

# Percorrer uma coleção

- Podemos utilizar o ciclo for(each) para percorrer uma coleção:

```
/**
 * Média da turma
 *
 * @return um double com a média da turma
 */
public double media() {
    double tot = 0.0;

    for(Aluno a: lstAlunos)
        tot += a.getNota();

    return tot/lstAlunos.size();
}
```

```
/**
 * Quantos alunos passam?
 *
 * @return um int com nº alunos que passa
 */
public int quantosPassam() {
    int qt = 0;

    for(Aluno a: lstAlunos)
        if (a.passa()) qt++;

    return qt;
}
```

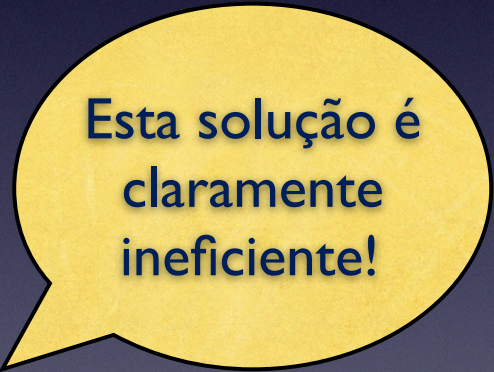
```
public boolean passa() {
    return this.nota >= Aluno.NOTA_PARA_PASSAR;
}
```

Na classe **Aluno**

- ... mas...
- podemos querer parar antes do fim
- podemos não ter acesso à posição do elemento na colecção (no caso dos conjuntos)
- estamos sempre a repetir o código do ciclo

```
/**
 * Alguns alunos passa?
 *
 * @return true se algum aluno passa
 */
public boolean algumPassa() {
    boolean algum = false;

    for(Aluno a: lstAlunos)
        if (a.passa())
            algum = true;
    return algum;
}
```



Esta solução é claramente ineficiente!

- logo, é necessário um mecanismo mais flexível para percorrer colecções

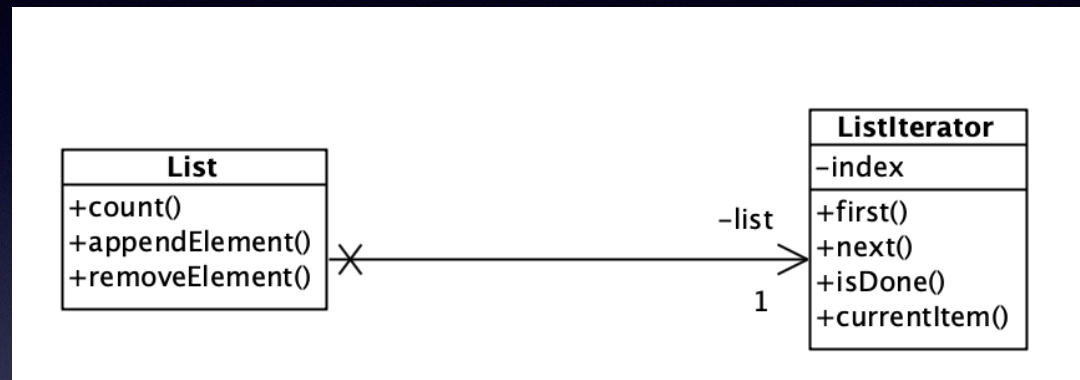


# Iteradores externos

- O **Iterator** é um padrão de concepção bem conhecido e que permite providenciar uma forma de aceder aos elementos de uma colecção de objectos, sem que seja necessário saber qual a sua representação interna
- basta para tal, que todas as colecções saibam criar um iterator!
- não precisamos saber como tal é feito!



- Um iterador de uma lista poderia ser:



- o iterator precisa de ter mecanismos para:
  - aceder ao objecto apontado
  - avançar
  - determinar se chegou ao fim

- Iterator API

## Method Summary

### Methods

Modifier and Type	Method and Description
boolean	<b>hasNext()</b> Returns <code>true</code> if the iteration has more elements.
<b>E</b>	<b>next()</b> Returns the next element in the iteration.
void	<b>remove()</b> Removes from the underlying collection the last element returned by this iterator (optional operation).



- Utilizando Iterators...

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean algumPassa() {
    boolean alguem = false;
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext() && !alguem) {
        a = it.next();
        alguem = a.passa();
    }
    return alguem;
}
```



lista de  
alunos

- remover alunos...

