**COMPLEJIDAD TEMPORAL**

Método para agregar una tarea

public String addTask(String Id, String name, String description, String dateLimit, int priority) {

        Task task = new Task(Id, name, description, dateLimit, priority);

        if (task.getPriority() == 0) {

            queue.add(task);

            agenda.insert(task.getId(),task);

        } else {

            priorityTasks.insert(task.getPriority(), task);

            agenda.insert(task.getId(),task);

        }

        return "Se agrego correctamente";

    }

1. Crear una instancia de **Task**: O(1)
2. Verificar la prioridad de la tarea: O(1)
3. Si la prioridad es igual a 0:
   * Agregar la tarea a **queue**: O(1)
   * Insertar la tarea en **agenda**: O(1)
4. Si la prioridad no es igual a 0:
   * Insertar la tarea en **priorityTasks**: O(1)
   * Insertar la tarea en **agenda**: O(1)

Dado que todas las operaciones son de tiempo constante, podemos sumarlas para obtener la complejidad total:

O(1) + O(1) + (O(1) + O(1)) = O(1)

Método para agregar un recordatorio

 public String addReminder(String Id, String name, String description, String dateLimit, int priority) {

        Reminder reminder = new Reminder(Id, name, description, dateLimit, priority);

        agenda.insert(Id, reminder);

        return "Se agrego correctamente";

    }

1. **Creación de la instancia de Reminder**:
   * La creación de una instancia de **Reminder** se realiza en tiempo constante (O(1)), ya que no depende del tamaño de los datos de entrada ni de ninguna estructura de datos. La operación se realiza en un tiempo fijo.
2. **Inserción en la estructura de datos agenda**:
   * La operación de inserción en **agenda** se realiza mediante la línea **agenda.insert(Id, reminder);**. La complejidad de esta operación depende de cómo esté implementada la estructura de datos **agenda**.
   * Si asumimos que la inserción en **agenda** se implementa de manera eficiente y es de tiempo constante (O(1)), entonces esta operación es O(1).

Dado que ambas operaciones son de tiempo constante (O(1)) en este análisis, la complejidad temporal total del método **addReminder** es O(1).

Método para modificar

public String modify(String modify, String Id, String m) {  
 Agenda agendaToModify = agenda.search(Id);  
 Agenda copy = new Task(agendaToModify);  
 switch (m) {  
 case "name":  
 agendaToModify.setName(modify);  
  
 break;  
 case "description":  
 agendaToModify.setDescription(modify);  
  
 break;  
 case "dateLimit":  
 agendaToModify.setDateLimit(modify);  
  
 break;  
 case "priority":  
 agendaToModify.setPriority(Integer.parseInt(modify));  
 break;  
  
 default:  
 break;  
 }  
 userAction(2, copy);  
 return "Se modifico correctamente";  
}

1. **Búsqueda en agenda**:
   * La operación de búsqueda en la estructura de datos **agenda** depende de su implementación. Si se realiza una búsqueda lineal, la complejidad sería O(n), donde n es el número de elementos en **agenda**.
2. **Creación de una copia de agenda**:
   * La creación de una copia de la estructura de datos **agendaToModify** se asume como una operación de tiempo constante (O(1)).
3. **Switch-Case**:
   * El bloque **switch** evalúa el valor de **m** y realiza una de las operaciones de modificación. Cada una de estas operaciones es de tiempo constante (O(1)).
4. **Llamada a userAction**:
   * La llamada a la función **userAction** se asume como una operación de tiempo constante (O(1)).

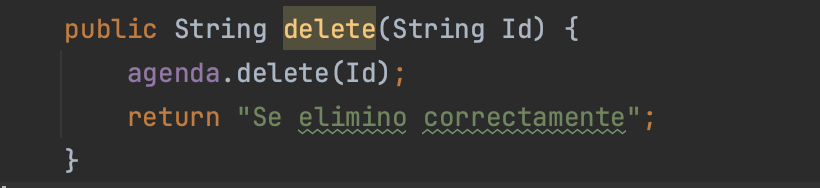
la complejidad sería O(1) en el peor de los casos debido a que las demás operaciones son de tiempo constante. Sin embargo, si la búsqueda en **agenda** es lineal, la complejidad sería O(n)

maxHeapify

public void maxHeapify(int position){  
 int left = getLeft(position);  
 int right = getRigth(position);  
 int largest = position;  
  
 if (left < heapSize){  
 if ( list.get(left).getKey().compareTo(list.get(position).getKey()) > 0 )largest = left;  
 }  
  
 if (right < heapSize){  
 if ( list.get(right).getKey().compareTo(list.get(largest).getKey()) > 0 ) largest = right;  
 }  
  
 if (largest != position){  
 HeapNode temporal = list.get(position);  
 list.set(position, list.get(largest));  
 list.set(largest, temporal);  
 maxHeapify(largest);  
 }  
}

