Aula 05: 28/03/2017

Tópico

- Filas priorizadas

Filas priorizadas

Leitura: Filas priorizadas, Paulo feofiloff, Priority queues, S&W

Vídeo: Priority queues, Sedgewick

Uma **fila priorizada** (ou **fila com prioridades**) é um ADT (*abstract data type*) que generaliza tanto a fila quanto a pilha.

PQ de máximo

Uma fila priorizada decrescente ou **PQ de máximo** é um ADT que manipula um conjunto de itens por meio de duas operações fundamentais:

- inserção de um novo item no conjunto e
- remoção de um item máximo.

Isso significa que uma fila priorizada manipula itens comparáveis.

API

```
public class MaxPQ<Item extends Comparable<Item>>
-------------------------------------------------
       MaxPQ()   cria  uma PQ de máximo
       MaxPQ(int cap)  cria uma PQ de máximo com capacidade cap
       MaxPQ(Item[] a) cria uma PQ de máximo com os itens que estão em a[]
  void insert(Item v)  insere o item v nesta PQ
  Item max()           devolve um item máximo deste PQ
  Item delMax()        remove e devolve um item máximo desta PQ
boolean isEmpty()      esta PQ está vazia?
   int size()          número de itens desta PQ
```

A expressão `Item extends Comparable<Item>` na definição da API indica que o tipo genérico `Item` deve satisfazer a interface Comparable, ou seja, deve ter um método `compareTo()` para comparamos objetos do tipo `Item`. Por exemplo, os tipos referência `Integer`, `Double` e `String` implementam a interface `Comparable`. mas não é o caso dos tipos primitivos int e double.

```
Welcome to DrJava.  Working directory is /home/coelho/mac0323/
> String r = "Como "
> String s = "é "
> String t = "bom "
> String u = "estudar "
> String v = "MAC0323!"
```

```
> String w = "estudar "
> r + s + t + u + v
"Como é bom estudar MAC0323!"
> r.compareTo(s)
-166
> r.compareTo(t)
-31
> r.compareTo(s) # < 0 significa r < s
-166
> r.compareTo(t) # < 0 significa r < t
-31
> r.compareTo(u) # < 0 significa r < u
-34
> r.compareTo(v) # < 0 significa r < v
-10
> u.compareTo(t) # > 0 significa u > t
3
> u.compareTo(v) # > 0 significa u > v
24
> u.compareTo(w) # == 0 significa u == w
0
>
```

Cliente

Exibe as 10 maiores transações. O que define que uma transação é maior que o outra é o seu valor.

```
> java TopM 10 < transactions.txt
    Thompson    2/27/2000  4747.08
    Tarjan      2/12/1994  4732.35
    Tarjan      1/11/1999  4409.74
    Hoare       8/18/1992  4381.21
    Tarjan      3/26/2002  4121.85
    Hoare       2/10/2005  4050.20
    Turing     10/12/1993  3532.36
    Knuth      11/11/2008  3284.33
    Hoare       5/10/1993  3229.27
    Dijkstra    8/22/2007  2678.40
```

```
public class TopM {

   public static void main(String[] args) {
      int m = Integer.parseInt(args[0]);

      MinPQ<Transaction> pq = new MinPQ<Transaction>(M + 1);

      while (StdIn.hasNextLine()) {
         pq.insert(new Transaction(StdIn.readLine()));
         if (pq.size() > m)
            pq.delMin();
```

