



**UNIVERSIDAD CENTROAMERICANA
JOSÉ SIMEÓN CAÑAS
DEPARTAMENTO DE ELECTRÓNICA E
INFORMÁTICA**

ANÁLISIS DE ALGORITMOS

Ciclo 02/2024

Investigación:
TALLER 2

Estudiantes:

**MEJIA HERNADEZ, SALVADOR MARCELO, 00072020
MERINO CHAVEZ, CHRISTIAN ALEXIS, 00002521
ROGEL CAMPOS, ALEXANDER RAFAEL, 00100922**

Catedrático:
Emmanuel Araujo, ING.

Fecha de entrega: Sábado 12 de octubre de 2024.

Infografía-Etapa II

MONTÍCULOS

HEAP

¿Qué es un Montículo?

Un montículo (o heap en inglés) es una estructura de datos del tipo árbol binario completo, en la que los nodos siguen una relación de orden específica. Es una implementación eficiente para obtener el mínimo o máximo en un conjunto de datos, dependiendo del tipo de montículo.

Tipos de montículos

- Montículo Máximo.
- Montículo Mínimo.
- Montículo de Fibonacci.
- Montículo Binomial.
- Montículo Ternario.

Propiedades:

Montículo Mínimo: el valor de cada nodo es menor o igual al de sus hijos. Es decir, el valor mínimo siempre está en la raíz del árbol.

Montículo Máximo: el valor de cada nodo es mayor o igual al de sus hijos. El valor máximo siempre está en la raíz

Operaciones Comunes:

- Insertar
- Eliminar el elemento mínimo o máximo
- Peek

Aplicaciones:

- Colas de Prioridad: base para implementar colas de prioridad
- Algoritmo de Dijkstra: Utiliza montículos para encontrar el camino más corto desde un nodo en un gráfico ponderado.
- Heap Sort: Un algoritmo de ordenamiento que utiliza la propiedad de los montículos para ordenar un arreglo.

Código-Etapa III

Enlace GitHub:

https://github.com/AlexCampos03/AA022024_Taller2 Grupo10.git

Análisis Formal-Etapa IV

Void intercambiar (int & a, int & b); — O(1)

Análisis Formal

- void intercambiar(int & a, int & b) { // Función intercambiar
 int temp = a; // O(1): Asigna el valor de 'a' a 'temp'
 a = b; // O(1): Asigna el valor de 'b' a 'a'.
 b = temp; // O(1): Asigna el valor de 'temp' a 'b'
}

Complejidad: La complejidad de esta función es $O(1)$ (constante), ya que se realizan tres operaciones de asignación sin importar el tamaño de los datos. Esto significa que el tiempo de ejecución no aumenta con el tamaño de la entrada.

- int leeSalarios (const string& Archivo, int salarios[], int maxSalario) {
 ifstream archivo(Archivo); // O(1): Abrir el archivo es una operación constante.
 if (!Archivo.is_open()) {
 cerr << "Error al abrir el archivo: " << Archivo << endl;
 return 0; // O(1): Salida inmediata si hay un error.
 }
 int i=0; // O(1): Inicialización de la variable 'i'.
 while (archivo >> salarios[i] && i < maxSalario) {
 i++; // O(1): Incremento del índice 'i'.
 }
 archivo.close(); // O(1): Cerrar el archivo es una operación constante
 return i; // O(1): Devuelve el número de salarios leídos.
}

Complejidad: La apertura y cierre del archivo, la verificación de su apertura, la inicialización de variables y el manejo de errores son operaciones constantes ($O(1)$).

La operación crítica es el bucle while que lee los salarios. Este bucle se ejecuta hasta un máximo de $\max(\text{Salario})$ veces, es decir, su complejidad es $O(n)$, donde n es el número de salarios leídos.

La complejidad total de leeSalarios es $O(n)$, dominada por la cantidad de lecturas realizadas.

i Intercambiar, leeSalarios, While Son partes no recursivas!