1. **Cálculo de índices y asignación de variables locales**:
   * Las líneas donde se calculan los índices **left** y **right** a partir de la posición actual y se inicializa la variable **largest** son operaciones de tiempo constante (O(1)).
2. **Comprobación y comparación de valores**:
   * Las líneas donde se verifica si el índice **left** o **right** es menor que **heapSize** y se comparan los valores en esos índices son operaciones de tiempo constante (O(1)).
3. **Intercambio de elementos y llamada recursiva**:
   * Si se determina que es necesario realizar un intercambio (cuando **largest** es diferente de **position**), se intercambian elementos en la lista y se realiza una llamada recursiva a **maxHeapify**.
   * La llamada recursiva es la operación que determina la complejidad temporal de este método. En el peor de los casos, esta llamada puede descender a través de la altura del árbol, lo que da como resultado una complejidad de tiempo de O(log n), donde n es el número de elementos en el heap.

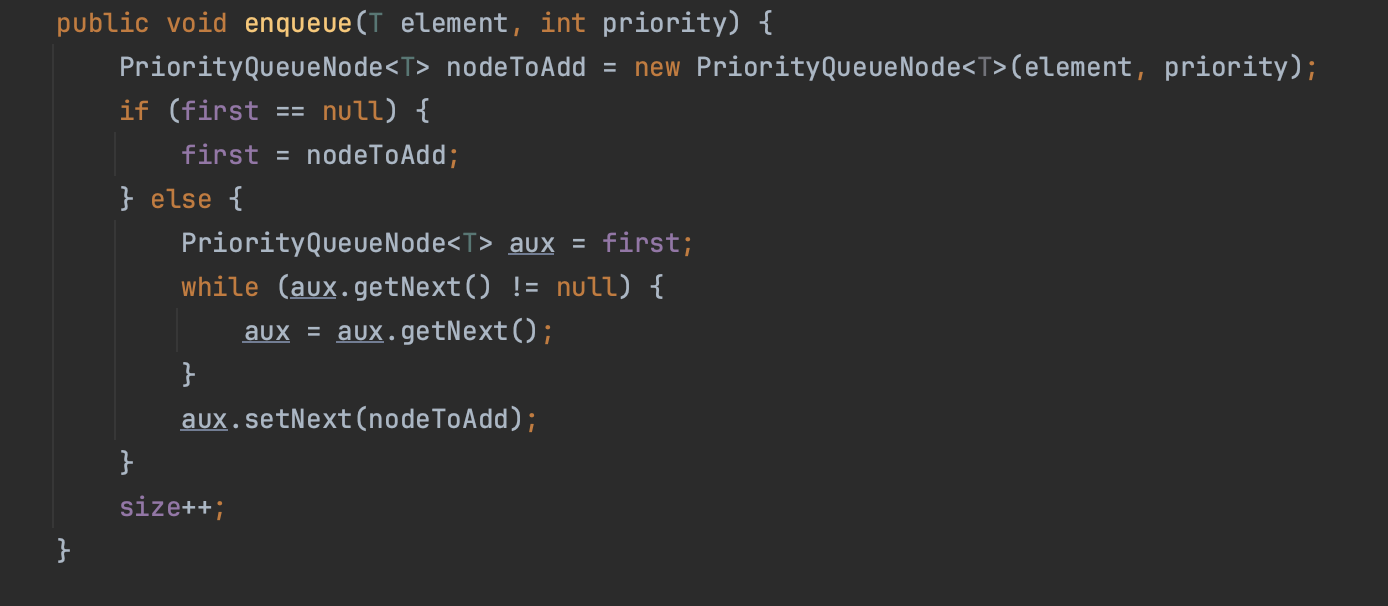
Método para eliminar



1. **agenda.delete(Id)**: Esta operación elimina una tarea de la agenda por su identificador **Id**, esta parte del método es O(1).
2. **return "Se eliminó correctamente"**: Esta operación simplemente devuelve una cadena de caracteres, lo cual es una operación de tiempo constante (O(1)).

