UNIVERSIDAD CENTROAMERICANA
JOSÉ SIMEÓN CAÑAS

Análisis de algoritmos
Sección 01

Ciclo 02/2024

Actividad:
Taller 2

Integrantes:

José Juventino Castillo Hernández 00048322
Cristofer Ricardo Díaz Alfaro 00071222
Oscar Ernesto Menjívar Ayala 00068422

Catedrático:
Enmanuel Araujo

Antiguo Cuscatlán, 12 de octubre del 2024

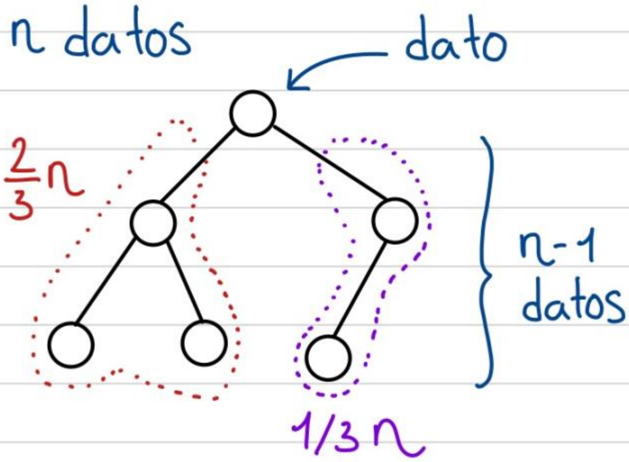
Heap.cc

```cpp
#include "heap.h"
#include "employee.h"

Employee heap[MAX_SIZE_HEAP];
int size = 0;

int Parent(int i) { return (i - 1) / 2; }                      O(1)
int LeftChild(int i) { return 2 * i + 1; }
int RightChild(int i) { return 2 * i + 2; }
bool IsHeapFull() { return size == MAX_SIZE_HEAP; }
bool IsHeapEmpty() { return size == 0; }

void HeapifyDown(int i, int _size) {                           T(n)= O(log_2 n)
    int current_max = SearchLargestElement(i, _size);          O(1)

    if (current_max != i) {                                    max(0, log_2 n)
        Employee temp = heap[i];                               O(1)
        heap[i] = heap[current_max];
        heap[current_max] = temp;

        HeapifyDown(current_max, _size);                       O(log_2 n)*
    }
}

void SiftDown(Employee data, int i) {                          T(n)= O(log_2 n)
    if (i == 0) {
        heap[0] = data;                                        max (0,1) → O(1)
        return;
    }

    int parent = Parent(i);                                    O(1)

    if (heap[parent].salario < data.salario) {                 max (0, log_2 n)
        heap[i] = heap[parent];                                O(1)
        SiftDown(data, parent);                                O(log_2 n)*
    } else {
        heap[i] = data;                                        O(1)
    }
}
int SearchLargestElement(int i, int _size) {
    int left = LeftChild(i);
    int right = RightChild(i);
```

* Desarrollo en la siguiente página

Recurrencia Heapify Down y Sift Down

$T(n) = T(2n/3) + O(1)$



La cantidad de datos en cada llamada recursiva es de $2n/3$ ya que tenemos un árbol desbalanceado, es decir un árbol que no tiene sus hojas llenas.

En el dato superior tenemos 1 nodo mientras que en las ramas tenemos n-1 nodos por lo tanto en un lado donde hay más tenemos $\approx 2/3$ de los datos y en el otro $\approx 1/3$

Resolviendo con teorema maestro.

$a = 1, \quad b = 3/2 \quad d = 0$

$$\left.\begin{array}{r} \log_{3/2}(1) = 0 \\ 0 = d \end{array}\right\} \text{caso 2}$$

$T(n) = n^0 \cdot \log_{3/2} n$

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

$\log_{3/2}$ ^ $\log_2$ tienen la misma complejidad asintótica ya que

$$\log_{3/2} n = \frac{\log n}{\log \frac{3}{2}}$$

El factor $1/\log$ es solo una constante y en notación $O(n)$ las constantes se omiten

