Heaps

Algoritmos y Estructuras de Datos

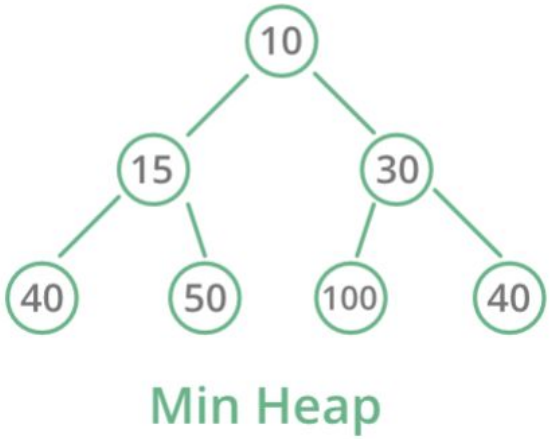
Plan del dia

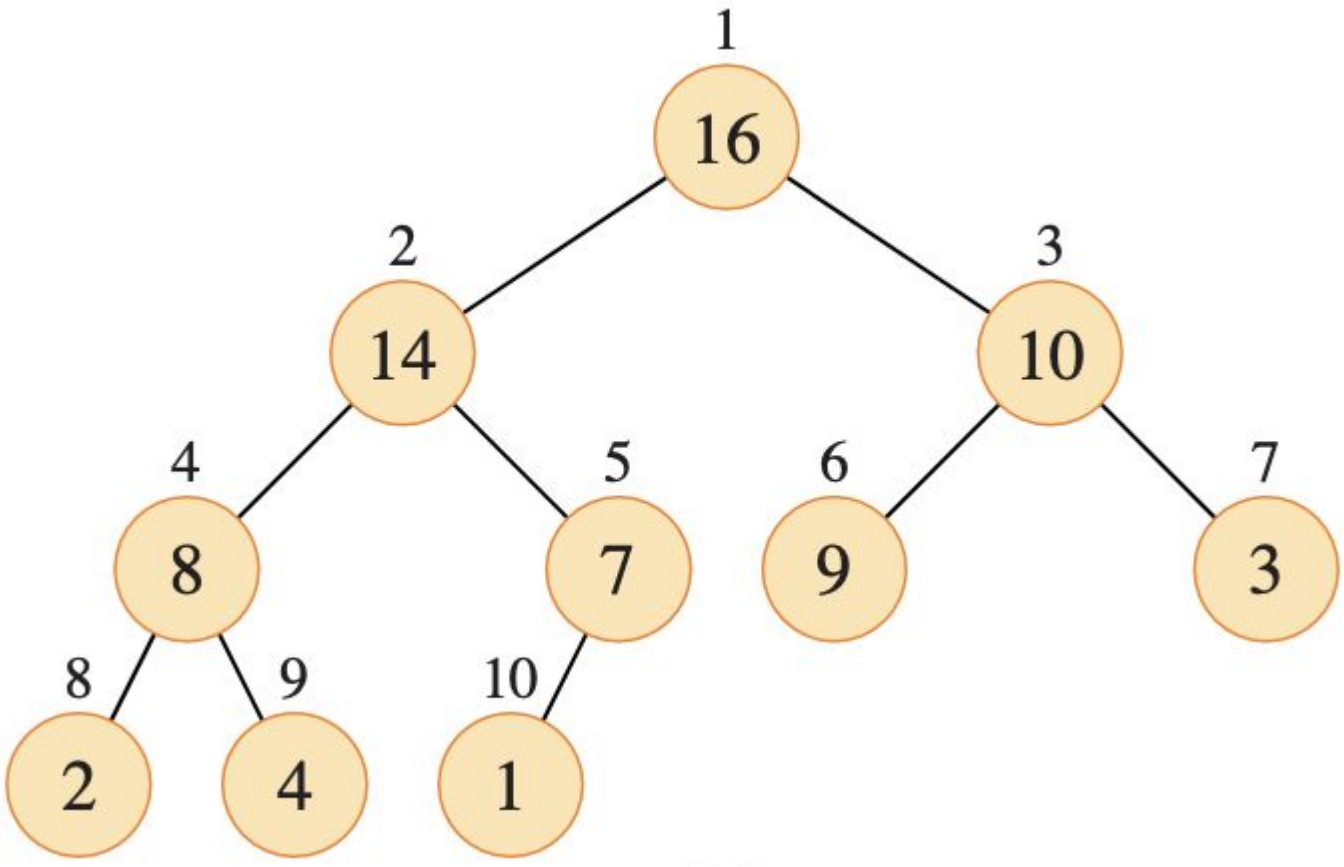
- Breve repaso de lo que es un heap
- Ejercicio de heap
- Break
- Repaso y ejercicios de tries con Ale

Repaso Heaps

¿Qué es un heap?

- Estructura de datos basada en árboles binarios la cual cumple con la propiedad de heap.
- Tipos de propiedades de heap:
  - Minheap
    - para cada nodo i, sus hijos son más grandes que el.
  - Maxheap
    - para cada nodo i, sus hijos son más chicos que el.

