Aula 06: 22/03/2018

Tópico

- Leftist heaps

Leitura: *The Art of Computer Programming: Sorting and Search*, D.E.Knuth. Seção 5.2.3

Leftist heaps

Leftist heaps da mesmo forma que os mais modernos *binomial heaps* e *Fibonacci heaps*, são estruturas de dados que realizam a operação de `union()` de heaps eficientemente. Hmm, deve ter sido um dos primeiro, ou talvez o primeiro dos chamados *mergeable heaps*.

Cliente

Exibe as menores transações

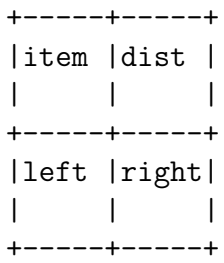
```
meu_prompt> java BottomM 10 < transactions.txt
Turing      1/11/2002    66.10
Knuth       6/14/1999   288.34
Turing      6/17/1990   644.08
Dijkstra    9/10/2000   708.95
Dijkstra   11/18/1995   837.42
Hoare       8/12/2003  1025.70
Bellman    10/26/2007  1358.62
Knuth       7/25/2008  1564.55
Turing      2/11/1991  2156.86
Tarjan     10/13/1993  2520.97
```

```
public class BottomM {
   public static void main(String[] args) {
      int M = Integer.parseInt(args[0]);
      MaxLeftist<Transaction> pq = new MaxLeftist<Transaction>();
      while (StdIn.hasNextLine()) {
         pq.insert(new Transaction(StdIn.readLine()));
         if (pq.size() > M)
            pq.delMax();
      }

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

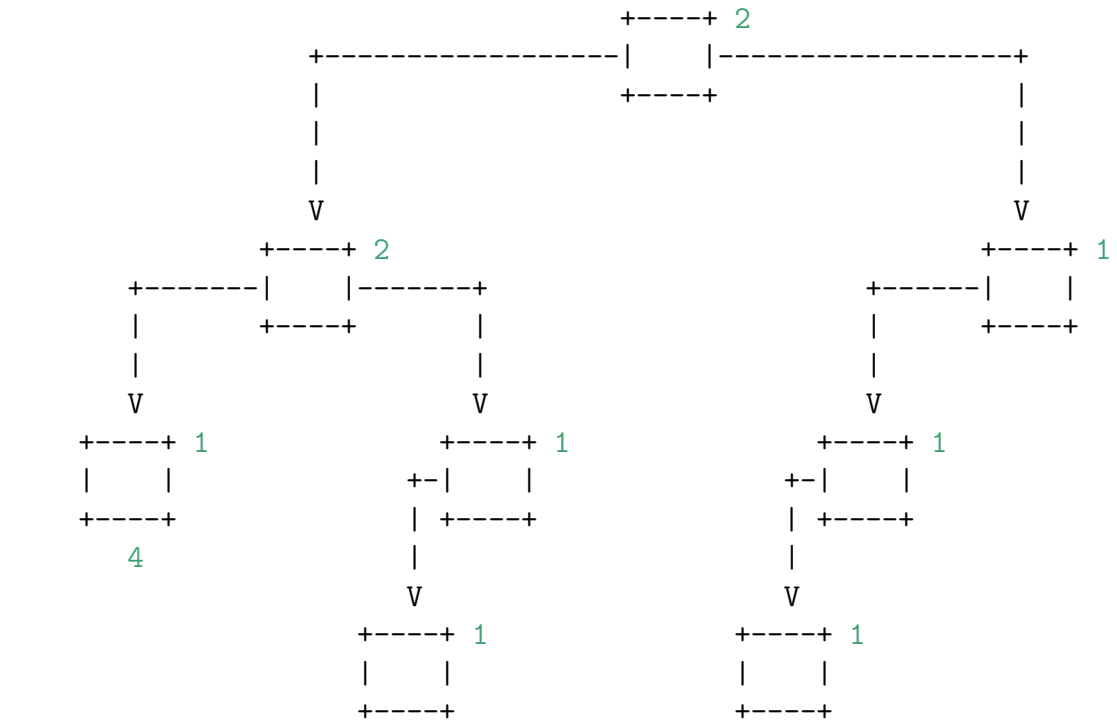
Árvores esquerdistas

Cada nó de uma árvore esquerdista (*Leftist tree*) terá quatro campos:




