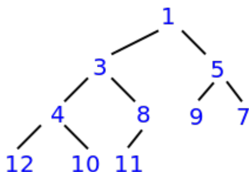
Filas de Prioridade e Heaps

Pedro Ribeiro

DCC/FCUP

2020/2021



1	2	3	4	5	6	7	8	9	10
1	3	5	4	8	9	7	12	10	11

Filas de prioridade - Motivação

- As urgências de um hospital funcionam com um sistema conhecido como **Triagem de Manchester**, que permite classificar a gravidade da situação de cada paciente, atribuindo-lhe uma das seguintes cores:

EMERGENTE	VERMELHO
MUITO URGENTE	LARANJA
URGENTE	AMARELO
POUCO URGENTE	VERDE
NÃO URGENTE	AZUL

- A ordem que os pacientes são atendidos dependa da sua **prioridade**. Por exemplo, um doente vermelho que chegue depois é sempre atendido antes de qualquer doente verde ou azul, mesmo que estes estejam nas urgências à espera há muito tempo

Filas de prioridade - Definição

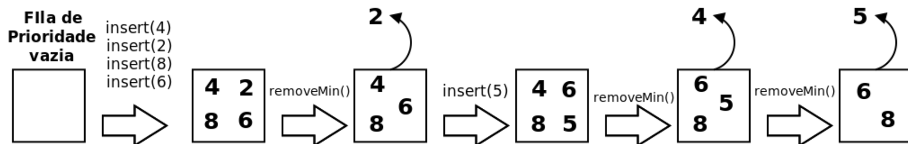
- Os TADs que conhecemos dependem apenas da **ordem de chegada** e não se ajustam (diretamente) a um processo como este:
  - Uma **pilha** é sempre LIFO (sai o último elemento que entrou)
  - Uma **fila** é sempre FIFO (sai o primeiro elemento que entrou)
  - Um **deque** apenas permite aceder ao primeiro ou último elemento
- Precisamos de um TAD que tenha em conta as **prioridades**.
  - Se fossem sempre só as 5 prioridades do sistema de triagem, podíamos ter usar filas. Mas o que acontece se forem mais?
  - E para um caso mais geral onde a quantidade de diferentes prioridades não está limitada? (ex: a prioridade é um número qualquer, um *double*)
- Uma **fila de prioridade** (**Priority Queue**) é um TAD para guardar uma coleção de elementos suportando três operações principais:
  - **insert(x)** que adicionar um elemento x à coleção
  - **peek()** que devolve (sem retirar) o elemento mais **prioritário**
  - **remove()** que devolve e retira o elemento mais **prioritário**

Filas de prioridade - Aplicações

- As filas de prioridade são úteis em muitos mais cenários. Aqui ficam mais alguns **exemplos** onde podem ser aplicadas:
  - Uma fila (ex: correios) com **atendimento prioritário** (ex: grávidas)
  - Um router com **tráfego prioritário** (ex: chamadas VoIP)
  - Simulação de **eventos discretos**: imaginemos vários eventos que vão começar em alturas diferentes. Podemos usar filas de prioridade para saber qual o próximo evento a acontecer (hora de início é a prioridade)
  - Existem muitos **algoritmos** que vão aprender que usam filas de prioridade com um bloco básico. Alguns exemplos:
    - **Algoritmo de Dijkstra** (caminhos mais curtos): para saber qual o próximo nó mais perto que ainda não foi processado
    - **Algoritmo de Prim** (para árvores de suporte de custo mínimo): para saber qual o nó mais próximo da árvore que ainda não foi adicionado
    - **Algoritmo A*** (pesquisa *best-first*): para saber qual o próximo nó a visitar que tenha o melhor valor da heurística usada

Filas de Prioridade - o que é mais prioritário?

