Colas de prioridad

Notas de clase

Estructuras de datos y algoritmos

Facultad TIC - UPB

Spring 2025

Jorge Mario Londoño Peláez & Varias AI

July 7, 2025

Descripción de la unidad

Chapter Description

Contents

1 Definición, Aplicaciones

Una **cola de prioridad** es un Abstract Data Type (ADT) que almacena elementos con prioridades asociadas. A diferencia de las colas (FIFO) o pilas (LIFO), el orden de extracción se basa en la prioridad, no en el orden de llegada.

En una **MaxPQ** (cola de máxima prioridad), el elemento con la mayor prioridad se elimina primero.

En una **MinPQ** (cola de mínima prioridad), el elemento con la menor prioridad se elimina primero.

La prioridad determina el orden en que se procesan los elementos.

Ejemplos de aplicación de la cola de prioridad

- **Gestión de tareas en un sistema operativo:** Se usan para priorizar la ejecución de procesos, dando preferencia a los más importantes. Por ejemplo, un proceso del sistema podría tener mayor prioridad que un programa de usuario.
- **Algoritmo de Dijkstra (camino más corto):** Se usa una **MinPQ** para seleccionar el nodo más cercano al origen que aún no se ha visitado.
- **Compresión de datos (algoritmo de Huffman):** Se usa una **MinPQ** para construir el árbol de Huffman, uniendo los nodos con menor frecuencia.
- **Simulación de eventos discretos:** Se usa una PQ para gestionar la secuencia de eventos, procesando primero los eventos con el tiempo de ocurrencia más próximo. Por ejemplo, simular el tráfico en una ciudad.

Ejemplo sencillo: Imagina una sala de emergencias en un hospital. Los pacientes no se atienden en el orden en que llegan, sino según la gravedad de su condición. Los pacientes con problemas más graves (mayor prioridad) se atienden primero.

2 API de cola de prioridad

La API básica de una cola de prioridad (asumiendo una MaxPQ) incluye las siguientes operaciones:

- `insert(key)`: Inserta un nuevo elemento con clave `key` en la cola de prioridad.
- `max()`: Retorna el elemento con la clave máxima (mayor prioridad) en la cola de prioridad, sin eliminarlo.
- `delMax()`: Elimina y retorna el elemento con la clave máxima en la cola de prioridad.
- `changeKey(element, key)`: Cambia la clave del `element` a `key`.
- `size()`: Retorna el número de elementos en la cola de prioridad.
- `isEmpty()`: Retorna verdadero si la cola de prioridad está vacía, falso en caso contrario.

En el caso de la cola de mínima prioridad (MinPQ), las operaciones son similares, pero se nombran `min()` y `delMin()`.

3 Árboles binarios completos y montículos

3.1 Definición y propiedades del árbol binario completo

Un **árbol binario completo** es un árbol binario en el que todos los niveles, excepto posiblemente el último (el de las hojas), están completamente llenos, y todos los nodos en el último nivel están lo más a la izquierda posible.

Propiedades:

- Un árbol binario completo con n nodos tiene una altura $h = \lfloor \lg n \rfloor$ [1].
- El número de nodos en un árbol binario completo de altura h está entre 2^h y $2^{h+1} - 1$.

3.2 Montículo

Un **montículo** (o *heap* en inglés) es un árbol binario completo que cumple la **propiedad del montículo**:

- **MaxPQ:** La clave de cada nodo es mayor o igual que las claves de sus hijos. La raíz tiene la clave más grande.
- **MinPQ:** La clave de cada nodo es menor o igual que las claves de sus hijos. La raíz tiene la clave más pequeña.

Los montículos se suelen representar como arreglos para facilitar el cálculo de los índices de los hijos. Si un nodo está en la posición k, sus hijos estarán en las posiciones $2k$ y $2k + 1$ (asumiendo que el arreglo empieza en 1). Asimismo, si un nodo esta en la posición j, si padre está en la posición $j \div 2$ [2]. Se prefiere la representación por medio de arreglo porque resulta más eficiente en términos de espacio (vs. usar objetos Nodo que tienen un overhead adicional).

3.3 Operaciones auxiliares: `swim` y `sink`

`swim` y `sink` son cruciales para mantener la propiedad del montículo tras insertar o eliminar elementos.

