presentación.js workshop.java

Club de Algoritmia US {

[Universidad de Sevilla]

Sesión 12: Colas de prioridad

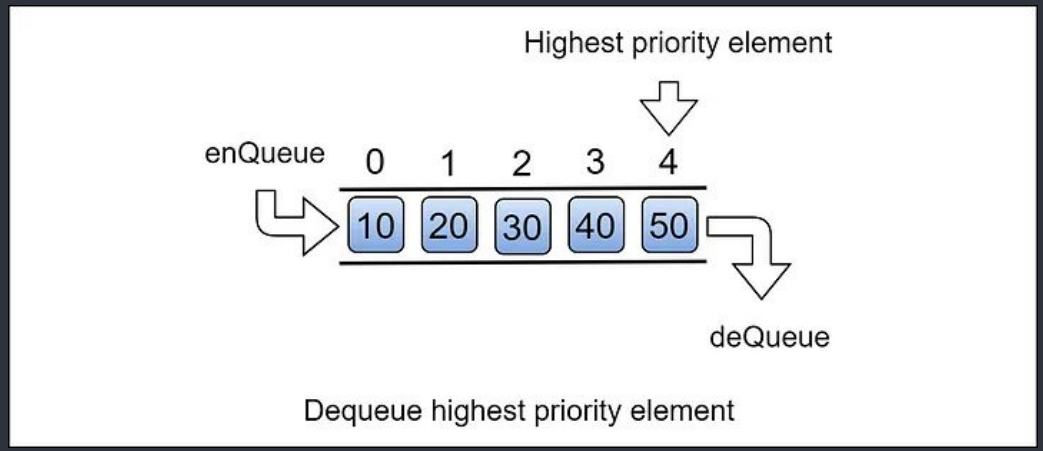
}

priority_queue.py

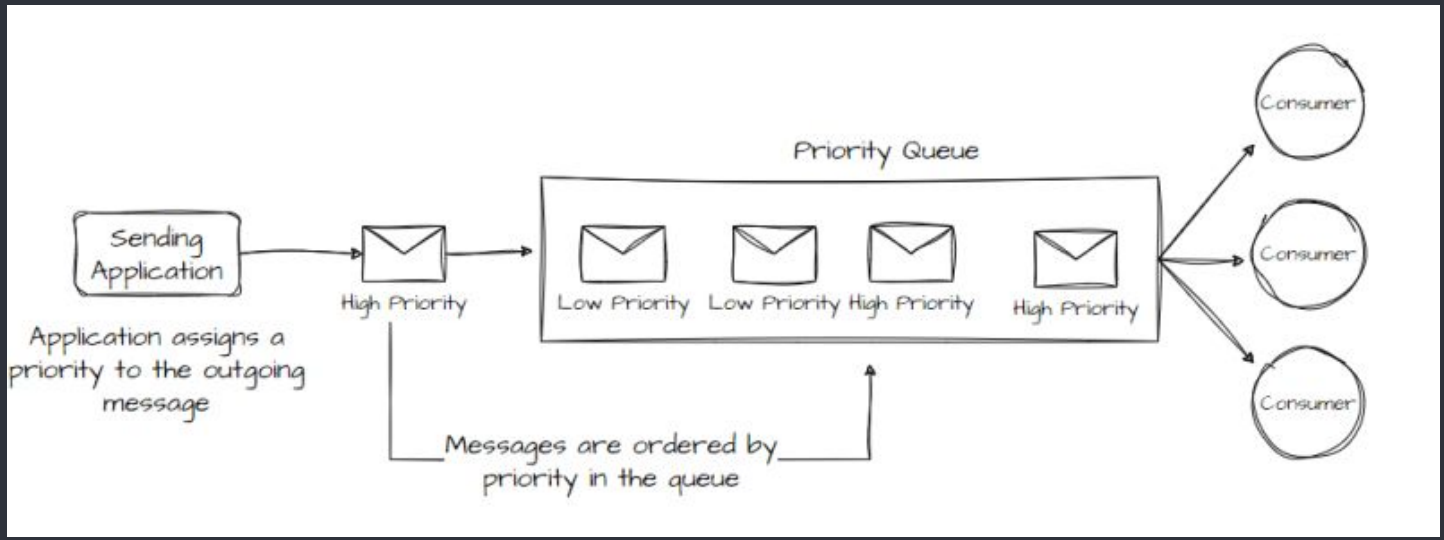
workshop.hs

Colas de prioridad

Es una estructura de datos parecida a la cola, pero con una característica adicional: cada elemento está asociado a un **valor de prioridad.**



Colas de prioridad: ‘Ejemplo’

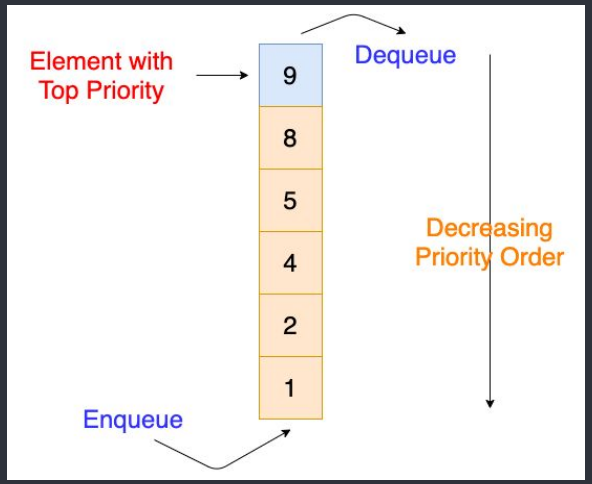


priority_queue.py

workshop.hs

Colas de prioridad: ‘Características’

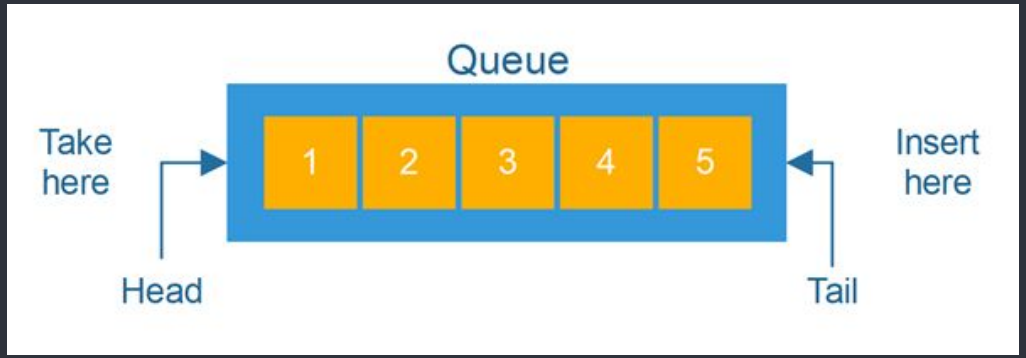
1. **Asociación de prioridad:** Cada elemento tiene una prioridad. Los elementos con alta prioridad se atienden antes que los de baja prioridad.



Colas de prioridad: ‘Características’

¿Qué ocurre si aparecen elementos con la misma prioridad?

Salen según su orden de entrada en la cola.



Colas de prioridad: ‘Características’

2. Operaciones básicas

- **Insertar (enqueue):** Agrega un nuevo elemento a la cola con una prioridad en específica.
- **Eliminar el máximo/mínimo (dequeue)**: Elimina y devuelve el elemento con la prioridad más alta (o baja, dependiendo de la implementación) de la cola.
- **Consultar máximo/mínimo:** Devuelve el elemento con la prioridad más alta (o baja) sin eliminarlo.

