreso > heap[mayor].ingreso) ⟶ O(1)
12         mayor = derecha; ⟶ O(1)
13
14     if (mayor != i) ⟶ O(1)
15     {
16         swap(heap[i], heap[mayor]); ⟶ O(1)
17         maxHeapify(heap, mayor, tamano); ⟶ T(n/2)
18     } ⟶ O(1)
19 } ⟶ O(1)
```

Razonamiento para definir cada pieza de la recurrencia: maxHeapify se ejecuta cuando es necesario realizar un intercambio entre el nodo actual y uno de sus hijos, lo que ocurre si uno de los hijos tiene un valor mayor que el nodo actual. Luego de hacer el intercambio, la función se vuelve a llamar para el subárbol donde se realizó el cambio, continuando con el proceso de reorganización del heap.

Las operaciones involucradas en esta función, cómo calcular los índices de los hijos, hacer comparaciones entre el nodo actual y sus hijos, y realizar el intercambio cuando es necesario, son operaciones primitivas por lo tanto tienen una magnitud de O(1).

La llamada recursiva a maxHeapify tiene un tiempo de ejecución de T(n/2) porque, tras un intercambio, el algoritmo se aplica a uno de los subárboles del nodo que fue intercambiado. Dado que estamos trabajando con un árbol binario completo, cada subárbol tiene aproximadamente la mitad del tamaño del árbol original. Esto significa que en cada paso, el tamaño del problema se reduce a la mitad (n/2).

Resolviendo la recurrencia mediante Teorema Maestro

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

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

$$log_b a = log_2 1 = 0$$

$$O = O \rightarrow T(n) = O(n^d log_n)$$

$$\boxed{\therefore O(\log n)}$$

Por lo tanto, la recurrencia $T(n) = T\left(\frac{n}{2}\right) + O(1)$ refleja correctamente que el orden de magnitud de la función maxHeapify es $\boldsymbol{O(log\ n)}$.

```
1  // Busca un empleado por su nombre en el heap
2  int buscarHeap(Empleado *heap, string nombre, int tamano)
3  {
4      for (int i = 0; i < tamano; i++) ──► O(n)
5      {
6          if (heap[i].nombre == nombre) ──► O(1)
7              return i; ──► O(1)
8      }
9      return -1; // No encontrado ──► O(1)
10 }
11
12 // Modifica el salario de un empleado en el heap
13 void modificarIngresoHeap(Empleado *heap, int indice, float nuevoIngreso, int tamano)
14 {
15     heap[indice].ingreso = nuevoIngreso; ──► O(1)
16
17     while (indice != 0 && heap[(indice - 1) / 2].ingreso < heap[indice].ingreso) ──► O(log n)
18     {
19         swap(heap[indice], heap[(indice - 1) / 2]); ──► O(1)
20         indice = (indice - 1) / 2; ──► O(1)
21     }
22     maxHeapify(heap, indice, tamano); ──► O(log n)
23 }
```

```
1  #endif ──► O(1)
```

Los archivos de encabezado, tienen un orden de magnitud O(1), ya que no dependen del tamaño de la entrada y son operaciones primitivas del compilador.

La declaración de las funciones no ejecuta ningún código, por lo que su orden de magnitud es O(1).

Análisis de la función insertarHeap: Las primeras tres líneas (incremento de tamano, asignación de i y asignación de heap[i]) tienen un orden de magnitud O(1), ya que son operaciones primitivas que no dependen del tamaño de los datos.

El bucle while tiene una condición que depende de la estructura del heap y el índice i. El número de iteraciones está ligado a la altura del heap, que en el peor de los casos es O(log n) ,donde n es el número de elementos en el heap. Dentro del bucle, se realiza un intercambio y una actualización de i, ambos con complejidad O(1).

Por lo que el orden de magnitud de la función insertarHeap es:

$$O(1) + O(1) + (\log n) = O(\log n)$$

Para el análisis de la función buscarHeap: El ciclo for recorre el arreglo heap de tamaño tamano. En el peor de los casos, debe recorrer todos los elementos, por lo que tiene una complejidad O(n).

Cada comparación dentro del ciclo if tiene una complejidad O(1), y el retorno cuando encuentra el empleado es también O(1).

La función completa tiene una complejidad total O(n), ya que el ciclo domina la ejecución.

Análisis de la función modificarIgresoHeap: La asignación de nuevoIngreso tiene complejidad O(1).

El bucle while ajusta el heap moviendo el elemento hacia arriba, el número de iteraciones del while depende de la altura del heap, por lo que la complejidad es O(log n).

La llamada a maxHeapify reorganiza el heap si es necesario, y su complejidad también es O(log n).

El orden de magnitud total del archivo heap.hpp está dominado por las operaciones de inserción, modificación y reorganización del heap, lo que nos da un orden final de $\boldsymbol{O(log\ n) + O(n) + O(log\ n) = O(log\ n)}$

Análisis del archivo heapsort.hpp

