DOCUMENTO TÉCNICO – MONTONES BINARIOS

INTEGRANTES:

Diego Cárdenas

Felipe Calvache

Zayra Gutiérrez

PROFESOR: SEBASTIÁN CAMILO MARTÍNEZ REYES

ALGORITMOS Y ESTRUCTURAS DE DATOS

Resumen:

En la combinación de estructuras de datos para montones binarios, se utilizarán arreglos, arboles binarios, nodos y diccionarios. Se enfocará en la raíz del árbol, que es el elemento en la primera posición del arreglo cuando se utiliza una política FIFO. Dependiendo de la estructura y prioridad del árbol, se aplicará la política correspondiente para encolar el arreglo y asegurar que los padres de cada nodo cumplan con la condición requerida. Esta condición puede ser que la raíz sea mayor que sus hijos o que la raíz sea menor que sus hijos. Para facilitar esto, se crearán dos objetos llamados PriorityQueue y Heap, los cuales se explicarán más adelante.

Entrada y salida:

Para el procesamiento de entrada, se utilizarán arreglos que contengan triplas que incluyen datos para el ordenamiento: "fecha", "nombre" y "edad". En este caso, la prioridad de ordenamiento se basará en la edad y puede ser mínima o máxima.

La implementación del código se basa en una estructura de datos conocida como montón binario, también denominada "Heap", que se caracteriza por tener un valor en cada nodo o raíz del árbol que es mayor o igual al de sus hijos. El código base incluye funciones como BuildMaxHeapify y BuildMinHeapify, así como otras que permiten:

- Obtener la raíz del montón binario con GetRoot().

- Insertar un nuevo nodo en el montón y verificar si el elemento es mayor o menor con Insert, que se implementa en InsertMax o InsertMin respectivamente.

- Eliminar un nodo del montón y verificar si el elemento es mayor o menor con Delete, que se implementa en DeleteMax o DeleteMin respectivamente, y luego reorganizar el montón.

- Actualizar un valor del nodo utilizando las funciones Delete e Insert, eliminando el valor del nodo a actualizar, insertando el nuevo valor y reorganizando el montón con las construcciones anteriores.

En este trabajo, se ha mejorado el código base al implementar nuevas funciones como minHeapify, y se han importado librerías adicionales para lograr mejores resultados y ejemplos que permitan verificar el correcto desempeño del código.

Código:

```python
import math
import random
From random import randint
From datetime import date
From datetime import datetime

class PriorityQueue:
    def init (self, A, property_check=0, config=True):
        self.heap = Heap(A, property_check, config)
        self.config =config

    def push(self, element):
        if self.config:
            self.heap.insertMax (element)
        else:
            self.heap.insertMin (element)
    def pop (self):
        element = self.heap.getRoot()
        if self.config:
            self.heap.deleteMax (element)
        else:
            self.heap.deleteMin (element)
        return element
    def len(self):
        return len(self.heap)
```

```python
class Heap:
    def init (self, A, property_check = 0, config=True):
        self.property_check = property_check
        self.elements = []
        self.config = config
        if config:
            for e in A:
                self.insertMax(e)
        else:
            for e in A:
                self.insertMin(e)

    def getElements(self):
        return self.elements

    def left(self, i):
        return 2*i + 1

    def right(self, i):
        return 2*(i + 1)

    def height(self):
        return math.floor(math.log(len(self.elements), 2)) + 1

    def parent(self, i):
        return (i-1)//2 if i % 2 != 0 else (i//2 - 1)
```

```
    def buildMaxHeapify(self):
        for i in range(len(self.elements)//2, -1, -1):
            self.maxHeapify(i)

    def buildMinHeapify(self):
        for i in range(len(self.elements)//2, -1, -1):
            self.minHeapify(i)

    def getRoot(self):
        return self.elements[0]

    def insertMax(self, e):
        self.elements.append(e)
        self.buildMaxHeapify()

    def insertMin(self, e):
        self.elements.append(e)
        self.buildMinHeapify()

    def deleteMax(self, key):
        self.elements.remove(key)
        self.buildMaxHeapify()

    def deleteMin(self, key):
        self.elements.remove(key)
        self.buildMinHeapify()

```

```
def updateMax(self, old_key, new_key):
    self.deleteMax(old_key)
    self.insertMax(new_key)

def updateMin(self, old_key, new_key):
    self.deleteMin(old_key)
    self.insertMin(new_key)

def maxHeapify(self, root):
    left, right, largest = self.left(root), self.right(root), root
    if left < len(self.elements) and self.elements[root][self.property_check] < self.elements[left][self.property_check]:
        largest = left
    if right < len(self.elements) and self.elements[largest][self.property_check] < self.elements[right][self.property_check]:
        largest = right
    if largest != root:
        self.elements[largest], self.elements[root] = self.elements[root], self.elements[largest]
        self.maxHeapify(largest)

def minHeapify(self, root):
    left, right, largest = self.left(root), self.right(root), root
    if left < len(self.elements) and self.elements[root][self.property_check] > self.elements[left][self.property_check]:
        largest = left
    if right < len(self.elements) and self.elements[largest][self.property_check] > self.elements[right][self.property_check]:
        largest = right
    if largest != root:
        self.elements[largest], self.elements[root] = self.elements[root], self.elements[largest]
        self.minHeapify(largest)
```

```
107
108     def len(self):
109         return len(self.elements)
110
111 class Persona:
112     def init(self, nombre="", edad=1):
113         self.nombre = nombre
114         self.edad = edad
115
116     def str(self):
117         return str({self.nombre, self.edad})
118
119     def lt(self, other):
120         return self.edad < other.edad
121
122 def main():
123     MAX_BOUND = 72
124     MIN_BOUND = 18
125     now = date.today()
126     nombres = ["Juliana", "Daniela", "Andrea", "Juanes", "Alberto"]
127     data = [(str(now), random.choice(nombres), randint(MIN_BOUND, MAX_BOUND)) for i in range(10)]
128     pq = PriorityQueue(data, 1, False)
129     while len(pq) > 0:
130         print('El siguiente a atender es: ', pq.pop())
131
132 main()
```