Análisis Formal

```
Void heapify(int salarios[], int n, int i) {
    int mayor = i; // O(1)
    int izq = 2 * i + 1; // O(1) } operaciones primitivas
    int der = 2 * i + 2; // O(1)

    if (izq < n && salarios[izq] > salarios[mayor])
        mayor = izq; // O(1)

    if (der < n && salarios[der] > salarios[mayor])
        mayor = der; // O(1)

    if (mayor != i) {
        intercambiar(salarios[i], salarios[mayor]);
        heapify(salarios, n, mayor); // O(n/2) → recurrencia
    }
}

void intercambiar(int &a, int &b) {
    int temp = a; // O(1)
    a = b; // O(1) } operaciones primitivas
    b = temp; // O(1)
```

Analisis Formal

```
void ordenarSalarios( int salarios[], int n ) { // función ordenar
    for ( int i = n / 2 - 1; i >= 0; i-- ) { O(n/2)
        heapify( salarios, n, i ); O(log n)
    } // O(n/2) + O(1)
```

```
for ( int i = n - 1; i > 0; i-- ) { O(n-1)
    intercambiar( salarios[0], salarios[i] ); es O(1)
    heapify( salarios, i, 0 ); O(n/2)
} // O(n/2) → el peor de los casos es O(n log n)
```

```
- void insertarHeap( int salarios[], int &n, int valor ) {
    n++;
    int i = n - 1; // operación primitiva O(1)
    salarios[i] = valor;
```

```
while ( i != 0 && salarios[(i-1)/2] < salarios[i] ) { O(n/2)
    intercambiar( salarios[i], salarios[(i-1)/2] );
    i = (i-1)/2; // O(1)
}
```

|| el peor de los casos para InsertarHeap es O(n/2)

utilizando el teorema maestro para $T(n) = T(\frac{n}{2}) + O(1)$
donde $a=1$ $b=2$ y $d=0$ podemos decir que aplican
el caso "dos" $T(n) = O(n^{\frac{1}{2}} \log n)$ = esto nos dejar la ecuación
para reemplazar, $T(n) = O(\log n)$, resolviendo por análisis simple
la recurrencia final nos estarian quedando $(n-1)(\log n)$
 $n \log n \approx \log n$

$O(\log n) - O(\log n) // \text{Const.}$ | Seleccionamos solo a
 $O(n \log n)$

Para Ordenar Salario $O(n \log n)$,

Para Insertar Heap $O(n \log n)$ |

```
int eliminarMax (int salarios[], int & n) {
    if (n <= 0) return -1; // O(1)
    if (n == 1) { // O(1)
        n--;
        return salarios[0]; // O(1)
    }
}
```

```
int raiz = salarios[0]; // O(1)
salarios[0] = salarios[n-1]; // O(1)
n--;
heapify(salarios, n, 0); // O(log n)
return raiz; // O(1)
} → No es recursiva
```

```
bool buscarMax (int salarios[], int n, int valor) {
    for (int i = 0; i < n; ++i) // O(n)
        if (salarios[i] == valor) // O(1)
            return true; // O(1)
    return false; // O(1)
} → No es recursiva
```