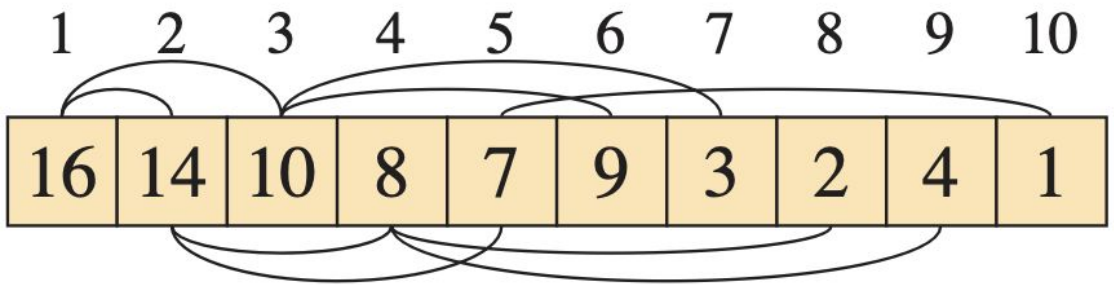




Ejemplo de max-heap

Implementación

- Los heaps se pueden implementar mediante arrays
- Un nodo en la posición i tiene:
  - su padre en la posición i/2
  - su hijo izquierda en la posición 2i
  - su hijo derecho en la posición 2i+1



Operaciones de un heap

Operación	Qué hace	Complejidad
próximo	devuelve el elemento de prioridad máxima (o minima si el heap es min-heap)	O(1)
insertar	agrega un elemento al heap, manteniendo la prioridad	O(log n)
eliminar	elimina un elemento del heap, manteniendo la prioridad	O(log n)
modificar	modifica un elemento del heap, manteniendo la prioridad	O(log n)

Operaciones de un heap

Operación	Qué hace	Complejidad
próximo	devuelve el elemento de prioridad máxima (o minima si el heap es min-heap)	O(1)
insertar	agrega un elemento al heap, manteniendo la prioridad	O(log n)
eliminar	elimina un elemento del heap, manteniendo la prioridad	O(log n)
modificar	modifica un elemento del heap, manteniendo la prioridad	O(log n)

¿Cómo el heap mantiene su prioridad?

Operaciones de un heap

Operación	Qué hace	Complejidad
próximo	devuelve el elemento de prioridad máxima (o minima si el heap es min-heap)	O(1)
insertar	agrega un elemento al heap, manteniendo la prioridad	O(log n)
eliminar	elimina un elemento del heap, manteniendo la prioridad	O(log n)
modificar	modifica un elemento del heap, manteniendo la prioridad	O(log n)

¿Cómo el heap mantiene su prioridad? utilizando heapify

✨ Heapify ✨

- Procedimiento el cual mantiene la propiedad en un heap.
- Dos tipos:
  - heapify up
    - arrancamos desde el nodo que agregamos y lo comparamos con su padre, si no se cumple la propiedad de heap, lo intercambiamos con su padre
    - este proceso sigue hasta llegar a la raíz o se restaura la propiedad de heap
  - heapify down
    - arrancamos desde la raíz, comparamos este nuevo elemento en la raíz con sus hijos; si la propiedad del heap se viola (por ejemplo, si es menor que uno de sus hijos en un *max heap*), se intercambia con el hijo más grande (en un *max heap*) o más pequeño (en un *min heap*).
    - complejidad: O(log n)

Usos de heapify

Característica	Heapify Up	Heapify Down
uso	Después de insertar un elemento	Después de eliminar un elemento
dirección	Hacia arriba (desde el nuevo nodo hasta la raíz)	Hacia abajo (desde la raíz hacia las hojas)
comparación	Comparación con el padre	Comparación con los hijos
situación / proposito	Para mantener la propiedad del heap tras una inserción	Para mantener la propiedad del heap tras una eliminación

Ejercicio

Dado un array de longitud n, obtener los k elementos más chicos del array.
requiere: k es menor al tamaño del arreglo

Ejemplos:

Input: *[1, 23, 12, 9, 30, 2, 50], K = 3*
Output: [*1, 2, 9]*

Input: *[11, 5, 12, 9, 44, 17, 2], K = 2*
Output: [*2, 5]*

Resolver el problema con una complejidad temporal: O(n log k)

Idea inicial

¿Cual seria una idea inicial para resolver el problema?

Idea inicial

¿Cual seria una idea inicial para resolver el problema?

- Opción 1:
  - recorremos el array k veces buscando el elemento más chico, teniendo en cuenta los que vamos encontrando.

Idea inicial

¿Cual seria una idea inicial para resolver el problema?

- Opción 1:
  - recorremos el array k veces buscando el elemento más chico, teniendo en cuenta los que vamos encontrando.
  - complejidad: O(kn)

Idea inicial

¿Cual seria una idea inicial para resolver el problema?

- Opción 1:
  - recorremos el array k veces buscando el elemento más chico, teniendo en cuenta los que vamos encontrando.
  - complejidad: O(kn)

- Opción 2:
  - ordenar el array
  - imprimir los primeros k elementos

Idea inicial

¿Cual seria una idea inicial para resolver el problema?

