

Colas de prioridad

Montículos

Heaps

Idea general

- Existen aplicaciones donde es necesario encontrar el mayor (o el menor) elemento de un conjunto.
- No es eficiente mantener todos los elementos en orden. Es necesario agregar o eliminar elementos con frecuencia.
- La cola de prioridad captura estos requerimientos. Su interfaz ofrece métodos para remover el mayor elemento y agregar elementos.

Aplicaciones de la cola de prioridad

- Atender/despachar eventos en orden de su prioridad.
- Planificación de tareas en sistemas operativos.
- Manejo de eventos en simuladores.
- Ordenación por montículo.

Definición del API

```
public class MaxPQ<Key extends Comparable<Key>>
```

	MaxPQ()	Crear cola de prioridad
	MaxPQ(int max)	Crear cola de prioridad de tamaño fijo
	MaxPQ(Key[] a)	Crear cola de prioridad a partir de un arreglo de llaves
void	insert(Key v)	Agregar una llave
Key	max()	Obtener la mayor llave
Key	delMax()	Borrar la mayor llave y devolverla
boolean	isEmpty()	Esta vacía la cola?
int	size()	Número de elementos en la cola

Cola de mínima prioridad

- Simplemente se implementa la política opuesta: Las operaciones `max()` y `delMax()` se reemplazan por operación `min()` y `delMin()` para obtener el mínimo y borrar el mínimo elemento de la cola respectivamente.
- Solo requiere cambios menores en la implementación: Invertir orden de comparaciones.
- Se denomina MinPQ.

Ejemplo de aplicación

- Se tiene la lista de todas las canciones (N) con su votación.
- Determinar las $M=10$ canciones más populares.
- Sea N el número de canciones y M el número de elementos a seleccionar.

Soluciones sencillas

- Ordenar la lista tomando como llave la votación y tomar los primeros M elementos

$$\sim N \log(N) + M$$

- Mantener los M mayores elementos en un arreglo no ordenado. Al procesar cada elemento compararlo con los M actuales. Si el nuevo elemento es mayor que el mínimo, reemplazar el mínimo.

$$\sim NM$$

Montículo – *Binary heap*

- Las soluciones previas no son muy eficientes.
- Una idea para una implementación eficiente es utilizar un árbol tal que todo nodo padre es mayor o igual a sus nodos hijos.
- Esta propiedad se denomina propiedad del montículo.
- Obtener el mayor elemento es simplemente obtener la raíz del árbol.
- Pero hay que considerar también como agregar, eliminar o cambiar nodos del árbol.

Representación mediante árbol binario completo

- Una representación estándar de árbol binario utiliza 3 referencias por nodo.
- Existe una solución más sencilla para cuando el árbol binario es completo. Se dice que el árbol es completo cuando empezando en la raíz y recorriendo todos los niveles sucesivos de izquierda a derecha, todos los niveles están llenos.
- Es posible representar un árbol binario completo en un arreglo.

Montículo / *Heap*

- Es una colección de llaves que cumplen la propiedad del montículo y representadas en un árbol binario completo.
- Ejemplo
- Se observa que en el arreglo* (y para el árbol completo) los nodos hijos del nodo i se encuentran en las posiciones $2i$ y $2i+1$.
- El nodo padre de un nodo j se encuentra en la posición $j \div 2$.

* El arreglo se indexa con posiciones $[1..N]$ para facilitar la aritmética.

Altura de árboles binarios completos

- Proposición: La altura de un árbol binario completo de N nodos es $\lfloor \lg(N) \rfloor$
- Dem: Por inducción matemática en N

Implementación del montículo

- Se utiliza un arreglo $pq[1..N]$ (la posición 0 simplemente no se usa).
- Las operaciones básicas que se requieren son:
 - eliminar la raíz
 - agregar una llave
 - cambiar el valor de una llave
- Cualquiera de estas operaciones puede invalidar la propiedad del montículo. Se restaura intercambiando nodos del árbol.

Operaciones sink/swim

- Si una llave se hace mayor, puede superar el valor de su padre y por tanto se debe intercambiar hasta que su antecesor sea mayor o igual: Operación *swim* (flotar).
- Si una llave se hace menor, puede pasar a ser inferior que sus hijos. Se intercambia con el mayor de los hijos, hasta que no se encuentren hijos mayores: Operación *sink* (hundir).