```
private class Node {
    private Item item;
    private Node left, right;
    private int dist;
    [...]
}
```

```
r.dist = 0,                                  se r == null
r.dits = 1 + min{r.left.dist, r.right.dist}, se r != null
```

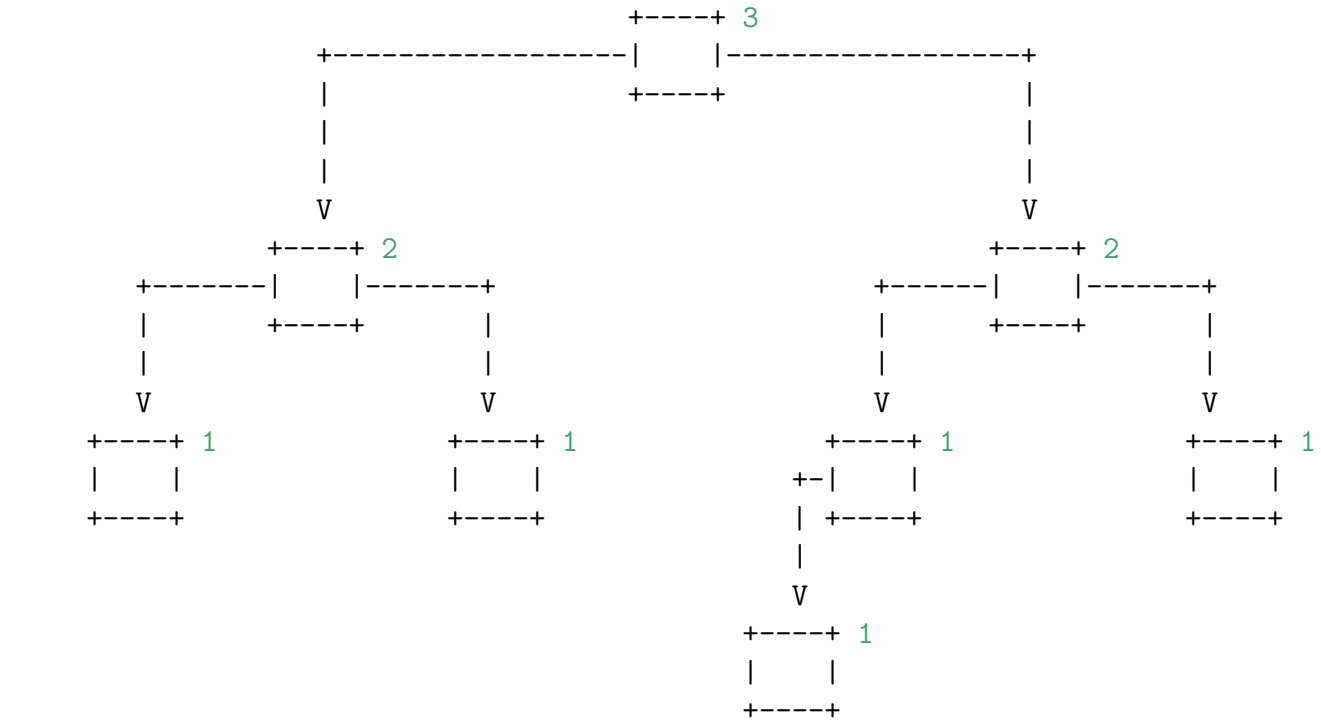


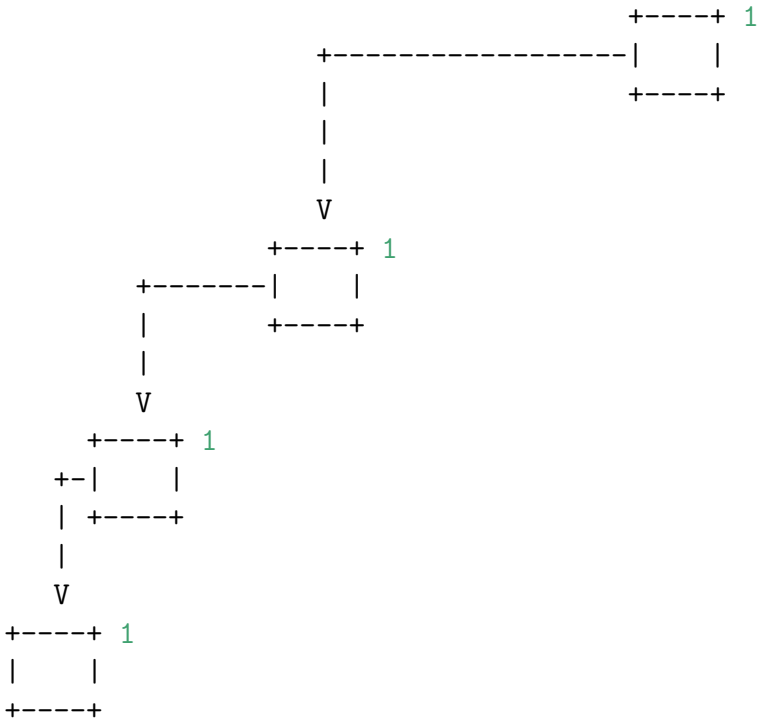
Uma árvore é **esquerdista** se

