

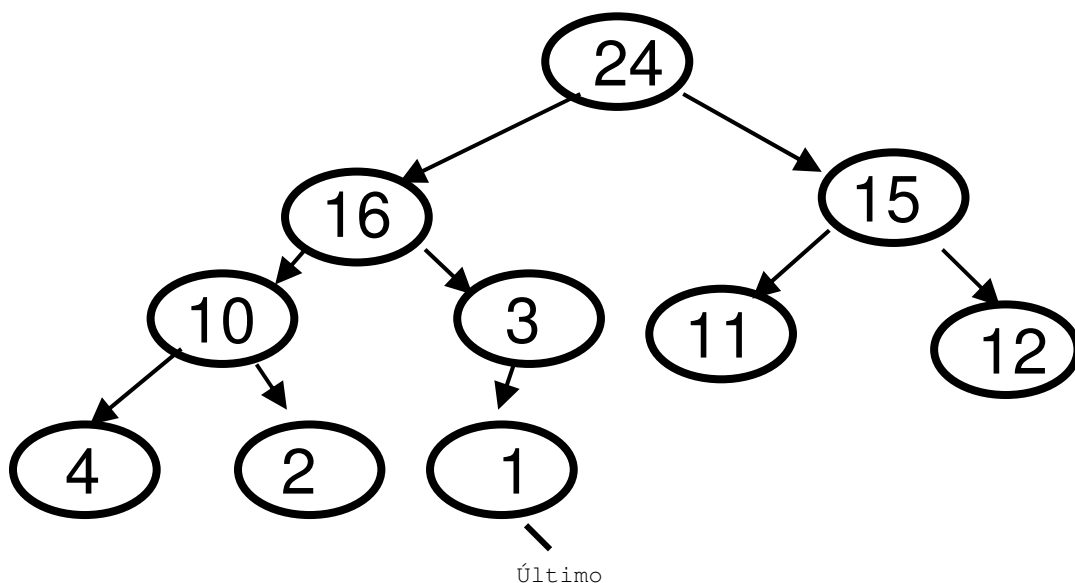
Heapsort

Definición y propiedades de un heap

Un heap es un montículo (traducción literal).

Un heap es un árbol binario casi completamente balanceado con el que se representa un conjunto de elementos, de tal manera que cada vértice del árbol es menor que sus dos hijos; así, en la raíz del árbol se encontrará al elemento mas pequeño del heap. Llamaremos con último al índice de la hoja situada al extremo derecho del heap.

Por ejemplo, el siguiente árbol binario es un heap.



Se dice que un árbol está completamente balanceado si tiene 2^i vértices en el nivel i , para i de $1..k$, donde k es el nivel más profundo del árbol; y se dice que está casi completamente balanceado si le faltan algunos vértices en el ultimo nivel; por lo tanto, las hojas del árbol se encontraran en los niveles k y $k-1$; los vértices en el nivel mas profundo se ubican en posiciones consecutivas de izquierda a derecha.

En un heap, cada subárbol -con raíz en cualquier vértice- es también un heap, y un árbol con un solo vértice también es un heap. La relación entre el numero de vértices de un heap y la altura del árbol está dada por

$$h = \text{piso}(\lg n)$$

Un heap se puede representar mediante un vector de acuerdo con la siguiente propiedad: si el vértice k está representado por el elemento de índice j en el vector, los hijos izquierdo y derecho de k serán los elementos de índices $2j$ y $2j+1$, respectivamente. Por otra parte, un elemento del vector de índice j , tiene a su padre en el elemento de índice $\text{piso}(j/2)$. Además, la raíz del heap será el primer elemento del vector.

Así, por ejemplo, el heap de arriba se puede representar por el vector:

24	16	15	10	3	11	12	4	2	1
									Último

De algunos ejemplos de arboles que no son heaps.

La operación heapify

Esta operación se realiza sobre un subárbol; es una operación recursiva que intercambia el vértice raíz del subárbol, k , con el menor de sus hijos, en caso de que por lo menos uno de ellos lo fuera, y luego realiza la misma operación heapify sobre el subárbol con raíz en el nuevo vértice k ; en caso de que ambos hijos fuesen mayor o igual que el vértice k , no se realiza acción alguna.

Para calcular el tiempo de ejecución de esta operación, primero debemos reconocer que la operación heapify es diseñada por división y conquista, donde la operación división incluye la comparación de la raíz con sus hijos para determinar quien es menor, la conquista consiste en realizar la misma operación heapify sobre uno de los dos subárboles, o ninguna, y la operación de combinación consiste en nada. Este proceso de división y conquista concluye cuando el subárbol se reduce a un solo vértice; entonces ya no es necesario efectuar mas divisiones.

Así, pues, el tiempo de ejecución de heapify sobre un árbol de altura h satisface la siguiente ecuación de recurrencia:

$$t(h) = 1 + t(h-1)$$

donde h es la altura del árbol.

Con la condición de borde

$$t(1) = 1$$

que corresponde a la operación heapify de un árbol de altura 1.