- Para poder pensar como implementar, precisamos de conseguir **definir** o que significa ser **"mais prioritário"**
- Sem perda de generalidade, vamos assumir que estamos a trabalhar com elementos **comparáveis** e que o mais prioritário é o **menor**.
  - Por exemplo, se tivermos os inteiros {8,4,5}, o menor é o 4.
  - Num caso como o a triagem, bastaria associar às cores mais prioritárias números menores (ex: vermelho=1, laranja=2, amarelo=3)
  - Se nos fosse útil o maior, e não o menor, basta mudar as prioridades de forma correspondente (ex: se guardarmos o negativo dos números, o "menor" é o que originalmente era maior)
- Deste modo, a operação **remove()**, pode também ser pensada como um **removeMin()** (e vamos chamar **min()** à operação **peek()**)



Filas de Prioridade - Implementações

- Como podemos implementar uma fila de prioridade usando as estruturas de dados que já conhecemos anteriormente?
  - **Lista Desordenada:** array ou lista ligada sem nenhuma ordem. Inserir é fácil mas devolver mínimo implica pesquisa linear
  - **Lista Ordenada:** array ou lista ligada por ordem crescente. Devolver mínimo é fácil (está no início) mas inserir implica manter ordem
  - **Árvore Binária de Pesquisa:** inserir e retirar (folha mais à esquerda) tem custo associado à altura da árvore

	insert	**min**	**removeMin**
Lista Desordenada	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Lista Ordenada	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Árvore Bin. Pesquisa *(se estiver equilibrada)*	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

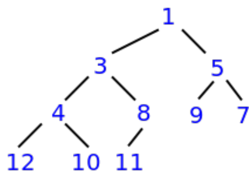
Nota: se a árvore de pesquisa não estiver equilibrada, inserir e remover pode custar $\mathcal{O}(n)$. Podemos também criar variável dedicada para conter o mínimo e passar a responder a min() em $\mathcal{O}(1)$

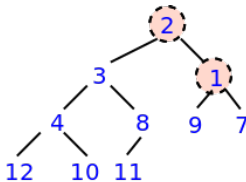
Heaps - Invariante

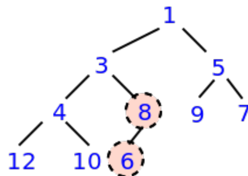
- Vamos ver uma (outra) solução **especializada** e muito **eficiente**
- Uma **heap** é uma árvore que obedece à seguinte restrição:

Invariante de (min)Heap

O pai de qualquer nó tem sempre mais prioridade que o filho, ou seja, numa **minHeap**, o pai é sempre *menor* que os filhos







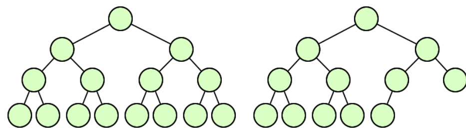
Nota: numa *maxHeap*, um nó seria sempre maior que o filho

Heaps - Altura O(log n)

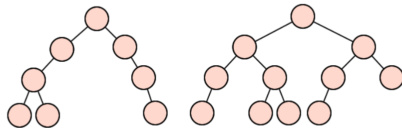
- Para garantir a eficiência das operações associadas, uma heap deve ser também uma **árvore binária completa**:

Árvore completa

Uma árvore onde todos os níveis (excepto potencialmente o último) estão completamente preenchidos com nós, e todos os nós estão o mais à esquerda possível.



2 exemplos de árvore completas

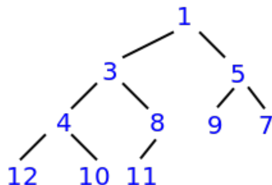


2 exemplos de árvores que não são completas

- Numa árvore completa com n nós, a altura é $\mathcal{O}(\log n)$
  - É uma árvore muito equilibrada e já vimos antes uma explicação, mas intuitivamente, podem pensar que para *aumentar em 1 unidade* a altura, é necessário *duplicar* o número de elementos

Heaps - Mapeamento num array

- A maneira mais fácil e compacta de implementar uma heap é usar um **array** que *implicitamente* representa a árvore.
  - Os elementos aparecem num array numa *ordem em largura* (de cima para baixo, da esquerda para a direita)
  - Se colocarmos a raíz na posição 1, então:
    - O filhos do nó na posição x estão nas posições $x * 2$ e $x * 2 + 1$
    - O pai de um nó x está na posição $x/2$ (divisão inteira)
- Vejamos um exemplo:



