Universidad Centroamericana José Simeón Cañas

Departamento de Electrónica y Informática

Análisis de Algoritmos

Taller 2

Catedraticos

Ing. Mario López

Msc. Enmanuel Amaya

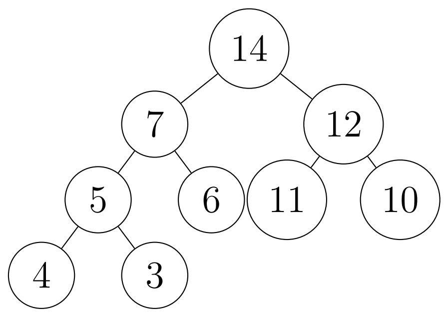
Integrantes

Gerardo Daniel Olivares Caceres 00214917
Julio Alberto Rodriguez Valencia 00163922
Pablo Enrique Vides Zavala 00080323

12 de octubre del 2024

Introducción

Heaps



La estructura de datos *heap o montículo* es un tipo de árbol binario que satisface la propiedad de *heap*, donde cada nodo es mayor (o menor, en el caso de un *min-heap*) que sus nodos hijos. Esta propiedad permite que el elemento máximo (o mínimo) esté siempre en la raíz, lo que facilita la implementación de algoritmos de prioridad.

Propiedades del Heap

Las principales propiedades del heap son las siguientes:

- Un heap es un árbol binario completo, lo que significa que todos los niveles del árbol están completamente llenos, excepto posiblemente el último, que se llena de izquierda a derecha.

- En un *max-heap*, el valor de cada nodo es mayor o igual que el de sus hijos. En un *min-heap*, el valor de cada nodo es menor o igual que el de sus hijos.

Los montículos se suelen representar como arreglos, donde la posición de un nodo padre y sus hijos sigue una relación específica. Por ejemplo, para un nodo en la posición i, sus hijos se encuentran en las posiciones

$2i + 1$ y $2i + 2$. Esta representación permite realizar operaciones de inserción y eliminación en tiempo logarítmico, lo que los hace eficientes para aplicaciones como *colas de prioridad* o algunos algoritmos de gráfos.

Análisis del Programa

1. Función de heapMaximo

Análisis del Algoritmo

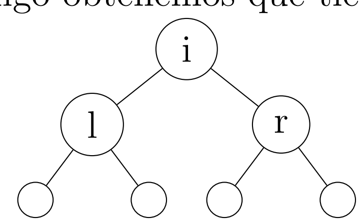
```
void heapMaximo(Empleado arrLocal[], int l, int i) { $c_1$
    int cabezera = i; $c_2$
    int izquierda = 2 * i + 1; $c_3$
    int derecha = 2 * i + 2; $c_4$

    if (izquierda < l && arrLocal[izquierda].salario > arrLocal[cabezera].salario
    ) { $c_5$
        cabezera = izquierda; $c_6$
    }
    if (derecha < l && arrLocal[derecha].salario > arrLocal[cabezera].salario) {
    $c_7$
        cabezera = derecha;$c_8$
    }
    if (cabezera != i) { $c_9$
        swap(arrLocal[i], arrLocal[cabezera]);  $c_10$
        heapMaximo(arrLocal, l, cabezera);$T(\frac{2n}{3})$
    }
}
```

Dado que la *driving function* es $\Theta(1)$, se puede plantear la siguiente recurrencia

$$T(n) = T(\frac{2n}{3}) + \Theta(1)$$

Para un nodo i de un árbol de n nodos, su hijo izquierdo o derecho es un subárbol a lo mucho $\frac{2n}{3}$ nodos. Además al analizar la sección de comparaciones del código obtenemos que tienen un tiempo de $\Theta(1)$



Dado que se cumplen las condiciones, se puede resolver la recurrencia por el segundo caso del Teoerema Maestro, que dicta

Si existe una $k \geq 0$ tal que $f(n) = \Theta(n^{\log_b a} \lg n^k)$ entonces $T(n) = \Theta(n^{\log_b a} \lg n^{k+1})$

tenemos que

$$a = 1, b = \frac{3}{2}, k = 0$$

. Tenemos que

$$log_{\frac{3}{2}} 1 = 0$$

Por lo tanto,

$$T(n) = n^0 \log_2 n$$

Dado que $d = 0$

$$T(n) = O(\log_2 n)$$

Tambien si sabemos que el heap siempre tendra una altura $h = lg(n)$, podemos tener lo equivalente a $O(h)$