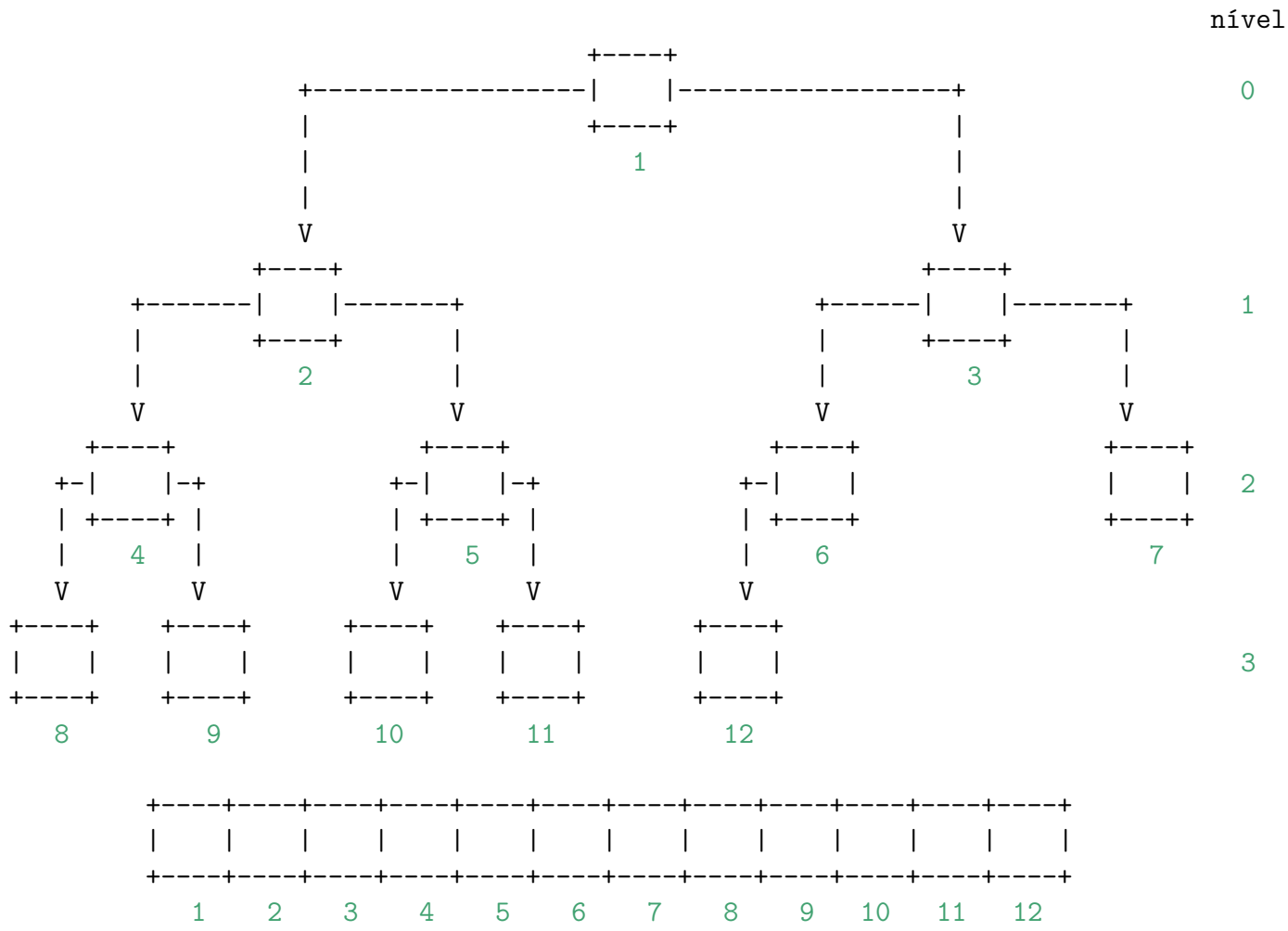
```
      }

      // As M maiores transações estão na pq.
      Stack<Transaction> stack = new Stack<Transaction>();
      while (!pq.isEmpty())
         stack.push(pq.delMin());
      for (Transaction t : stack)  // foreach
         StdOut.println(t);
   }
}
```

A classe `Transaction` possui um método `compareTo()`

```
public class Transaction implements Comparable<Transaction> {

   private final String name;
   private final String date;
   private final double value;

   // Construtor. Cria uma transação a partir da string tr.
   public Transaction(String tr) {
      . . .
   }

   // Compara esta transação com a transação that.
   public int compareTo(Transaction that) {
      if (this.value > that.value) return +1;
      if (this.value < that.value) return -1;
      return 0;
   }
}
```