Dado que ambas operaciones en el método son de tiempo constante, estas se sumaran para obtener la complejidad temporal total del método **delete** 🡪 O(1) + O(1) = O(1)

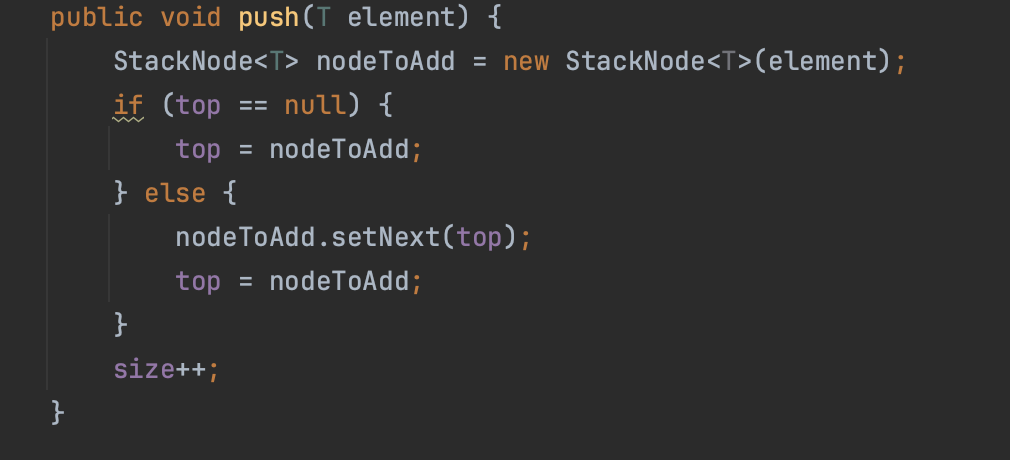
Método para agregar un elemento a la cola de prioridad



1. Crear un nuevo nodo **PriorityQueueNode** con el elemento y la prioridad especificados. Esto es una operación de tiempo constante, O(1).
2. Verificar si el **first** (primer elemento de la cola de prioridad) es nulo. Esto es una operación de tiempo constante, O(1).
3. Si el **first** no es nulo, el método entra en un bucle **while** que recorre la cola de prioridad hasta encontrar el último nodo. La complejidad de este bucle depende del tamaño de la cola de prioridad, por lo que es O(n), donde "n" es la cantidad de elementos en la cola de prioridad.
4. Una vez que se encuentra el último nodo, se agrega el nuevo nodo después de ese último nodo, lo cual es una operación de tiempo constante, O(1).
5. Incrementar el valor de **size** en 1. Esto es una operación de tiempo constante, O(1).

Dado que el bucle **while** es la parte que tiene una complejidad dependiente del tamaño de la cola, la complejidad temporal total del método **enqueue** será dominada por ese bucle. Por lo tanto, en el peor caso, cuando el bucle debe recorrer todos los elementos de la cola, la complejidad es O(n), donde "n" es la cantidad de elementos en la cola de prioridad. En el mejor caso, cuando la cola está vacía, la complejidad es O(1), ya que no se ejecuta el bucle y se agrega el elemento directamente al principio.

Método Push



1. Crear un nuevo nodo **StackNode** con el elemento especificado. Esta es una operación de tiempo constante, O(1).
2. Verificar si **top** es nulo. Operación de tiempo constante, O(1).
3. Si **top** no es nulo, el método realiza dos operaciones adicionales:
   * Asigna el nodo recién creado como el siguiente nodo del nodo **top**. Operación de tiempo constante, O(1).
   * Actualiza **top** para que apunte al nuevo nodo. Esto también es una operación de tiempo constante, O(1).
4. Incrementar el valor de **size** en 1. Esto es una operación de tiempo constante, O(1).

Dado que todas las operaciones del método son de tiempo constante, la complejidad temporal total del método **push** es O(1)