- `swim(k)` (Flotar): Si la clave del nodo en la posición `k` es mayor que la de su padre, se intercambia con el padre. Se repite hasta que la clave del nodo no sea mayor que la de su padre o hasta llegar a la raíz. Se usa tras la inserción.
- `sink(k)` (Hundir): Si la clave del nodo en la posición `k` es menor que la de alguno de sus hijos, se intercambia con el hijo mayor (en un MaxPQ). Se repite hasta que la clave del nodo no sea menor que la de sus hijos o hasta llegar a una hoja. Se usa tras la eliminación.

Pseudocódigo de `swim(k)`:

procedure SWIM(A, k)
    while $k > 1$ **and** $A[k] > A[k \div 2]$ **do**
        Intercambiar $A[k]$ y $A[k \div 2]$
        $k \leftarrow k \div 2$
    end while
end procedure

[1] La convención $\lg n = \log_2 n$

[2] Se utiliza $\div$ para denotar la division entera, es decir $j \div 2 = \lfloor j/2 \rfloor$

Pseudocódigo de `sink(k)`:

```
procedure SINK(A, k, N)
    while 2k ≤ N do
        j ← 2k
        if j < N and A[j] < A[j + 1] then
            j ← j + 1
        end if
        if A[k] ≥ A[j] then
            break
        end if
        Intercambiar A[k] y A[j]
        k ← j
    end while
end procedure
```

Complejidad: Ambas operaciones tienen complejidad $O(\log n)$ porque, en el peor caso, deben recorrer la altura del árbol.

3.4 Operaciones de la PQ con montículos

Las operaciones de la PQ se implementan eficientemente con montículos binarios completos (representados como arreglos):

- `insert(key)`:
  1. Añade el nuevo elemento al final del arreglo.
  2. Llama a `swim(n)`, donde n es la posición del nuevo elemento.

  Complejidad: $O(\log n)$ por la operación `swim`.

- `max()`:
  1. Devuelve el elemento en la primera posición del arreglo (la raíz del montículo).

  Complejidad: $O(1)$.

- `delMax()`:
  1. Intercambia la raíz (el máximo) con el último elemento del arreglo.
  2. Elimina el último elemento del arreglo (que ahora contiene el máximo).
  3. Llama a `sink(1)` para restaurar la propiedad del montículo desde la raíz.

  Complejidad: $O(\log n)$ por la operación `sink`.

4 Ordenación por montículo: Heapsort

Heapsort es un algoritmo de ordenación que usa montículos. Tiene dos fases:

1. **Construcción del montículo (Heapify):** Transforma el arreglo de entrada en un montículo.
2. **Fase de ordenación:** Extrae repetidamente el elemento máximo del montículo y lo coloca al final del arreglo ordenado.

Construcción del montículo a partir de un arreglo (Heapify):

Se puede transformar un arreglo en un montículo en tiempo lineal $O(n)$ usando el método *heapify*. Se trata el arreglo como un árbol binario completo y se aplica `sink` a todos los nodos no hoja, desde el último hasta la raíz.

Pseudocódigo de `Heapify(A,N)`:

```
procedure HEAPIFY(A, N)
    for k ← N ÷ 2 downto 1 do SINK(A, k, N)
    end for
end procedure
```

El algoritmo `Heapify` tiene una complejidad $O(n)$.

Algoritmo Heapsort:

1. Construye un MaxPQ a partir del arreglo de entrada usando *heapify*.
2. Repite hasta que el montículo esté vacío:
   (a) Intercambia la raíz del montículo (el elemento máximo) con el último elemento del montículo.
   (b) Reduce el tamaño del montículo en 1.
   (c) Aplica `sink(1)` a la raíz para restaurar la propiedad del montículo.

Heapsort tiene una complejidad de $O(n \log n)$ en el peor, promedio y mejor caso. Es un algoritmo *in-situ* (no requiere memoria adicional).

Bibliografía

[1] Aditya Bhargava. *Grokking Algorithms.* Manning Publications, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[3] Narasimha Karumanchi. *Data Structures and Algorithms Made Easy.* CareerMonk Publications, 2011.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design.* Pearson, 2005.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms.* Addison-Wesley Professional, 4th edition, 2011.