PQ de máximo implementada em um heap decrescente

Árvores em vetores



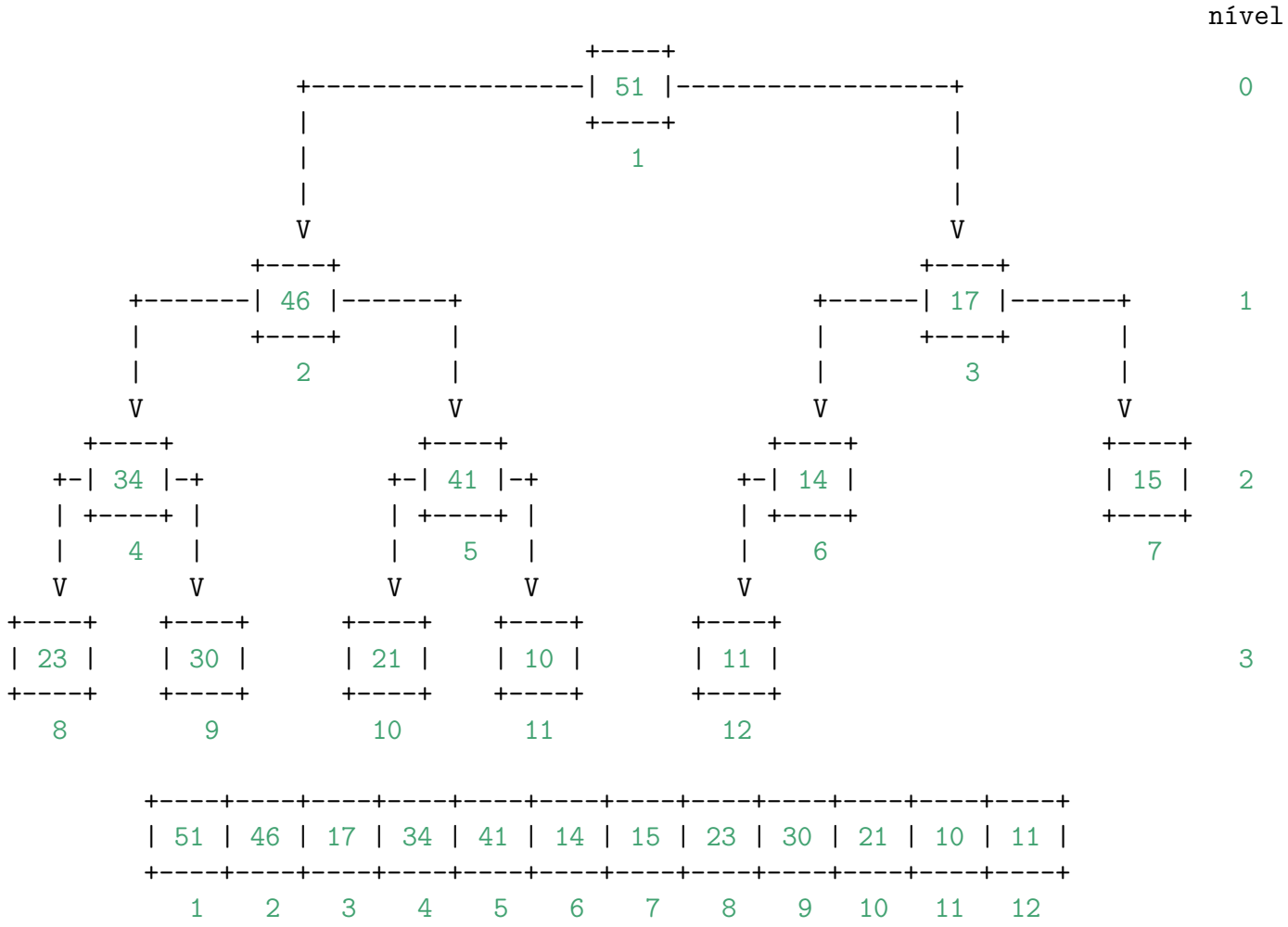
- filho esquerdo de `p`: `2*p`
- filho direito de `p`: `2*p + 1`
- pai de `f`: `f/2`
- nível da raiz: 0
- nível de um no `i`: $\sim \lg i$
- altura da raiz: $\sim \lg n$
- altura de um nó `i`: $\sim \lg(n/i)$
- altura da folha: 0
- número de nós de altura `h`: $\sim n/2^{h+1}$

Max-heap

Um vetor `pq` é um *max-heap* se

```
pq[i/2].compareTo(pq[i]) >= 0 # pq[i/2] >= pq[i]
```

para `i = 2,3,...,n`.



Implementação

```
public class MaxPQ<Item extends Comparable<Item>> {

   private Item[] pq;
   private int n = 0; // heap fica em pq[1..n]

   public MaxPQ(int maxN) { // construtor
      pq = (Item[]) new Comparable[maxN+1];
   }

   public boolean isEmpty() {
      return n == 0;
   }
```

```
public int size() {
   return n;
}

public void insert(Item v) {
   pq[++n] = v;
   swim(n);
}

public Item delMax() {
   Item max = pq[1];
   exch(1, n--);
   pq[N+1] = null;  // avoid loitering
   sink(1);
   return max;
}

/** métodos de manutenção do max-heap
  */
private void swim(int k) {
   while (k > 1 && less(k/2, k)) {
      exch(k/2, k);
      k = k/2;
   }
}

// versão com as variáveis p para "pai" e f para "filho"
private void swim(int f) {
   int p = f/2;
   while (p > 0 && less(p, f)) {
      exch(p, f);
      f = p; p = f/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;
   }
}

// versão com as variáveis p para "pai" e f para "filho"
private void sink(int p) {
   int f = 2*k;
   while (f <= n) {
```

```
            if (f < n && less(f, f+1)) f++;
            if (!less(p, f)) break;
            exch(p, f);
            p = f; f = 2*p;
        }
    }


    /** métodos de acesso aos itens
      */
    private boolean less(int i, int j) {
        return pq[i].compareTo(pq[j]) < 0;
    }

    private void exch(int i, int j) {
        Item t = pq[i];
        pq[i] = pq[j];
        pq[j] = t;
    }

}
```

Proposição. A altura de um heap com `n` nós é $\lceil \lg n \rceil$.

Proposição. Em uma fila priorizada com `n` itens implementada com um heap,

- cada inserção faz no máximo $1 + \lg n$ comparações e
- cada remoção do máximo faz no máximo $2 \lg n$ comparações.

Comentários

Multiway heaps. Alterar a representação para termos árvores ternárias. Existe um balanço entre a aridade do heap e o consumo de tempo. Quanto maior o grau, menor a altura, mas a consulta aos filhos de um nó consome mais tempo, pois cada nó tem mais filhos.

Redimensionamente. Podemos utilizar a política de redimensionamento na implementação do heap.

Imutabilidade dos Itens. A nossa implementação mantém referência a objetos criados pelo cliente. Esses objetos podem ser alterados e a consistência do fila priorizada vai para o espaço... Alternativamente, é possível manter clones dos objetos.