Implementación de las operaciones swim / sink

```
Key[] pq;  
private int n;  
  
private void swim(int k) {  
    while(k>1 && less(k/2,k)) {  
        exch(k/2,k);  
        k /= 2;  
    }  
}  
  
private void sink(int k) {  
    while (2*k <= N) {  
        int j = 2*k;  
        if (j < N && less(j, j+1)) j++;  
        if (!less(k, j)) break;  
        exch(k, j);  
        k = j;  
    }  
}
```

Implementación de otras operaciones

```
public void insert(Key v) {  
    pq[++N] = v;  
    swim(N);  
}
```

```
public Key delMax() {  
    Key max = pq[1];  
    exch(1, N--);  
    pq[N+1] = null;  
    sink(1);  
    return max;  
}
```

Implementación completa

Eficiencia de las operaciones del montículo

Proposición

- El algoritmo `insert` requiere como máximo $\lg(N)$ comparaciones.
- El algoritmo `deletMax` requiere como máximo $2\lg(N)$ comparaciones.

Dem

- `insert` agrega el elemento al final y hace una operación `swim`, la cual compara como máximo con todos los nodos padre en la ruta a la raíz.
- `deletMax` reemplaza la raíz y hace una operación `sink`, la cual requiere 2 comparaciones por nivel del árbol.

Ejemplo

Problema de seleccionar los M mayores

- Utilizar un montículo invertido (MinPQ) de tamaño constante $M+1$.
- Para cada uno de los N elementos se invoca `insert` y se elimina el menor elemento (raíz).
- Observar que al final de cada iteración la cola contiene los M mayores elementos hasta el momento.
- Eficiencia: comparaciones $\sim N \lg(M)$, espacio $\sim M$.

Ordenación por montículo

Heapsort

- Idea general: La operación delMax() retorna el mayor valor y lo elimina.
- Iterar obteniendo el mayor valor y guardandolo en una pila. Al final todos los elementos están en orden.
- Se puede ahorrar el espacio adicional, ubicando los elementos en las últimas posiciones del mismo vector.

Creación del montículo

- Esta idea requiere que los datos a ordenar estén en un montículo.
- Una forma eficiente de convertir un vector en un montículo es recorriéndolo de derecha a izquierda e invocar la operación sink en cada nodo. Esto efectivamente crea un subheap en cada nodo.
- Al llegar a la raíz el arreglo es un montículo.

Creación de montículo

- Proposición: La creación del montículo utilizando el proceso de derecha a izquierda y la operación sink requiere un número de comparaciones $\sim 2N$ y un número de intercambios $\sim N$.

Crear montículo: Demostración $\sim N$

$$\begin{aligned}C(h) &= \sum_{p=0}^{h-1} 2^p (h-p) \\C(h-1) &= \sum_{p=0}^{h-2} 2^p (h-1-p) \\&= \sum_{p=0}^{h-2} 2^p (h-p) - \sum_{p=0}^{h-2} 2^p \\&= \sum_{p=0}^{h-2} 2^p (h-p) - (2^{h-1} - 1)\end{aligned}$$

Tomando la diferencia

$$C(h) - C(h-1) = 2^{h-1} (h - (h-1)) + (2^{h-1} - 1)$$

y por multiples sustituciones

$$\begin{aligned}C(h) &= C(h-1) + (2^h - 1) \\C(h-1) &= C(h-2) + (2^{h-1} - 1) \\&\vdots\end{aligned}$$

$$C(1) = C(0) + (2^1 - 1) \quad , \text{ con } C(0) = 0$$

y sustituyendo hacia atras

$$C(2) = C(1) + (2^2 - 1) = (2^1 - 1) + (2^2 - 1)$$

\vdots

$$\begin{aligned}C(h) &= \sum_{k=1}^h (2^k - 1) \\&= \sum_{k=1}^h 2^k - \sum_{k=1}^h 1 \\&= 2^{h+1} - 1 - h\end{aligned}$$

por la relacion entre la altura y los nodos

$$N = 2^{h+1} - 1$$

y sustituyendo

$$C(N) = N - \lg(N + 1)$$

Algoritmo de ordenación por montículo (*heapsort*)

```
public static void sort(Comparable[] pq) {  
    int N = pq.length;  
    for (int k = N/2; k >= 1; k--)  
        sink(pq, k, N);  
    while (N > 1) {  
        exch(pq, 1, N--);  
        sink(pq, 1, N);  
    }  
}
```

[Código fuente completo](#)

Eficiencia de *heapsort*

- El primer ciclo for es la creación de montículo que requiere $\sim 2N$ comparaciones y $\sim N$ intercambios.
- El segundo ciclo hace $N-1$ intercambios y N operaciones sink. Cada operación sink hace como máximo $\sim \lg(N)$ intercambios y comparaciones.
- Ordenación heapsort tiene orden de crecimiento:

$$\sim N \lg(N)$$