Colas de prioridad: ‘Implementaciones’

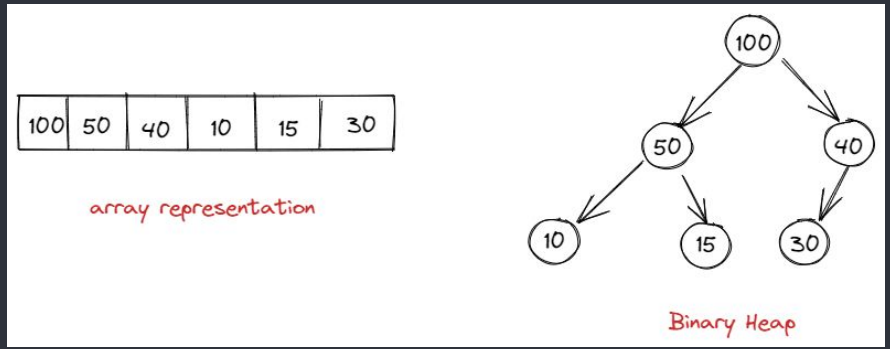
Las colas de prioridad pueden implementarse con listas enlazadas, binary heap o los árboles de búsqueda binarios, siendo los **binary heap** la opción más eficiente.

Implementation	peek	enqueue	dequeue
Linked list	O(1)	O(n)	O(1)
Binary heap	O(1)	O(log n)	O(log n)
Binary search tree	O(1)	O(log n)	O(log n)

Implementación: ‘Binary heap’

El montículo binario (o binary heap) es la implementación más utilizada.

Un **heap** es un árbol binario que contiene 2 restricciones adicionales.



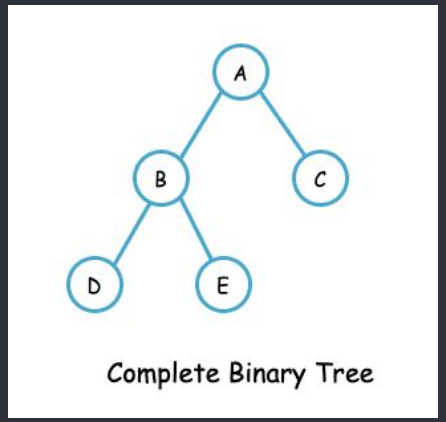
priority_queue.py

workshop.rs

Implementación: ‘Binary heap’

1. Árbol binario completo:

Todos los niveles del árbol están llenos, excepto el último nivel, que está lleno de izquierda a derecha.



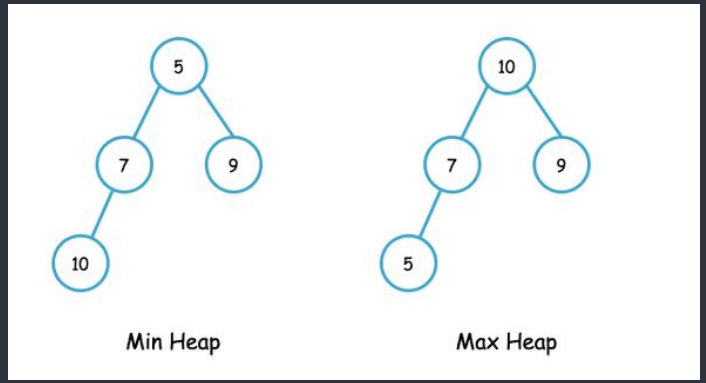
priority_queue.py

workshop.rs

Implementación: ‘Binary heap’

2. Propiedad de montículo:

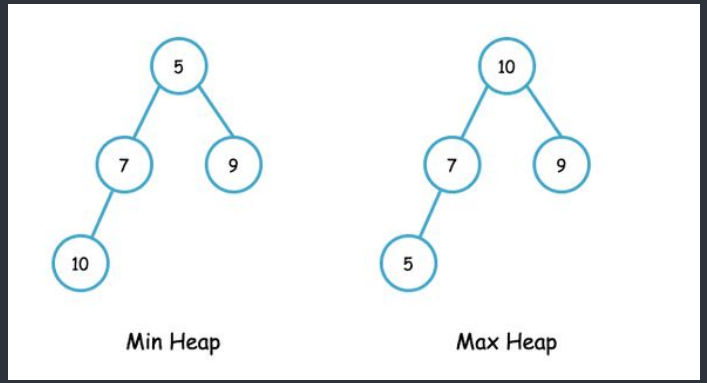
Min-Heap: En un min-heap, cada nodo es menor o igual que sus hijos. El elemento más pequeño está en la raíz.