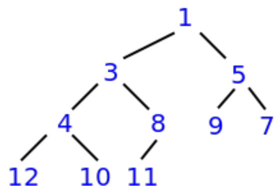
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	3	5	4	8	9	7	12	10	11

Ex: filhos da posição 3 (nó 5) estão nas posições 3*2=6 (nó 9) e 3*2+1=7 (nó 7). O pai da posição 3 é o nó na posição 3/2=1 (nó 1).

- Como a árvore é completa, isto significa que o array fica com posições consecutivas preenchidas

Heaps - operação min()

- Como cada nó é menor que os seus filhos, o menor nó de todos terá de estar garantidamente na raíz da heap (1º elemento do array):



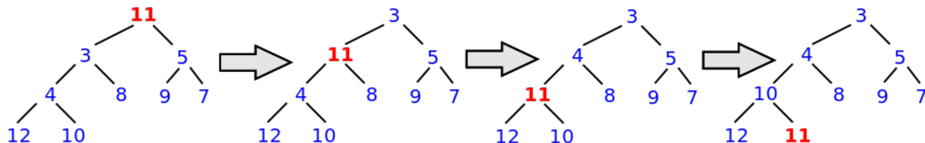
1	2	3	4	5	6	7	8	9	10
1	3	5	4	8	9	7	12	10	11

- **min():** tempo $\mathcal{O}(1)$

Heaps - operação removeMin()

- Depois de remover a raíz é necessário repor as condições de heap. Para isso, faz-se o seguinte:
  - Pega-se no último elemento do array e coloca-se na posição da raíz (a árvore fica completa)
  - O elemento "baixa" (**downHeap**), trocando com o menor dos filhos, até que a invariante de heap esteja reposta
  - No máximo percorre-se toda a altura da árvore, que é $\mathcal{O}(\log n)$

Um exemplo para a heap anterior, depois de retirado o mínimo (1) e colocado o último elemento (11) na sua posição (a raíz):

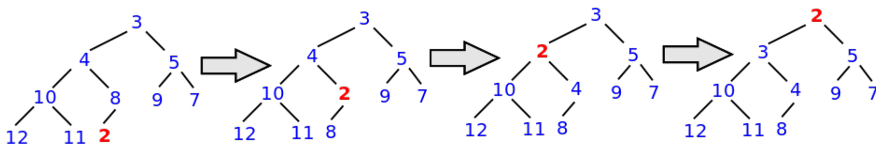


- **removeMin():** tempo $\mathcal{O}(\log n)$

Heaps - operação insert(x)

- **Inserir** um elemento passa por:
  - Colocá-lo logo a seguir à última posição ocupada, a primeira livre do array (a árvore fica completa)
  - O elemento "sobe" (**upHeap**), trocando com o pai, até que a invariante de heap esteja reposta
  - No máximo percorre-se a altura da árvore, que é $\mathcal{O}(\log n)$

Um exemplo para a inserção de 2 na heap do slide anterior:



- **insert(x):** tempo $\mathcal{O}(\log n)$

HeapSort

- As heaps sugerem um **algoritmo de ordenação** óbvio. Para ordenar n elementos, basta fazer o seguinte:
  - Criar uma heap com os n elementos
  - Retirar um a um os n elementos da heap
- Como os elementos saem por ordem de prioridade, vão aparecer... por ordem crescente!
- Este processo implica n inserções, seguidas de n remoções. Como o custo de cada operação é logarítmico, o custo total será de $\mathcal{O}(n \log n)$
- Este algoritmo (na sua essência) é conhecido como **HeapSort**