```
r.left.dist >= r.right.dist
```

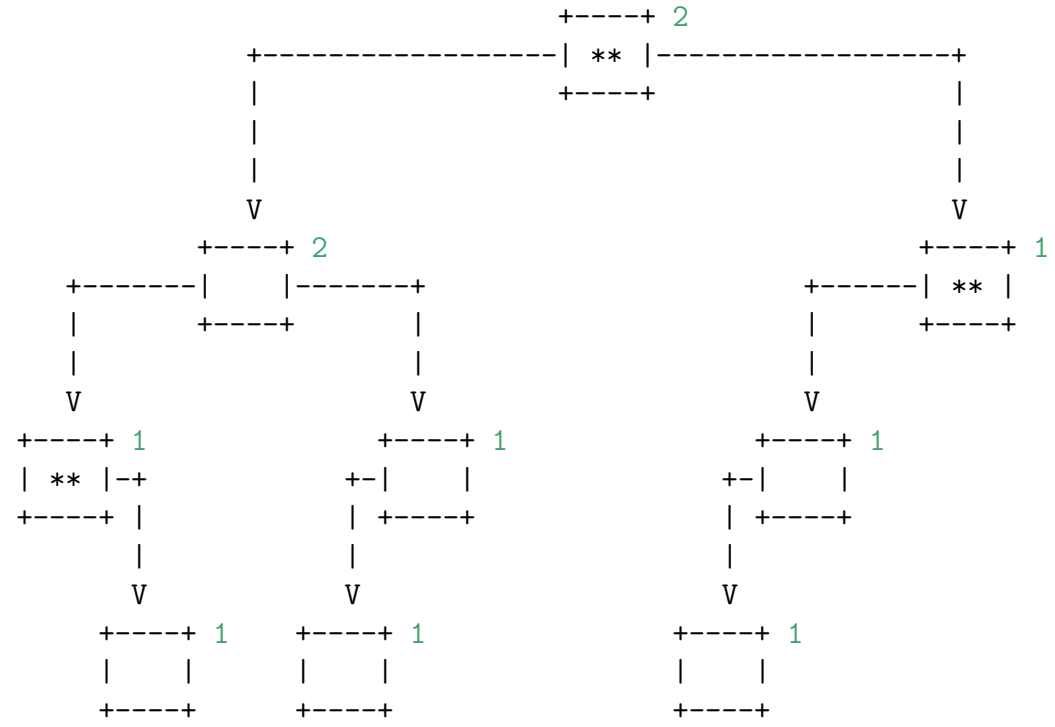
para todo nó `r`.

Exemplos:





Árvore não esquerdista:



** = nós que violam a condição esquerdista

Caminho direitista

O **caminho direitista** a partir de um nó `r` é o caminho

```
r -> r.right -> r.right.right -> r.right.right.right -> ... -> null
```

Em uma árvore esquerdista vale que