Implementación: ‘Binary heap’

2. Propiedad de montículo:

Max-Heap: En un max-heap, cada nodo es mayor o igual que sus hijos. El elemento más grande está en la raíz.



Implementación: ‘Binary heap’

Para visualizar cómo funciona el binary tree, podéis acceder al siguiente enlace:

https://visualgo.net/en/heap

Implementación: ‘Binary heap’

priority_queue.py

workshop.py

Colas de prioridad en Python

En Python, puedes implementar una cola de prioridad utilizando el módulo heapq.

```python
import heapq

class PriorityQueue:
    def init(self):
        self.elements = []

    def is_empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]
```

priority_queue.py

workshop.cpp

Colas de prioridad en C++

En C++, puedes utilizar la clase “priority_queue” de la librería “queue” de la STL.

```cpp
#include <queue>
using namespace std;

int n, elem = 1
bool meh;

priority_queue<int> cola;
cola.push(elem);  // O(log n)
cola.pop();  // O(log n)
n = cola.size();  // O(1)
meh = cola.empty();  // O(1)
n = cola.top();  // O(1)
```

priority_queue.py

workshop.c

Colas de prioridad en C

En C tienes que implementar las colas de prioridad por tu cuenta (o nos puedes pedir el archivo).

```c
#include<stdio.h>
#include<malloc.h>

void insert();
void del();
void display();

struct node {
    int priority•,
    int info;
    struct node *next;
} *start, *q, *temp, *new;

typedef struct node *N;
```

priority_queue.py

workshop.c

Colas de prioridad en Java

En Java se pueden importar de la librería estándar:

```
import java.util.PriorityQueue
```

También podéis echar un vistazo a la implementación manual. Estará con las diapositivas.

```
class PriorityQueue {
        int[] pq;
        int n = 0;

        public PriorityQueue(int size) {
            pq = new int[size + 1];
        }

        public PriorityQueue(int[] initial) {
            this(initial.length);
            for (int i = 0; i < initial.length; i++) {
                insert(initial[i]);
            }
        }

       .
       .
       .
```

LeetCode - Número 1046

Problema: ‘Last Stone Weight’

{

> Dado un array de enteros representando el peso de piedras en cada paso tomamos las dos más pesadas y las hacemos chocar. Si tienen el mismo tamaño, ambas piedras se destruyen. Si no, la piedra menos pesada se destruye y a la pesada se le resta el peso de la otra. Iterando este proceso buscamos conocer el peso de la piedra final. Si no hay ninguna piedra al final devolveremos 0.

}

Problema: ‘Last Stone Weight’

{

Input: stones = [2,7,4,1,8,1]
Output: 1

Input: stones = [1]
Output: 1

}

Solución

```python
import heapq

class Solution:
    def lastStoneWeight(self, stones: List[int]) -> int:
        stones = [-stone for stone in stones]
        heapq.heapify(stones)

        while len(stones) > 1:
            first = -heapq.heappop(stones)
            second = -heapq.heappop(stones)

            if first != second:
                heapq.heappush(stones, -(first - second))

        return -stones[0] if stones else 0
```

LeetCode - Número 373

Problema: ‘Find K Pairs with Smallest Sums’

{

Dadas dos listas ordenadas de enteros y un entero k, encontrar los k pares de elementos (uno de cada lista) cuya suma sea menor.