```
1  #ifndef HEAPSORT_HPP
2  #define HEAPSORT_HPP                                   } O(1)
3
4  #include "heap.hpp"
5
6  // Declaraciones de funciones
7  void maxHeapify(Empleado *array, int i, int tamano);
8  void construirMaxHeap(Empleado *array, int tamano);    } O(1)
9  void ordenarHeap(Empleado *array, int tamano);
10
11 // Construir un MaxHeap para ordenar
12 void construirMaxHeap(Empleado *array, int tamano){
13     for (int i = tamano / 2 - 1; i >= 0; i--) ──► O(n/2) ──► O(n)
14         maxHeapify(array, i, tamano); ──► O(log n)
15 }
16
17 // Funcion para ordenar usando el heap
18 void ordenarHeap(Empleado *array, int tamano){
19     construirMaxHeap(array, tamano); ──► O(n log n)
20
21     for (int i = tamano - 1; i > 0; i--) ──► O(n)
22     {
23         swap(array[0], array[i]); // Mueve el mayor al final del array ──► O(1)
24         maxHeapify(array, 0, i);  // Ajusta el heap ──► O(log n)
25     }
26 }
27
28 #endif ──► O(1)
```

Las definiciones de las librerías tienen complejidad O(1), ya que son operaciones primitivas del compilador y no dependen del tamaño de los datos. La declaración de las funciones no ejecuta ningún código, por lo que su complejidad es O(1).

Análisis de la función construirMaxHeap: El ciclo for recorre desde tamano / 2 - 1 hasta 0. Como recorre aproximadamente la mitad de los elementos del arreglo (nodos internos del heap), la complejidad del ciclo es O(n/2), que simplificamos a O(n).

Dentro del ciclo, se llama a la función maxHeapify, que tiene una complejidad de O(log n) (como vimos en el archivo heap.hpp).

En total, la complejidad del bucle se puede expresar como $O(n) \times O(\log n) = O(n \log n)$, por lo que la función tiene un orden de magnitud de $O(n \log n)$.

Análisis de la función ordenarHeap: La llamada a construirMaxHeap tiene una complejidad de $O(n \log n)$, como se explicó antes.

El ciclo for se ejecuta desde i = tamano - 1 hasta i > 0, recorre todos los elementos del arreglo, lo que le da una complejidad O(n).

Dentro del ciclo:

La operación swap tiene una complejidad O(1).

La llamada a maxHeapify tiene una complejidad O(log n), ya que ajusta el heap para mantener sus propiedades después de intercambiar los elementos.

La complejidad del bucle es $O(n) \times O(n) = O(n \log n)$

Por lo que la función construirMaxHeap tiene un orden de magnitud de:

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

La complejidad total del archivo heapsort.hpp está dominada por las funciones de construcción y ordenación del heap, ambas con orden de magnitud $O(n \log n)$.

Análisis del archivo main.cc

```cpp
#include <iostream>
#include <iomanip>
#include "reader.hpp"
#include "heapsort.hpp"
#include "heap.hpp"

using namespace std;

const int Filas = 2000; // Numero de filas (Datos de los empleados) a leer

int main(int argc, char *argv[]){

    Empleado datos[Filas];
    int tamano = 0; // Para manejar el tamaño dinamico del heap

    cargar_datos(datos, Filas);
    tamano = Filas; // El tamaño se iguala al numero de filas cargadas

    int opcion = 0;
    do{
        cout << "\n\t=============== Almacenes Salem ===============\n" << endl;
        cout << "1. Ordenar salarios" << endl;
        cout << "2. Salir" << endl;
        cout << "\nSelecciona una opcion: ";
        cin >> opcion;

        switch (opcion)
        {
        case 1:
            // Ordenar los empleados por salario
            ordenarHeap(datos, tamano);

            // Mostrar los datos ordenados
            cout << left << setw(25) << "Nombre"
                 << setw(30) << "Puesto"
                 << setw(15) << "Ingreso" << endl;
            cout << "-----------------------------------------------------------------" << endl;

            for (int i = tamano - 1; i >= 0; i--)
            {
                cout << left << setw(25) << datos[i].nombre
                     << setw(30) << datos[i].puesto
                     << "$ " << setw(14) << fixed << setprecision(2) << datos[i].ingreso << endl;
            }
            break;

        case 2:
            cout << "Saliendo del programa..." << endl;
            break;

        default:
            cout << "Opcion invalida....." << endl;
            break;
        }