```
r.dist = r.right.dist + 1
```

para todo nó `r`. Ou seja, `r.dist` é o comprimento do caminho direitista a partir de `r`.

Fato

Se `r` é um nó de uma árvore esquerdista com `n` nós na subárvore que o tem como raiz, então

$$n \geq 2^{r.dist} - 1.$$

Demonstração. Por indução em $d =$ `r.dist`.

Se $d = 1$, então `n` $\geq 1 = 2^d - 1$.

Suponha que $d \geq 2$ e que a desigualdade vale para $d - 1$.

Temos que `r.right.dist` $= d - 1$, e por indução a subárvore direita tem `r` tem pelo menos $2^{d-1} - 1$ nós.

Temos que `r.left.dist` $\geq$ `r.right.dist` $= d - 1$. Logo, `r.left` possui um nó com `dist` $= d - 1$ e, portanto, possui pelo menos $2^{d-1} - 1$ nós.

Portanto, a subárvore de raiz r tem pelo menos

$$2^{d-1} - 1 + 2^{d-1} - 1 + 1 = 2^d - 1.$$

Consequência

Se `r` é um nó de uma árvore esquerdista com `n` nós, então `r.dist` $\leq \lg n$.

Heaps esquerdistas (Leftist heaps)

Um **heap esquerdista** (= *leftist heap*) é uma árvore esquerdista tal que

```
r.item.compareTo(r.left.item) <= 0
r.item.compareTo(r.right.item) <= 0
```

para todo nó `r` tal que `r.left != null` para a primeira desigualdade fazer sentido e `r.right != null` para a segunda desigualdade fazer sentido

Merge

A operação básica de um heap esquerdista é `merge(r1, r2)` de duas árvores esquerdista `r1` e `r2`. O consumo de tempo dessa operação é proporcional a soma dos comprimentos dos caminhos direitistas de `r1` e `r2` e portanto será proporcional a $\lg n$, onde `n` é o número de nós na árvore resultante.

No `merge(r1, r2)` fazemos *essencialmente* a intercalação das listas ligadas formadas pelos caminhos direitistas de `r1` e `r2`:

```
Node merge(Node r1, Node r2) {
    if (r1 == null) return r2;
    if (r2 == null) return r1;
    if (r1.item.compareTo(r2.item) < 0) {
        Node t = r1;  r1 = r2; r2 = t;
    }
    r1.right = merge(r1.right, r2);
}
```

O *essencialmente* é devido ao fato de ser necessário acertamos os campos `dist` durante a volta da recursão.

As operações `insert()` e `delMax()` de um heap esquerdista de apoiam na operação `merge()`.

Implementação

```
public class MaxLeftist<Item extends Comparable<Item>> {
    private Node root;
    private int n;

    private class Node {
        private Item item;
        private Node left, right;
        private int dist;

        public Node(Item item, Node left, Node right, int dist) {
            this.item = item;
            this.left = left;
            this.right = right;
            this.dist = dist;
        }

        public String toString() {
            String s = "{Item: " + item + "(" + left + "),[" +
                right + "] " + dist + "}";
            return s;
        }
    }

    /**
     * Initializes an empty priority queue.
     */
    public MaxLeftist() { }

    public boolean isEmpty() {
        return root == null;
    }

    public int size() {
        return n;
    }

    public void insert(Item item) {
        Node s = new Node(item, null, null, 1);
        root = merge(root, s);
        n++;
    }

    public Item delMax() {
        Item max = root.item;
        root = merge(root.left, root.right);
        n--;
        return max;
    }
```

```
    /**
     * transforms this into the union of this and that.
     * detona com that.
     */
    public void union(MaxLeftist<Item> that) {
        if (that == null) return ;
        this.root = merge(this.root, that.root);
        this.n += that.n;
    }

    // método administrativo para manter a consistência da Leftist heap
    private Node merge(Node r1, Node r2) {
        if (r1 == null) return r2;
        if (r2 == null) return r1;

        // r1 != null e r2 != null
        if (less(r1, r2)) {
            Node tmp = r1; r1 = r2; r2 = tmp;
        }

        // r1 aponta para o maior item
        if (r1.left == null) r1.left = r2;
        else {
            r1.right = merge(r1.right, r2);
            if (r1.left.dist < r1.right.dist) {
                // exchange left and right
                Node t = r1.left;
                r1.left = r1.right;
                r1.right = t;
            }
            r1.dist = r1.right.dist + 1;
        }
        return r1;
    }

    private boolean less(Node r, Node s) {
        return r.item.compareTo(s.item) < 0;
    }
}
```