Función Main

```
int main() {
    const int MAX_SALARIOS = 1000; — O(1)
    int salarios[MAX_SALARIOS]; — O(1)
    int numSalarios = leerSalarios("C:\\Users\\Usuario\\Desktop\\Salarios.txt",
        Salarios, MAX_SALARIOS); — O(n)
    if (numSalarios == 0) { — O(1)
        cerr << "No se pudieron leer los datos del archivo." << endl; O(1)
        return 1; — O(1)
    }
    int opcion; — O(1)
    do {
        cout << "\n--- MENU ---" << endl; — O(1)
        cout << "1. Insertar un nuevo salario" << endl; — O(1)
        cout << "2. Buscar un salario" << endl; — O(1)
        cout << "3. Eliminar el salario máximo" << endl; — O(1)
        cout << "4. Ordenar los salarios" << endl; — O(1)
        cout << "5. Mostrar los primeros 10 salarios" << endl; — O(1)
        cout << "6. Salir" << endl; — O(1)
        cout << "Seleccione una opción: "; — O(1)
        cin >> opcion; — O(1)
```

```
switch(Opcion){
```

```
    case 1: {
```

```
        int nuevoSalario; — O(1)
```

```
        cout << "Ingrese el nuevo salario a insertar: "; — O(1)
```

```
        cin >> nuevoSalario; — O(1)
```

```
        insertarHeap(Salarios, numSalarios, nuevoSalario); — O(log n)
```

```
        cout << "Salario $" << nuevoSalario << " insertado correctamente." << endl; — O(1)
```

```
        break; — O(1)
```

```
}
```

```
    case 2: {
```

```
        int salarioBuscado; — O(1)
```

```
        cout << "Ingrese el salario a buscar: "; — O(1)
```

```
        cin >> SalarioBuscado — O(1)
```

```
        bool encontrado = buscarHeap(Salarios, numSalarios, salarioBuscado); — O(n)
```

```
        if (encontrado) { — O(1)
```

```
            cout << "El salario $" << SalarioBuscado << " se encontro en el heap." << endl; — O(1)
```

```
} else {
```

```
    cout << "El salario $" << SalarioBuscado << " no se encontro en el heap." << endl; — O(1)
```

```
}
```

```
        break; — O(1)
```

```
}
```

```
    case 3: {
```

```
        int salarioEliminado = eliminarMax(Salarios, numSalarios); — O(log n)
```

```
        if (salarioEliminado != -1) { — O(1)
```

```
            cout << "El salario maximo eliminado es: $" << salarioEliminado << endl; — O(1)
```

```
} else {
```

```
    cout << "El heap esta vacio, no se puede eliminar." << endl; — O(1)
```

```
}
```

```
        break; — O(1)
```

```
}
```

```

Case 4: {
    Ordenar Salarios(Salarios, numSalarios); — O(n log n)
    cout << "Salarios ordenados correctamente." << endl; O(1)
    break; — O(1)
}

Case 5: {
    cout << "Primeros 10 salarios (disponibles):" << endl; — O(1)
    for (int i = 0; i < 10 && i < numSalarios; ++i) {
        cout << i + 1 << ". " << salarios[i] << endl; — O(1)*10 = O(10)
    }
    break; — O(1)
}

Case 6: {
    cout << "Saliendo del programa..." << endl; — O(1)
    break; — O(1)
}
default:
    cout << "Opcion no valida." << endl; — O(1)
}

} while (opcion != 0); — O(n) = el peor de los casos
return 0; — O(1)
}

```

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

Ensayo-Etapa IV

El proceso del análisis de las funciones que implementamos permitió observar de manera clara la eficiencia de los algoritmos aplicados. A través de cada línea de código, identificamos que la función heapify tiene un comportamiento logarítmico gracias a su estructura recursiva, asegurando un ajuste eficiente dentro de la estructura del heap. De igual forma se logró establecer cómo las operaciones más simples, como las asignaciones y comparaciones, contribuyen con un costo constante $O(1)$, mientras que la complejidad global de ordenar los salarios mediante heapsort alcanza $O(n\log n)$.

Durante el análisis del flujo del programa, se identificó que la estructura mediante un menú permite la interacción del usuario de manera dinámica. Este tipo de análisis es crucial para poder saber el comportamiento del software a medida que los datos aumentan. En el peor de los casos, las operaciones dentro del heap aseguran un rendimiento logarítmico, evitando la degradación significativa del rendimiento.

En resumen, el taller nos mostró la importancia de aplicar el Teorema Maestro en la resolución de recurrencias, lo cual nos permite hacer el análisis de algoritmos recursivos. En general, el análisis es un paso clave en la comprensión de cómo los algoritmos afectan el rendimiento global del programa.