    } while (opcion != 2);

    return 0;
}
```

Las inclusiones de librerías y archivos tienen complejidad O(1), ya que son instrucciones del compilador que no dependen del tamaño de los datos. La declaración de una constante global tiene una complejidad O(1).

La declaración de variables tiene una complejidad O(1), ya que no implica ningún cálculo o bucle.

La función cargar_datos tiene una complejidad de O(n), ya que recorre las 2000 filas para cargar los datos. Igualar el tamaño al número de filas es una operación simple con complejidad O(1).

Mostrar el menú de opciones y leer la opción seleccionada tiene una complejidad O(1)), ya que son operaciones de entrada y salida.

Analizando el case 1:

La función ordenarHeap tiene una complejidad de O(n log n), ya que es el algoritmo de ordenación basado en heapsort.

El ciclo for recorre los datos desde el último empleado hasta el primero, lo que tiene una complejidad O(n), con operaciones constantes en cada iteración O(1).

El orden de magnitud total del caso 1 es $O(n \log n) + O(n) = O(n \log n)$

Analizando el case 2:

Este caso tiene una complejidad O(1), ya que solo muestra un mensaje y sale del programa.

Analizando el case default:

Este caso tiene una complejidad O(1), ya que muestra un mensaje de error si la opción ingresada no es válida.

El ciclo do-while se ejecutará al menos una vez, y puede ejecutarse varias veces si el usuario no selecciona la opción 2. Su complejidad está limitada por el número de iteraciones que realice, pero en general, el camino más costoso es el de ordenar los salarios, con una complejidad $O(n \log n)$.

Análisis del programa total

Una vez analizado el código de cada archivo del programa para definir el orden de magnitud total del programa en general, se considerara el análisis de cada archivo utilizado y las funciones más importantes que contribuyen al tiempo de ejecución.

- **Orden de magnitud del archivo reader.hpp:** El orden de magnitud es de O(n).
- **Orden de magnitud del archivo heap.hpp:** El orden de magnitud es de O(log n).
    - **Función insertarHeap:** El orden de magnitud es de O(log n).
    - **Función maxHeapfy:** Orden de magnitud es de O(log n).
- **Orden de magnitud del archivo heapsort.hpp:** El orden de magnitud es de O(n log n).
    - **Funcion ordenarHeap:** Orden de magnitud es O(n log n).
- **Orden de magnitud del arcgivo main.cc:** El orden de magnitud es de O(n log n).

Ahora teniendo en cuenta el orden de magnitud de cada función y cada archivo en genera, sabemos que el programa presenta diversas operaciones importantes, cada una con diferentes órdenes de complejidad. Entre estas operaciones destacan la carga de datos desde el archivo, la inserción y modificación de empleados en un heap, y la ordenación de salarios utilizando el algoritmo heapsort.

La función cargar_datos tiene una complejidad de O(n), ya que recorre los datos de cada empleado para cargarlos desde el archivo.

Las operaciones de inserción y modificación de empleados en el heap, implementadas en el archivo heap.hpp, tienen una complejidad de O(log n). Estas operaciones ajustan el heap conforme se insertan o modifican elementos, pero son más rápidas que las operaciones lineales y tampoco representan el mayor costo en términos de tiempo de ejecución.

La función ordenarHeap, ubicada en el archivo heapsort.hpp, que implementa el algoritmo de heapsort. Es responsable de ordenar los salarios de los empleados y tiene una complejidad de $O(n \log n)$ .Dado que el proceso de ordenación se realiza sobre todo el conjunto de empleados y se ejecuta una vez, este es el paso que domina el comportamiento del programa en términos de eficiencia temporal.

Por lo tanto, aunque el programa tiene varias operaciones con diferentes órdenes de magnitud, el paso dominante es la ordenación de salarios mediante heapsort, con una

complejidad de $O(n \log n)$ .Esto se debe a que heapsort implica construir un heap máximo y luego ajustar el heap mientras se ordenan los elementos, lo que involucra repetidas llamadas a la función maxHeapify, cuya complejidad es $O(n \log n)$.

$Por\ lo\ que\ el\ orden\ de\ magnitud\ total\ del\ programa\ es\ de\ O(n \log n)$

Conclusiones

Al analizar nuestro programa, hemos reflexionado sobre la importancia de diseñar soluciones eficientes, especialmente cuando se trabaja con grandes volúmenes de datos, como es el caso de los empleados en una empresa. La implementación del algoritmo de heapsort, con su orden de magnitud $O(n \log n)$, nos permitió ordenar los salarios de manera óptima, convirtiendo el proceso de ordenación en la operación dominante del sistema. A pesar de que otras funciones como la carga de datos y la inserción en el heap tienen complejidades menores, el comportamiento global del programa está determinado por la eficiencia de heapsort. Esta elección algorítmica no solo resultó adecuada en términos de tiempo de ejecución, sino que también fue crucial para respetar las restricciones de memoria impuestas por el proyecto.

Esta experiencia nos ha hecho reflexionar sobre la union entre las diferentes partes de un programa. Cada componente, desde las operaciones simples hasta los algoritmos más complejos, juega un papel en el rendimiento global. Aunque algunas partes del código tienen una complejidad lineal, el uso de un algoritmo más avanzado para la ordenación garantizó un equilibrio entre velocidad y recursos. Esto resalta la importancia de seleccionar algoritmos adecuados para resolver problemas de manera eficiente, logrando así un programa que no solo cumple con los objetivos, sino que también optimiza los recursos disponibles en la ejecución.