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
Output: [[1,2],[1,4],[1,6]]

Input: nums1 = [1,1,2], nums2 = [1,2,3], k = 2
Output: [[1,1],[1,1]]

}

Solución (memory exceeded error)

```python
import heapq

class Solution:
    def kSmallestPairs(self, nums1: List[int], nums2: List[int], k: int) -> List[List[int]]:
        if not nums1 or not nums2:
            return []

        min_heap = []
        for i in range(len(nums1)):
            for j in range(len(nums2)):
                heapq.heappush(min_heap, (nums1[i] + nums2[j], i, j))

        result = []
        while min_heap and len(result) < k:
            sum_val, i, j = heapq.heappop(min_heap)
            result.append([nums1[i], nums2[j]])

        return result
```

problemas.py

workshop.java

Problema: ‘Find K Pairs with Smallest Sums’

{

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3

(1,2) (1,4) (1,6)

}

Problema: ‘Find K Pairs with Smallest Sums’

{

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3

(1,2) (1,4) (1,6)

}

Problema: ‘Find K Pairs with Smallest Sums’

{

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3

(1,2) (1,4) (1,6)

(7,2)

}

Problema: ‘Find K Pairs with Smallest Sums’

{

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3

(1,2) (1,4) (1,6) (7,2)

}

Problema: ‘Find K Pairs with Smallest Sums’

{

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3

(1,2) (1,4) (1,6) (7,2)

(7,4)

}

Problema: ‘Find K Pairs with Smallest Sums’

{

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3

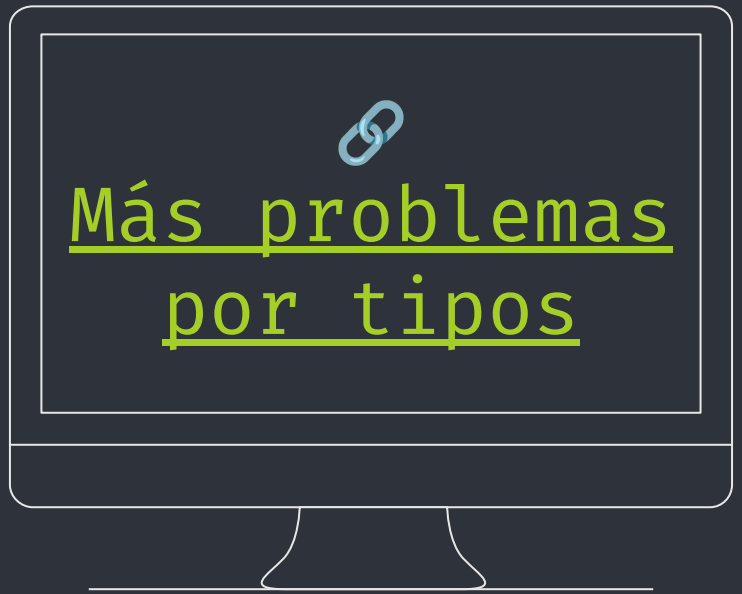
(1,2) (1,4) (1,6) (7,2) (7,4)

}

Solución

```python
import heapq

class Solution:
    def kSmallestPairs(self, nums1: List[int], nums2: List[int], k: int) -> List[List[int]]:
        if not nums1 or not nums2:
            return []

        min_heap = []
        for j in range(min(len(nums2), k)):
            heapq.heappush(min_heap, (nums1[0] + nums2[j], 0, j))

        result = []
        while min_heap and len(result) < k:
            sum_val, i, j = heapq.heappop(min_heap)
            result.append([nums1[i], nums2[j]])

            if i + 1 < len(nums1):
                heapq.heappush(min_heap, (nums1[i + 1] + nums2[j], i + 1, j))

        return result
```

LeetCode - Extra



¡Gracias!

¿Nos vemos la semana que viene?

- Última sesión el 31: Problemas variados
  - Nos tenéis que confirmar que venís XD
- ¿Quién se viene a escalar?