2. Función de construirMonticuloMaximo

Análisis del Algoritmo

```
void construirMonticuloMax(Empleado arrLocal[], int l){
    for (int i = (l / 2) - 1; i >= 0; i--) { //Construyendo heap mximo.
        heapMaximo(arrLocal, l, i);
    }
}
```

Para analizar este algoritmo, vemos la máxima cantidad de nodos a una altura h, por lo cual se puede deducir la siguiente expresión,

$$\lceil \frac{n}{2^{h+1}} \rceil$$

Para obtener la complejidad de tiempo, sumamos el costo por nodo ch la cantidad de nodos por nivel h, por lo cual tenemos que

$$\sum_{h=0}^{\lceil \lg(n) \rceil} (\frac{n}{2^{h+1}})ch$$
$$\leq \sum_{h=0}^{\lceil \lg(n) \rceil} (\frac{n}{2^{h+1}})ch$$
$$= cn \sum_{h=0}^{\lceil \lg(n) \rceil} (\frac{h}{2^{h}})$$
$$\leq cn \sum_{h=0}^{\infty} (\frac{h}{2^{h}})$$

Usando la derivada de la serie geométrica tenemos que

$$\sum_{n=0}^{\infty} nr^n = \frac{r}{(1-r)^2}$$

Por lo tanto,

$$\leq cn(\frac{\frac{1}{2}}{1-\frac{1}{2}})$$
$$\leq cn = O(n)$$

3. Función de ordenarMonticulo

Análisis del Algoritmo

```
void ordenarMonticulo(Empleado arrLocal[], int l) { c1
    construirMonticuloMax(arrLocal, l); O(n)
    for (int i = l - 1; i >= 0; i--) {c3
        swap(arrLocal[0], arrLocal[i]);c4
        heapMaximo(arrLocal, i, 0);O(lg(n))
    }
}
```

Esta implementación de heapsort, primero construye un monticulo máximo , itera desde el nodo mas profundo hasta la raíz , cambiando el nodo actual por la raíz lo cual es operación constante , y llamando a *heapMaximo(i)*, en ese mismo nodo, asi hasta llegar al elemento justo antes de la raíz.

El análisis anterior se puede modelar formalmente de la siguiente forma

$$T(n) = c_1 + O(n) + \sum_{h=0}^{n} c_3 + \sum_{h=0}^{n-1} c_4 + \sum_{h=0}^{n-1} O(lgn)$$

$$T(n) = c_1 + O(n) + c_3(n+1) + c_4 n + nO(\lg n)$$

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

4. Función de insertar

Análisis del Algoritmo

```
void insertar(Empleado empleado) { $c_1$
    if (heapSize == maxSize) { $c_2$
        cout << "El heap de empleados est lleno\n"; $c_3$
        return; $c_4$
    }
    empleados[heapSize] = empleado; $c_5$
    heapSize++; $c_6$
    construirMonticuloMax(empleados, heapSize); $O(n)$
}
```

La complejidad de tiempo es la siguiente,

$$T(n) = c_1 + c_2 + c_5 + c_6 + O(n) = O(n)$$

5. Función de buscar

Análisis del Algoritmo

```
int buscar(Empleado empleado) { $c_1$
    for (int i = 0; i < heapSize; i++) { $c_2, n+1$
        if (empleados[i].nombre == empleado.nombre && empleados[i].apellido ==
    empleado.apellido ) { $c_3, n$
            return i; $c_4, n$
        }
    }
    return -1; $c_5$
}
```

La complejidad de tiempo, en el peor de los casos es el siguiente,

$$T(n) = c_1 + c_2(n+1) + c_3 + c_5 = O(n)$$

6. Función de eliminar

Análisis del Algoritmo

```
void eliminar(Empleado empleado) { $c_1$
    int indice = buscar(empleado); $O(n)$
    if (indice == -1) {  $c_3$
        cout << "No se encontr el elemento.\n"; $c_4$
        return; $c_6$
    }

    empleados[indice] = empleados[heapSize - 1]; $c_7$
    heapSize--; $c_8$
    construirMonticuloMax(empleados, heapSize); $O(n)$
}
```