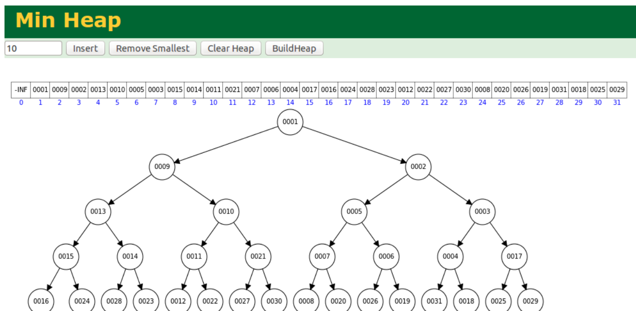
Nota: É possível fazer que com que a parte de criar uma heap a partir de n elementos (*heapify*) seja feita em $\mathcal{O}(n)$, e não em $\mathcal{O}(n \log n)$ (como acontece se for feito via n inserções).

Para isso, basta colocar os elementos no array (por qualquer ordem) e depois chamar downHeap a todas as posições, começando do penúltimo nível e subindo até chegar ao primeiro. Os mais curiosos podem ver uma prova do tempo linear para criar uma heap com n elementos no livro recomendado "*Introduction to Algorithms*".

Em todo o caso, a ordenação continua a ser $\mathcal{O}(n \log n)$, pois é sempre necessário remover os n elementos um a um, resultando num custo de $\mathcal{O}(n \log n)$.

- Podem visualizar inserção e remoção em heaps (experimentem o url indicado):

https://www.cs.usfca.edu/~galles/visualization/Heap.html



Heaps - Implementação

- Vamos agora implementar heaps em Java
- A principal decisão tem a ver em como usar a **noção de prioridade**:
  1. Uma hipótese seria fazer algo como os dicionários, e um "nó" da Heap ter **dois atributos**: o valor da prioridade (a "chave") e o objecto associado (o "valor")
  2. Outra hipótese é usar a **ordenação natural** dos objectos colocados na heap, ou seja, se admitirmos que os objectos são comparáveis, o objecto "menor" vai naturalmente ficar na raíz da árvore
- Para "imitar" de certa forma as filas de prioridade do próprio Java, vamos usar a 2ª hipótese
- Para mostrar as potencialidades da linguagem nesta fase avançada do semestre vamos também dar a hipótese de ser fornecido um **comparador customizado** (que não tem de seguir a ordenação natural), tal com acontece com as filas de prioridade do próprio Java.

Heaps - Implementação

- A assinatura dos principais métodos públicos da nossa classe que implementa uma minHeap vai ser a seguinte:

```
// Heap para objectos do tipo T
public class MinHeap<T> {
   MinHeap(int capacity) {/*...*/} // Construtor:  heap com dada capacidade
   MinHeap(int cap, Comparator<T> comp) {/*...*/} // Comparador customizado

   public int size() {/*...*/}         // Número de elementos
   public boolean isEmpty() {/*...*/} // Heap vazia?

   public T min() {/*...*/}      // Devolve o elemento mínimo
   public T removeMin {/*...*/} // Remove e devolve o elemento mínimo
   public void insert(T value) {/*...*/} // Inserir um elemento na heap
}
```

Heaps - Implementação

- Os atributos e os construtores:

```
import java.util.Comparator;

public class MinHeap<T> {
   private T[] data; // Guardar elementos entre posicoes 1 e size
   private int size; // Quantidade de elementos
   private Comparator<T> comparator; // Comparador (opcional)

   // Construtor (heap com uma dada capacidade)
   @SuppressWarnings("unchecked") // Por causa do array de genéricos
   MinHeap(int capacity) {
      // Java proíbe directamente array de genéricos, daí o cast
      data = (T[]) new Object[capacity+1];
      size = 0;
      comparator = null;
   }

   // Construtor (heap com uma dada capacidade e comparador customizado)
   MinHeap(int capacity, Comparator<T> comp) {
      this(capacity); // Chama o construtor padrão
      comparator = comp;
   }
   // ...
}
```