Por lo tanto el resotado final es $O(\log n)$

```
int SearchLargestElement(int i, int _size) {      ⟶ T(n) = O(1)
    int left = LeftChild(i);
    int right = RightChild(i);      } O(1)
    int current_max = i;

    if (left < _size && heap[left].salario > heap[current_max].salario) {      } max(0,1) → O(1)
        current_max = left;
    }

    if (right < _size && heap[right].salario > heap[current_max].salario) {      } max(0,1) → O(1)
        current_max = right;
    }

    return current_max;   O(1)
}

bool RemovebyId(int id) {      ⟶ T(n) = O(log2 n)
    if (IsHeapEmpty()) {
        std::cerr << "Heap is empty\n";      } max (0,1) → O(1)
        return false;
    }

    heap[id] = heap[size - 1];      } O(1)
    size--;

    HeapifyDown(id, size);      } O(log2 n)

    return true;   O(1)
}
```

```
int SearchSmallestElement(int i, int _size) {      ⟶ T(n) = O(1)
    int left = LeftChild(i);
    int right = RightChild(i);      } O(1)
    int current_min = i;

    if (left < _size && heap[left].salario < heap[current_min].salario) {      } max(0,1)
        current_min = left;
    }

    if (right < _size && heap[right].salario < heap[current_min].salario) {      } max (0,1)
        current_min = right;
    }

    return current_min;   O(1)
}
```

```cpp
int SearchSmallestElement(int i, int _size) {   ⟶ T(n)= O(1)
    int left = LeftChild(i);
    int right = RightChild(i);                   } O(1)
    int current_min = i;

    if (left < _size && heap[left].salario < heap[current_min].salario) {   } max(0,1)
        current_min = left;
    }

    if (right < _size && heap[right].salario < heap[current_min].salario) {   } max(0,1)
        current_min = right;
    }

    return current_min;   O(1)
}

void InsertData(Employee data) {   ⟶ T(n)= O(log₂n)
    if (IsHeapFull()) {
        std::cerr << "Heap is full\n";   } max(0,1) ⟶ O(1)
        return;
    }

    heap[size] = data;   O(1)
    SiftDown(data, size);   ⟶ O(log₂n)
    size++;   O(1)
}

Employee RemoveMin() {   ⟶ T(n)= O(log₂n)
    if (IsHeapEmpty()) {
        std::cerr << "Heap is empty\n";

        Employee emptyEmployee;                  } max(0,1) ⟶ O(1)
        emptyEmployee.salario = -1;
        return emptyEmployee;
    }

    Employee EmployeeMin = heap[0];   O(1)

    RemovebyId(0);   O(log₂n)

    return EmployeeMin;   O(1)
}
```

```
void DisplayHeap(){ ──────→ T(n)= O(n)
    std::cout << "Contenido del heap:" << std::endl; O(1)
    for (int i = 0; i < size; i++)
        std::cout << heap[i].salario << "\n"; } O(n)
}

void OrderHeap() { ──────→ T(n)= O(n log2 n)
    Employee sorted_employees[MAX_SIZE_HEAP]; } O(1)
    int original_size = size;

    for (int i = 0; i < original_size; i++) { O(n)
        sorted_employees[i] = RemoveMin(); O(log n) · O(n)
        std::cout << sorted_employees[i].nombre << " - " << O(1)
    }
}

int SearchHeapByValue(float value) { ──────→ T(n) = O(n)
    for (int i = 0; i < size; ++i) {
        if (heap[i].salario == value) { } O(n)
            return i;
        }
    }
    return -1; O(1)
}
```

Recuento

- Heapify Down $\longrightarrow O(\log_2 n)$
- Sift Down $\longrightarrow O(\log_2 n)$
- Insert Data $\longrightarrow O(\log_2 n)$
- Search Largest Element $\longrightarrow O(1)$
- Search Smallest Element $\longrightarrow O(1)$
- Remove Min $\longrightarrow O(\log_2 n)$
- Display Heap $\longrightarrow O(n)$
- Order Heap $\longrightarrow O(n \log n)$
- Search Heap By Value $\longrightarrow O(n)$

Main.cc

```cpp
#include <fstream>
#include "heap.h"
#include "employee.h"

void LoadDataFromFile(const char* filename) {      → T(n)= O(n log_2 n)
    std::ifstream file(filename);                  } max(0,1) → O(1)

    if (!file) {
        std::cout << "Sorry: Could not open file " << filename << ".\n";
        return;
    }

    Employee empleado;                             } O(1)
    int count = 0;
    while (file >> empleado.nombre >> empleado.apellido >> empleado.salario >> empleado.cargo) { → O(n)
        InsertData(empleado);  → O(n) · O(log n) → O(n log n)
        count++;  → O(1) · O(n)

        if (IsHeapFull()) {
            std::cout << "Error: Heap is full after " << count << " empleados.\n";   } O(1) · O(n)
            break;
        }
    }

    std::cout << "Loaded " << count << " employees from file " << filename << ".\n";   } O(1)
    file.close();
}
int main(void) {      → T(n)= O(n log_2 n)
    LoadDataFromFile("usuarios.txt");  → O(n log n)

    int option = 0;
    while (option != 2) {
        std::cout << "\nMENU\n";                       } O(1)
        std::cout << "1. Ordenar salarios\n";
        std::cout << "2. Salir\n";
        std::cout << "Opcion: ";
        std::cin >> option;

        switch (option) {
        case 1:
            OrderHeap();  → O(n log n)
            break;
        case 2:
            std::cout << "Saliendo ... \n";            } O(1)
            break;
        default:
            std::cout << "Opcion invalida\n";
        }
    }

    return 0;   } O(1
}
```

Conclusión:

Como equipo, la experiencia de desarrollar un algoritmo basado en montículos (heap) nos permitió aplicar los conceptos teóricos de estructuras de datos a una problemática que simula un caso real. La tarea de ordenar los salarios de los empleados en orden descendente, utilizando un algoritmo eficiente como el Heap Sort, nos demostró el rendimiento superior que ofrecen las estructuras de datos tipo heap para el ordenamiento y búsqueda en grandes conjuntos de datos, en comparación con otros algoritmos y estructuras de datos tradicionales.

Durante el desarrollo, enfrentamos ciertos desafíos, ya que ninguno de los integrantes del equipo tenía un conocimiento previo profundo sobre los montículos. Sin embargo, al distribuir las tareas de manera eficiente y mantener una comunicación constante, logramos aprender en conjunto y superar los obstáculos iniciales. Pudimos, además, analizar en detalle tanto los aspectos recursivos como no recursivos del código, utilizando el Teorema Maestro para determinar la complejidad del algoritmo, lo que nos permitió cuantificar su eficiencia y confirmar así que Heap Sort es adecuado para ordenar grandes conjuntos de datos en tiempo $O(n\log_2(n))$.

En definitiva, este taller nos enseñó la importancia de elegir estructuras de datos eficientes, como los montículos, para resolver problemas complejos, así como la relevancia del liderazgo y el trabajo en equipo. Cada miembro del equipo aportó desde sus fortalezas para alcanzar el objetivo final, demostrando que la colaboración y el análisis crítico son esenciales para el desarrollo de software de calidad.

Referencias:

http://users.diag.uniroma1.it/~aris/contents/teaching/data-mining-ds-2018/resources/clrs-heapsort.pdf