Se trata de operaciones primitivas, y llamado a otras funciones por lo tanto , tenemos que

$$T(n) = c_1 + O(n) + c_3 + c_7 + c_8 + O(n) = O(n)$$

7. Función de leerArchivo

Análisis del Algoritmo

```
void leerArchivo() { $c_1$
    ifstream file("empleados.txt");  $c_2$

    if (!file.is_open()) { $c_3$
```

```cpp
        cout << "Error: No se pudo abrir el archivo." << endl; c4
        return; c5
    }

    string line; c6
    while (getline(file, line)) { c7
        size_t pos1 = line.find(','); c8
        size_t pos2 = line.find(',', pos1 + 1); c9

        string nombre = line.substr(0, pos1); c10
        string apellido = line.substr(pos1 + 1, pos2 - pos1 - 1); c11
        float salario = stof(line.substr(pos2 + 1));  c12

        if (heapSize < 1000) { c13
            empleados[heapSize].nombre = nombre; c14
            empleados[heapSize].apellido = apellido; c15
            empleados[heapSize].salario = salario; c16
            heapSize++; c17
        } else { c18
            cout << "Se alcanz el nmero mximo de empleados." << endl; c19
            break; c20
        }
    }
    file.close(); c21
}
```

La complejidad de tiempo variara, de el numero de lineas, n, por lo tanto tenemos,

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 c_6 + c_7(n+1) + n(c_8 + c_9 + c_{10}+$$

$$c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{17} + c_{18} + +c_{21}$$

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

Sin embargo, como el número de lineas en el archivo es constante en este caso, $n = 1000$,

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

8. Función de eliminar

Análisis del Algoritmo

```
void mostrarDescendente() { c1
    ordenarMonticulo(empleados, heapSize); O(n lg(n))

    for (int i = heapSize - 1; i >= 0; i--) { c2, n + 1
        cout << empleados[i].nombre << " " << empleados[i].apellido << " Salario: " << empleados[i].salario << "\n"; c3, n
    }
}
```

La complejidad de este algoritmo es la siguiente,

$$T(n) = c_1 + O(n \lg(n)) + c_2(n+1) + c_3 n = O(n \lg n)$$

9. Función Principal

Análisis del Algoritmo

```
int main() { c1
    leerArchivo(); O(1)

    int opcion; c3

    do { c4
        cout << "1. Mostrar salarios en orden descendente" << "\n"; c5
        cout << "2. Salir" << "\n"; c6
        cout << "Seleccione una opcin: " c7;
        cin >> opcion; c8

        switch (opcion) { c9
            case 1:
                mostrarDescendente();O(n lg(n))
                cout << "\n"; c11
                break; c12
            case 2:
                cout << "Finalizando programa" << "\n";c13
                break; c14
            default:
                cout << "Opcin incorrecta" << "\n"; c15
                break; c16
        }
    } while (opcion != 4);
```

```

    return 0;
}
```

La complejidad de tiempo del programa completo es $O(n \lg(n))$

Reflexión

En base al trabajo anterior, podemos inferir que Heapsort ofrece rendimiento y eficiencia , lo que lo convierte en una opción buena para resolver el problema de ordenamiento. Una de sus principales ventajas es que requiere una memoria mínima, ya que ordena en el lugar sin necesidad de espacio adicional, a diferencia de Merge Sort o Quick Sort recursivo, que requieren memoria adicional para sus operaciones. Esto hace que Heapsort sea adecuado para entornos con recursos limitados.

Sin embargo, tiene sus desventajas. Heapsort se considera inestable, lo que significa que puede cambiar el orden relativo de elementos iguales, lo que puede no causar complicaciones en algunas aplicaciones. Además, es posible que no funcione de igual manera con estructuras de datos más complejas en comparación con otros algoritmos como Quick Sort, que normalmente tiene un mejor rendimiento en el caso promedio.

En términos de complejidad espacial, Heapsort opera en $O(1)$, lo que lo hace más eficiente en memoria que los algoritmos con requisitos de espacio $O(n)$. En general, si bien Heapsort es un algoritmo de ordenamiento eficiente, sus problemas de estabilidad y rendimiento con datos complejos pueden llevar a algunos a preferir alternativas, especialmente en los casos en los que mantener el orden de elementos iguales es crucial.Para concluir, es importante remarcar que Heap, tanto como otra estructuras de datos tiene sus ventajas, la cual ofrece varias aplicaciones ofreciendo un menor costo computacional, a la hora de ordenar grandes cantidades datos, como lo era la problmática de este taller. Sin embargo las aplicaciones del heap van mucho más alla del problema de *sorting*, si no que es utilizado en la implemtación de colas de prioridad o algoritmos de gráfos como $A*$, Dijkstra, o Prim.

Referencias

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press