Heaps - Implementação

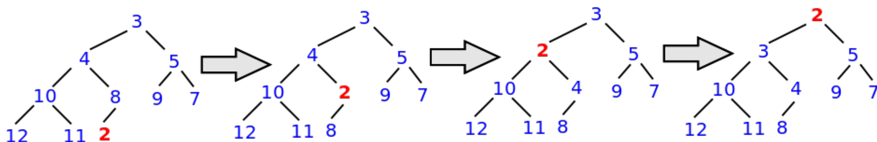
- Os métodos "fáceis" (notar que se deve começar a preencher o array a partir da posição 1, que irá conter a raíz da heap):
  (o código deste e dos próximos slides está dentro da classe *MinHeap*)

```
// Número de elementos guardados na heap
public int size() {
   return size;
}

// Heap vazia?
public boolean isEmpty() {
   return (size==0);
}

// Devolver (sem retirar) elemento mínimo
public T min() {
   if (isEmpty()) return null;
   return data[1];
}
```

Heaps - Implementação

- **Inserir** um elemento na heap: colocar no final do array e chamar *upHeap* até esse elemento ficar na sua posição:

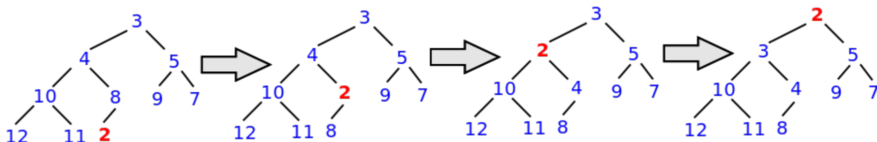
```
// Inserir um elemento na heap
public void insert(T value) {
   if (size+1 >= data.length) throw
      new RuntimeException("Heap is full");
   size++;
   data[size] = value;
   upHeap(size);
}
```



Heaps - Implementação

- **upHeap:** fazer subir o elemento enquanto for menor que o pai

```
// Fazer um elemento subir na heap até à sua posição
private void upHeap(int i) {
   // Enquanto o elemento for menor que o pai e não estiver na raíz
   while (i>1 && smaller(i, i/2)) {
      swap(i, i/2); // Trocar com o pai
      i = i/2;
   }
}
```



- Estamos a usar dois métodos auxiliares:
  - **smaller(i,j)**: devolve *true* se elemento da posição *i* for menor que o elemento da posição *j* ou *false* caso contrário
  - **swap(i,j)**: troca os elementos das posições *i* e *j*

Heaps - Implementação

- Os métodos **smaller** e **swap**:

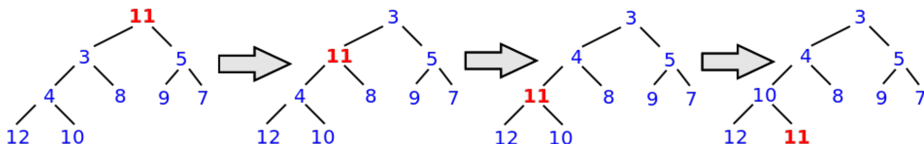
```
// Saber se o elemento na posição i é menor que o elemento na posição j
// Para que o java não se queixe do cast que diz que elementos são comparáveis
@SuppressWarnings("unchecked")
private boolean smaller(int i, int j) {
   // Se não existe comparador usar comparação natural
   if (comparator == null)
     return ((Comparable<? super T>) data[i]).compareTo(data[j]) < 0;
   // Se existe comparador...  usá-lo
   else
     return comparator.compare(data[i], data[j]) < 0;
}

// Trocar elementos entre as posições i e j
private void swap(int i, int j) {
   T tmp = data[i];
   data[i] = data[j];
   data[j] = tmp;
}
```

- Interface **Comparator** define método *compare(a,b)* que deve comparar objectos *a* e *b* e devolver número negativo, zero ou positivo como o *compareTo* (exemplo de uso daqui a pouco)

Heaps - Implementação

- **Remover** um elemento da heap: devolver raíz; colocar último na raíz e chamar *downHeap* até esse elemento ficar na sua posição:

```
// Remover e devolver elemento mínimo
public T removeMin() {
   if (isEmpty()) return null;
   T min = data[1];
   data[1] = data[size];
   data[size] = null; // Para ajudar no garbage collection
   size--;
   downHeap(1);
   return min;
}
```