- Opción 1:
  - recorremos el array k veces buscando el elemento más chico, teniendo en cuenta los que vamos encontrando.
  - complejidad: O(kn)

- Opción 2:
  - ordenar el array
  - imprimir los primeros k elementos
  - complejidad: O(n log n)

Idea inicial

¿Cual seria una idea inicial para resolver el problema?

- Opción 1:
  - recorremos el array k veces buscando el elemento más chico, teniendo en cuenta los que vamos encontrando.
  - complejidad: O(kn)

- Opción 2:
  - ordenar el array
  - imprimir los primeros k elementos
  - complejidad: O(n log n)

¿Hay una forma más eficiente de resolver el problema?

Idea inicial

¿Cual seria una idea inicial para resolver el problema?

- Opción 1:
  - recorremos el array k veces buscando el elemento más chico, teniendo en cuenta los que vamos encontrando.
  - complejidad: O(kn)

- Opción 2:
  - ordenar el array
  - imprimir los primeros k elementos
  - complejidad: O(n log n)

¿Hay una forma más eficiente de resolver el problema?

¡Si!, utilizando heaps

Resolviendo el problema con heaps

Resolviendo el problema con heaps

- Opción 3:
  - poner todos los elementos del array en un min-heap
  - sacar los primeros k elementos del heap

Resolviendo el problema con heaps

- Opción 3:
  - poner todos los elementos del array en un min-heap
  - sacar los primeros k elementos del heap
  - complejidad O(n log n)

Resolviendo el problema con heaps

- Opción 3:
  - poner todos los elementos del array en un min-heap
  - sacar los primeros k elementos del heap
  - complejidad O(n log n)

- Opción 4:
  - mantener un max-heap que tenga solo los k mínimos elementos.
  - iteramos sobre los elementos del array
    - si el heap está lleno, sacamos el elemento de mayor prioridad
    - insertamos el elemento

Resolviendo el problema con heaps

- Opción 3:
  - poner todos los elementos del array en un min-heap
  - sacar los primeros k elementos del heap
  - complejidad O(n log n)

- Opción 4:
  - mantener un max-heap que tenga solo los k mínimos elementos.
  - iteramos sobre los elementos del array
    - si el heap está lleno, sacamos el elemento de mayor prioridad
    - insertamos el elemento
  - Complejidad O(n log k)

Resolviendo el problema con heaps

- Opción 3:
  - poner todos los elementos del array en un min-heap
  - sacar los primeros k elementos del heap
  - complejidad O(n log n)

- Opción 4:
  - mantener un max-heap que tenga solo los k mínimos elementos.
  - iteramos sobre los elementos del array
    - si el heap está lleno, sacamos el elemento de mayor prioridad
    - insertamos el elemento
  - Complejidad O(n log k)

Esta opción resuelve el problema con la complejidad que buscábamos 🥳

Implementación

```
proc BuscarMenorK(in s: array<int>, k: int): array<int> {
h := new ColaDePrioridadLog<int>.colaDePrioridadVacia() // la cola de prioridad es de prioridad maxima (max-heap)
i := 0
while i < s.length
        if (h.tamaño() >= k) then
                tope := h.consultarMax
                if (tope > s[i]) then
                        h.sacarMayor()
                        h.insertar(s[i])
                endif
        endif
        if (h.tamaño() < k) then
                h.insertar(s[i])
        endif
        i := i+1

// ponemos en res los elementos obtenidos por el heap, en esta implementación los guardamos de menor a mayor, pero
// podriamos guardarlos en el orden que queramos
i := k-1
kmenores := new array<int>(k)
while i >= 0
        kmenores[i] = h.sacarMayor()
        i := i - 1
}
```

Complejidades de la implementación

```
proc BuscarMenorK(in s: array<int>, k: int): array<int> {
 h := new ColaDePrioridadLog<int>.colaDePrioridadVacia() // la cola de prioridad es de prioridad maxima (max-heap)
 //O(1)
 i := 0 //O(1)
 while i < s.length // O(n log k)
      if (h.tamaño() >= k) then // hacer esta comparacion nos cuesta O(1) pues h.tamaño() cuesta O(1)
           tope := h.consultarMax //O(log k)
           if (tope > s[i]) then //O(1)
                h.sacarMayor() //O(log k)
                h.insertar(s[i]) //O(log k)
           endif
      endif
      if (h.tamaño() < k) then // O(1)
           h.insertar(s[i]) //O(log k)
      endif
      i := i+1 //O(1)

 // ponemos en res los elementos obtenidos por el heap, en esta implementación los guardamos de menor a mayor,
 pero
 // podriamos guardarlos en el orden que queramos
 i := k-1 //O(1)
 kmenores := new array<int>(k) //O(k)
 while i >= 0 //O(k log k)
      kmenores[i] = h.sacarMayor() //O(log k)
      i := i - 1 //O(1)
 }
 Sumando las Complejidades nos quedaria O(1 + 1 + n log n + 1 + k + k log k) = O(n log k) pues k <= n
```

¿Preguntas?