La solución de esta ecuación es

$$t(h) = 1 + 1 + \dots + 1$$

donde la sumatoria de la derecha tiene h términos, por lo tanto:

$$t(h) = h$$

y

$$t(n) = \text{piso}(\lg n)$$

Finalmente

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

Como insertar elementos

La operación **insertar** introduce un elemento en el heap; este elemento se copia en la siguiente posición a ultimo, y se filtra hacia arriba hasta la raíz en caso de ser necesario, es decir el elemento que se inserta se compara con su padre y se intercambia con el en caso de que sea menor, hasta encontrar un vértice que sea menor o igual. En el peor de los casos esta operación requiere de h comparaciones; por lo tanto, su tiempo de ejecución será:

$$t(h) = h$$

o sea,

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

Como eliminar elementos

Para eliminar un elemento de un heap, se sustituye la raíz del heap con el último elemento y se realiza la operación heapify sobre la raíz del nuevo árbol; el vértice último se reduce en 1.

Por lo tanto, la operación **eliminación** tiene un tiempo de ejecución:

$$t(n) = h,$$

o sea,

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

que es el tiempo necesario para realizar la operación heapify sobre un árbol de altura h .

CrearHeap

CrearHeap(A, n)

Esta operación crea un heap a partir de un vector A de dimensión n . Se recorre el vector A en sentido descendente desde el elemento de índice n hasta el de índice 1, realizando la operación heapify sobre cada uno de estos elementos; en realidad, solo es necesario hacer el recorrido desde el elemento de índice $\text{piso}(n/2)$, puesto que los elementos desde $\text{techo}(n/2)$ hasta n son hojas y la operación heapify sobre una hoja consiste en hacer nada.

Heapsort

Heapsort(A, n)

Esta operación ordena en orden ascendente los elementos de un vector de dimensión n . Primero se construye un heap a partir de los elementos en el vector, mediante la operación CrearHeap, y luego, progresivamente, se eliminan los n elementos del heap. Esto se hace intercambiando el primer elemento del heap con el de índice último, último se disminuye en 1, y se realiza la operación heapify sobre el primer elemento o raíz del heap, hasta que último sea igual a 0. El vector queda ordenado en orden descendente.

Este ordenamiento se realiza in situ, es decir sobre el vector que representa a los datos por ordenar; por lo tanto no se requiere de mas memoria que la necesaria para almacenar dichos datos.

Análisis de Heapsort.

El tiempo de ejecución de heapsort es la suma de los tiempos para crear el heap a partir de la secuencia en el vector A y de los tiempos para eliminar los n elementos del heap.

$$t(n) = t_{\text{crearheap}}(n) + t_{\text{eliminar}}(n)$$

ambos tiempos están acotados por arriba por una función que pertenece a $O(n \lg n)$; por lo tanto el tiempo de ejecución de heapsort tendrá la misma cota superior.

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

Para calcular estos tiempos se podrían hacer cálculos con mayor o menor precisión, pero el resultado será el mismo.

Presentaremos a continuación los cálculos menos precisos.

$$t_{\text{crearheap}}(n) \leq \text{piso}(n/2) \cdot h$$

donde h es la altura del heap mas alto sobre el cual se realiza la operación `heapify`, y el factor $\text{piso}(n/2)$ corresponde al número de veces que se realiza la operación `heapify`;

Como $h = \text{piso}(\lg(n))$, y $\text{piso}(x) \leq x$

$$t_{\text{crearheap}}(n) = O(\text{nlg } n)$$

Ahora, en cuanto al tiempo de eliminación, considerando la situación mas desventajosa

$$t_{\text{eliminacion}}(n) \leq n \cdot h$$

donde el factor n corresponde a los n elementos que hay que eliminar del heap, y el factor h corresponde al tiempo para realizar la operación `heapify` en la situación mas desventajosa, sobre un árbol de altura h . Luego, puesto que

$$h = \text{piso}(\lg(n)), \text{ y } \text{piso}(x) \leq x$$

concluimos que

$$t_{\text{eliminacion}}(n) = O(\text{nlg } n)$$

Ahora seremos mas precisos en el cálculo de estos tiempos de ejecución. Primero calcularemos el tiempo para crear un heap a partir del vector A . Sea h la altura del árbol binario casi completamente balanceado que contiene a los elementos del vector A ; también supongamos que el árbol esta balanceado, de tal manera que trabajaremos con la situación mas adversa. Para crear el heap tendremos que realizar la operación `heapify` sobre cada uno de los vértices de los niveles $h-1, h-2, \dots, 1$; para cada una de estas operaciones el tiempo de ejecución será $1, 2, 3, \dots, h-1$, respectivamente; por lo tanto

$$t = 1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + \dots + (h-1) \cdot 2^1 + h \cdot 2^0$$

A partir de esta expresión se puede demostrar que:

$$t(n) \text{ pertenece a } teta(\lg(n))$$

Esto quiere decir que en el peor de los casos, el tiempo para crear un heap con n elementos crece de manera lineal con n .

Para eliminar los n elementos del heap en el peor de los casos se necesitará:

$$t = h \cdot 2^h + (h-1) \cdot 2^{h-1} + \dots + 2 \cdot 2^2 + 1 \cdot 2^1$$

de donde podemos demostrar que:

$$t(n) \text{ pertenece a } teta(n \lg(n))$$

Finalmente, como el tiempo de ejecución de `heapsort` será la suma del tiempo para crear un heap y el tiempo para eliminar los mismos, este tiempo en el peor de los casos será la suma de una función que crece linealmente con otra que crece como $\text{nlg}(n)$; por lo tanto, para `heapsort`

$$t(n) \text{ pertenece a } teta(n \cdot \lg(n))$$

Aplicaciones de la estructura de datos heap:

- Cola prioritaria
- Algoritmos voraces
- Simulación de eventos discretos
- Implementación de pilas y colas.