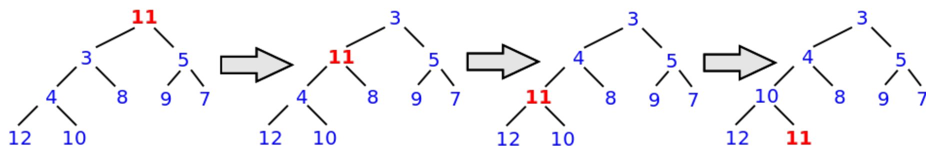


- `data[size] = null`: "apaga" a referência ao objecto na última posição. Se futuramente o elemento for removido, não existe nenhuma referência sua no array pode ser removido da memória com segurança pelo *Garbage Collector*.

Heaps - Implementação

- **downHeap:** fazer descer o elemento enquanto um dos filhos for menor (e trocar com o menor dos filhos)

```
// Fazer um elemento descer na heap até à sua posição
private void downHeap(int i) {
   while (2*i <= size) { // Enquanto estiver dento dos limites da heap
      int j = i*2;
      // Escolher filho mais pequeno (posicao i*2 ou i*2+1)
      if (j<size && smaller(j+1, j)) j++;
      // Se nó já é menor que filho mais pequeno, terminamos
      if (smaller(i, j)) break;
      // Caso contrário, trocar com filho mais pequeno
      swap(i, j);
      i = j;
   }
}
```



Heaps - Exemplo de uso

- Vamos ver agora alguns exemplos de uso.
- Para começar, uma heap de números inteiros:
  (sem indicar comparador é usada a ordenação natural)

```
// Criar uma heap h (para inteiros) com capacidade para 100 números
MinHeap<Integer> h = new MinHeap<>(100);

// Criar um array com 10 inteiros
int[] v = {10,4,3,12,9,1,7,11,5,8};

// Inserir na heap h todos os elementos do array [v]
for (int i=0; i<v.length; i++)
   h.insert(v[i]);

// Retirar um a um os elementos e imprimir
for (int i=0; i<v.length; i++)
   System.out.print(h.removeMin() + " ");
System.out.println();
```

Output:

```
1 3 4 5 7 8 9 10 11 12
```

Heaps - Exemplo de uso

- Podemos usar qualquer tipo comparável, e não só números inteiros.
- Aqui fica um exemplo com Strings:

```
// Criar uma heap h (para strings)
MinHeap<String> h = new MinHeap<>(100);

// Criar um array 5 strings
String[] v = {"heap", "arvore", "pilha", "fila", "deque"};

// Inserir na heap h todos os elementos do array [v]
for (int i=0; i<v.length; i++)
   h.insert(v[i]);

// Retirar um a um os elementos e imprimir
for (int i=0; i<v.length; i++)
   System.out.print(h.removeMin() + " ");
System.out.println();
```

Output:

```
arvore deque fila heap pilha
```

Heaps - Exemplo de uso

- Podemos usar classes nossas que implementem o interface *Comparable*. Por exemplo uma pessoa com primeiro e último nome:

```
class Person implements Comparable<Person> {
   String first, last;
   Person(String f, String l) {first=f; last=l;}
   public int compareTo(Person p) {return first.compareTo(p.first);}
   public String toString() {return "(" + first + " " + last + ")";}
}
```

- Testando com uma heap:

```
   MinHeap<Person> h = new MinHeap<>(100);
   Person[] v = new Person[3];
   v[0] = new Person("Pedro", "Ribeiro");
   v[1] = new Person("Luis", "Lopes");
   v[2] = new Person("Eduardo", "Marques");
   for (int i=0; i<v.length; i++) h.insert(v[i]);
   for (int i=0; i<v.length; i++) System.out.print(h.removeMin()+" ");
   System.out.println();
```

