**Laboratorio S12. AYED-02**

Jefer Alexis González Romero

**DOCUMENTO TÉCNICO**

**Implementación de montón binario minimal**

**Requisitos**

**Especificación**

heap = MinHeap([25, 13, 20, 8, 7, 17, 2, 5, 4])

heap.data = [2, 4, 5, 8, 7, 17, 20, 13, 25]

**4**

**7**

**5**

**2**

**17**

**20**

**8**

**13**

**25**

**Entrada:**

Se recibe un arreglo de elementos.

**Salida:**

Se hace un heap con la propiedad de MinHeap.

**Diseño**

**Estrategia**

La clase MinHeap tendrá como atributo “data” que representa el montón en un arreglo, además poseerá las funciones de insertar y eliminar elementos (.insert(element), .remove(element)). Para ver el montón como un árbol se hacen las funciones .parent(index), .lelft(index) y .right(index) que dan las ubicaciones en el arreglo del padre, el elemento a la izquierda y el de la derecha respectivamente de un elemento ubicado en “index”, junto a estos se agregan funciones que retornan el valor de cada uno.

Para mantenerla propiedad de MinHeap se crea la función min\_heapify y para construir un heap con esta propiedad “build\_heap”, y por último para obtener el elemento con menor valor get\_top().

Adjunto el programa con nombre “Heap”.

**Casos prueba**

|  |  |  |
| --- | --- | --- |
| Entrada | Justificación | Salida |
| heap = MinHeap([25, 13, 20, 8, 7, 17, 2, 5, 4]) | Crear un heap con propiedad de MinHeap | heap.data = [2, 4, 17, 5, 7, 25, 20, 13, 8] |
| heap.remove(4) | Función remove | heap.data = [2, 7, 5, 8, 25, 20, 13, 17] |
| heap.insert(3) | Función insert | heap.data = [2, 3, 5, 7, 25, 20, 13, 17, 8] |
| heap.get\_top(15) | Obtener el menor valor del heap | 2  heap.data = [3, 5, 7, 8, 20, 13, 17, 25] |

**Análisis**

**Temporal**

**min\_heapify**

n el mejor de los casos T(n) = Ω(1)

En el peor de los casos T(n) = O(logn)

**build\_heap**

n el mejor de los casos T(n) = Ω(1)

En el peor de los casos T(n) = O(n)

**Insert – remove – get\_top**

n el mejor de los casos T(n) = Ω(1)

En el peor de los casos T(n) = O(nlogn)

**Código**

**Documentación**

Dentro del código

**DOCUMENTO TÉCNICO**

**Implementación de las funciones de HeapSort para ambas versiones de montón binario**

**Requisitos**

**Especificación**

**Ejemplo:**

A = MaxHeap([5, 13, 2, 25, 7, 17, 20, 8, 4])

A.data = [25, 13, 20, 8, 7, 17, 2, 5, 4]

A.heap\_size = 9

**8**

**2**

**20**

**17**

**25**

**13**

**7**

**5**

**4**

Se intercambian en data el primer elemento y el último.

A.data = [4, 13, 20, 8, 7, 17, 2, 5, 25]

Ahora para mantener la propiedad de max-heap se usa max-heapify con los primeros 8 elementos del arreglo y con índice 0.

A.data = [20, 13, 17, 8, 7, 4, 2, 5, 25]

A.heap\_size = 8

**8**

**2**

**27**

**4**

**20**

**13**

**7**

**5**

Ahora se vuelve hacer un intercambio, pero del primero con el penúltimo:

A.data = [5, 13, 17, 8, 7, 4, 2, 20, 25]

Se usa max-heapify con los primeros 7 elementos y con índice 0:

A.data = [17, 13, 5, 8, 7, 4, 2, 20, 25]

A.heap\_size = 7

**8**

**2**

**5**

**4**

**17**

**13**

**7**

En este caso, este proceso se hace 9 veces, el cual es el número de elementos que tiene el heap.

Intercambio:

A.data = [2, 13, 5, 8, 7, 4, 17, 20, 25]

Max-heapify:

A.data = [13, 8, 5, 2, 7, 4, 17, 20, 25]

A.heap\_size = 6

**2**

**5**

**4**

**13**

**8**

**7**

Intercambio:

A.data = [4, 8, 5, 2, 7, 13, 17, 20, 25]

Max-heapify:

A.data = [8, 7, 5, 2, 4, 13, 17, 20, 25]

A.heap\_size = 5

**2**

**5**

**8**

**7**

4

Intercambio:

A.data = [4, 7, 5, 2, 8, 13, 17, 20, 25]

Max-heapify:

A.data = [7, 4, 5, 2, 8, 13, 17, 20, 25]

A.heap\_size = 4

**2**

**5**

**7**

**4**

Intercambio:

A.data = [2, 4, 5, 7, 8, 13, 17, 20, 25]

Max-heapify:

A.data = [5, 4, 2, 7, 8, 13, 17, 20, 25]

A.heap\_size = 3

**2**

**5**

**4**

Intercambio:

A.data = [2, 4, 5, 7, 8, 13, 17, 20, 25]

Max-heapify:

A.data = [4, 2, 5, 7, 8, 13, 17, 20, 25]

A.heap\_size = 2

**4**

**2**

Intercambio:

A.data = [2, 4, 5, 7, 8, 13, 17, 20, 25]

Max-heapify:

A.data = [2, 4, 5, 7, 8, 13, 17, 20, 25]

A.heap\_size = 1

**2**

**Entrada:**

Se recibe un montón binario.

**Salida:**

Los elementos del heap ordenados en “data”

**Diseño**

**Estrategia**

Se añade un atributo a MaxHeap y MinHeap que se llamará “heap\_size”, en este se guardará la posición del último elemento que tiene el heap y que están almacenado dentro de “data”. Este atributo se alterará a la hora de insertar o remover un elemento del montón.

Dentro de data se intercambian el primer elemento y el que tenga como índice el valor de heap\_size. Se le resta uno a heap\_size y para mantener la propiedad de la política del heap se usa el heapify correspondiente, el cual formará un nuevo heap con el número de elementos que indique heap\_size.

El proceso indicado anteriormente se hace n veces, donde n corresponde al número de elementos que hay en el heap.

Adjunto el programa con nombre “Heap”.

**Casos prueba**

|  |  |  |
| --- | --- | --- |
| Entrada | Justificación | Salida |
| heap = MaxHeap([5, 13, 2, 25, 7, 17, 20, 8, 4])  heap.data = [25, 13, 20, 8, 7, 17, 2, 5, 4] | Con política de MaxHeap | heap.data = [2, 4, 5, 7, 8, 13, 17, 20, 25] |
| Heap = MinHeap([5, 13, 2, 25, 7, 17, 20, 8, 4])  heap.data = [2, 4, 5, 8, 7, 17, 20, 13, 25] | Con política de MinHeap | heap.data = [25, 20, 17, 13, 8, 7, 5, 4, 2] |

**Análisis**

**Temporal**

En el mejor de los casos T(n) = Ω(1)

En el peor de los casos T(n) = O(nLn(n))

**Código**

**Documentación**

Dentro del código

**DOCUMENTO TÉCNICO**

**Implementación de la clase cola de prioridad**

**Requisitos**

**Especificación**

tareas = PriorityQueue([(tarea1, 3), (tarea2, 5), (tarea3, 1), (tarea4, 7)], True)

tareas.data = [('tarea4', 7), ('tarea2', 5), ('tarea3', 1), ('tarea1', 3)]

**Tarea4**

**Tarea3**

**Tarea1**

**Tarea2**

**Entrada:**

Se recibe un conjunto de elementos y la política que se va a manejar

**Salida:**

Se hace el PriorityQueue con los elementos dados.

**Diseño**

**Estrategia**

La clase PriorityQueue tiene como atributos data (maneja los elementos en un arreglo) y policy que indica que política se esta aplicando, ya sea la de MinHeap o MaxHeap. Esta clase tiene dos funciones, una para encolar nuevos elementos (.enqueue(element)) y otra para desencolar elementos (.dequeue()). Las funciones operarán dependiendo de la política que se este usando, si se usa MaxHeap, entonces dequeue devolverá el elemento que tenga la llave de mayor valor, y al contrario pasará con MinHeap que dará el menor.

Adjunto el programa con nombre “PriorityQueue”.

**Casos prueba**

|  |  |  |
| --- | --- | --- |
| Entrada | Justificación | Salida |
| tareas = PriorityQueue([(‘tarea1’, 3), (‘tarea2’, 5), (‘tarea3’, 1), (‘tarea4’, 7)], True) | Con política de MaxHeap | tareas.data = [('tarea4', 7), ('tarea2', 5), ('tarea3', 1), ('tarea1', 3)] |
| tareas = PriorityQueue([(‘tarea1’, 3), (‘tarea2’, 5), (‘tarea3’, 1), (‘tarea4’, 7)], True) | Con política de MinHeap | heap.data = [('tarea3', 1), ('tarea2', 5), ('tarea1', 3), ('tarea4', 7)] |

**Análisis**

**Temporal**

n el mejor de los casos T(n) = Ω(1)

En el peor de los casos T(n) = O(n)

**Código**

**Documentación**

Dentro del código

**Prototipo**

Dentro del programa con nombre “PriorityQueue”.

**Casos de prueba**

1. Le llega una lista de pedidos
2. Realiza una entrega
3. Se introduce un nuevo pedido
4. Hace 2 domicilios
5. Termina los pedidos