Output:

```
(Eduardo Marques) (Luis Lopes) (Pedro Ribeiro)
```

Heaps - Exemplo de uso

- Se mudarmos o método *compareTo*, mudamos a noção de prioridade e o comportamento da heap (agora comparamos o último nome):

```java
class Person implements Comparable<Person> {
   String first, last;
   Person(String f, String l) {first=f; last=l;}
   public int compareTo(Person p) {return last.compareTo(p.last);}
   public String toString() {return "(" + first + " " + last + ")";}
}
```

- Testando com uma heap:

```java
   MinHeap<Person> h = new MinHeap<>(100);
   Person[] v = new Person[3];
   v[0] = new Person("Pedro", "Ribeiro");
   v[1] = new Person("Luis", "Lopes");
   v[2] = new Person("Eduardo", "Marques");
   for (int i=0; i<v.length; i++) h.insert(v[i]);
   for (int i=0; i<v.length; i++) System.out.print(h.removeMin()+" ");
   System.out.println();
```

Output:

```
(Luis Lopes) (Eduardo Marques) (Pedro Ribeiro)
```

Heaps - Exemplo de uso

- E se quisermos usar uma comparação diferente da natural? É possível passar um comparador e a Heap irá usar esse comparador
- Normalmente, métodos do próprio Java que precisam de comparar (ex: Arrays.sort) podem também receber como input um comparador para ser usado ao invés da ordenação natural.
- Imaginemos por exemplo que queremos uma heap de Strings onde a prioridade é o tamanho e não a ordem alfabética (a ordem natural das Strings). Poderíamos criar a seguinte classe comparadora:

```
class LengthComparator implements Comparator<String> {
   // Assumindo que não são nulas
   public int compare(String a, String b) {
      // Conseguem perceber porque podemos subtrair?
      // Quando é que a subtração dá um número negativo?  E positivo?  E zero?
      return a.length() - b.length();
   }
}
```

(nota: podíamos ter usado uma classe "anónima" ou expressões lambda, mas não vamos abordar esses conceitos agora em EDados)

Heaps - Exemplo de uso

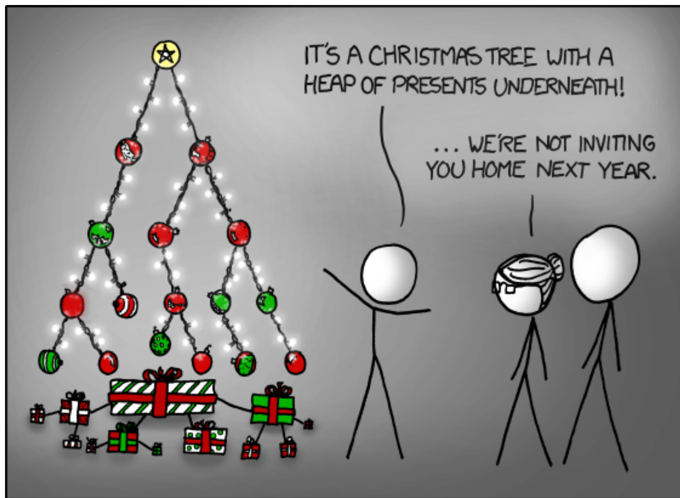
- Agora basta chamar a versão do construtor que tem um comparador

```
    // Criar uma heap h (para strings) com comparador customizado
    MinHeap<String> h = new MinHeap<>(100, new LengthComparator());

    // Criar um array 5 strings
    String[] v = {"aaaaa", "bbb", "cccc", "d", "ee"};

    // Inserir na heap h todos os elementos do array [v]
    for (int i=0; i<v.length; i++)
       h.insert(v[i]);

    // Retirar um a um os elementos e imprimir
    for (int i=0; i<v.length; i++)
       System.out.print(h.removeMin() + " ");
       System.out.println();
 }
```

Output:

```
d ee bbb cccc aaaaa
```

Heaps



(imagem do site xkcd.com)

"Not only is that terrible in general, but you just KNOW Billy's going to open the root present first, and then everyone will have to wait while the heap is rebuilt."