```
/**
 * Remover notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerPorNota(int nota) {
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext()) {
        a = it.next();
        if (a.getNota() < nota)
            it.remove();
    }
}
```



# Iterator<E>

- Em resumo...
- Todas as colecções implementam o método: **Iterator<E> iterator()** que cria um iterador activo sobre a colecção
- Padrão de utilização:

```
Iterator<E> it = colecção.iterator();  
E elem;  
  
while(it.hasNext()) {  
    elem = it.next();  
    // fazer algo com elem  
}
```



- Procurar:

```
boolean encontrado = false;
Iterator<E> it = coleção.iterator();
E elem;

while(it.hasNext() && !encontrado) {
    elem = it.next();
    if (criterio de procura sobre elem)
        encontrado = true;
}
// fazer alguma coisa com elem ou com encontrado
```

- Remover:

```
Iterator<E> it = coleção.iterator();
E elem;

while(it.hasNext()) {
    elem = it.next();
    if (criterio sobre elem)
        it.remove();
}
```

# Iteradores internos

- Todas as colecções implementam o método: **forEach()**
- Aceita uma função para *trabalhar* em todos os elementos da coleção
- É implementado com um for each...

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```



- Iterador externo

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    for(Aluno a: lstAlunos)
        a.sobeNota(bonus);
}
```

- Iterador interno - forEach()

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    lstAlunos.forEach((Aluno a) -> {a.sobeNota(bonus);});
}
```



# Expressões Lambda

- **(Tipo p, ...) -> {corpo do método}**
- Um método *anônimo*, que pode ser passado como parâmetro
- Expressão pode ser simplificada:

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguarBenta(int bonus) {
    lstAlunos.forEach(a -> a.sobeNota(bonus));
}
```

Tipo de **a** é **Aluno**, uma vez que **lstAlunos** é do tipo **List<Aluno>**



# Streams

- Todas as colecções implementam o método **stream()**
- Streams: sequências de valores que podem ser passados numa *pipeline* de operações.
- As operações alteram os valores (produzindo novas Streams ou *reduzindo* o valor a um só)

```
public int quantosPassam() {  
    int qt = 0;  
  
    for(Aluno a: lstAlunos)  
        if (a.passa()) qt++;  
  
    return qt;  
}
```

```
public long quantosPassam() {  
  
    return lstAlunos.stream().filter(a -> a.passa()).count();  
}
```



- Colecções implementam método **stream()**
  - Produz uma Stream
- Alguns dos principais métodos da API de **Stream**
  - `allMatch()` - determina se todos os elementos fazem match com o predicado fornecido
  - `anyMatch()` - determina se algum elemento faz match
  - `noneMatch()` - determina se nenhum elemento faz match
  - `count()` - conta os elementos da Stream
  - `filter()` - filtra os elementos da Stream usando um predicado
  - `map()` - transforma os elementos da Stream usando uma função
  - `collect()` - junta os elementos da Stream numa lista ou String
  - `reduce()` - realiza uma redução (fold)
  - `sorted()` - ordena os elementos da Stream
  - `toArray()` - retorna um array com os elementos da Stream



- **alguemPassa()** - utilizando Streams...

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    return lstAlunos.stream().anyMatch(a -> a.passa());
}
```

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    boolean alguem = false;
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext() && !alguem) {
        a = it.next();
        if (a.passa())
            alguem = true;
    }
    return alguem;
}
```

# Referências a métodos

- Classe::método
  - Permitem referir um método pelo seu nome
  - Úteis nas expressões lambda
  - Objecto que recebe a mensagem está implícito no contexto

```
public boolean alguemPassa() {  
    return lstAlunos.stream().anyMatch(Aluno::passa);  
}
```



- **getLstAlunos()**

```
public List<Aluno> getLstAlunos() {  
    return lstAlunos.stream().map(Aluno::clone).collect(Collectors.toList());  
}
```

```
public List<Aluno> getLstAlunos() {  
    List<Aluno> res = new ArrayList<>();  
  
    for(Aluno a: lstAlunos)  
        res.add(a.clone());  
    return res;  
}
```

- remover alunos utilizando Streams

```
/**
 * Remover as notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerAlunos(int nota) {
    lstAlunos = lstAlunos.stream()
        .filter(a -> a.getNota() >= nota)
        .collect(Collectors.toList());
}
```

mas...

```
public void removerPorNota(int nota) {
    lstAlunos.removeIf(a -> a.getNota() < nota);
}
```

```
/**
 * Remover notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerPorNota(int nota) {
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext()) {
        a = it.next();
        if (a.getNota() < nota)
            it.remove();
    }
}
```



- Existem Steams específicas para os tipos primitivos
  - IntStream - **mapToInt(...)**
  - DoubleStream - **mapToDouble(...)**
  - ...
- Alguns dos principais métodos específicos
  - average() - determina a média
  - max() - determina o máximo
  - min() - determina o mínimo
  - sum() - determina a soma



- **media()** - utilizando Streams...

```
/**
 * Média da turma
 *
 * @return um double com a média da turma
 */
public double media() {
    double tot = lstAlunos.stream()
        .mapToDouble(Aluno::getNota)
        .sum();
    return tot/lstAlunos.size();
}
```

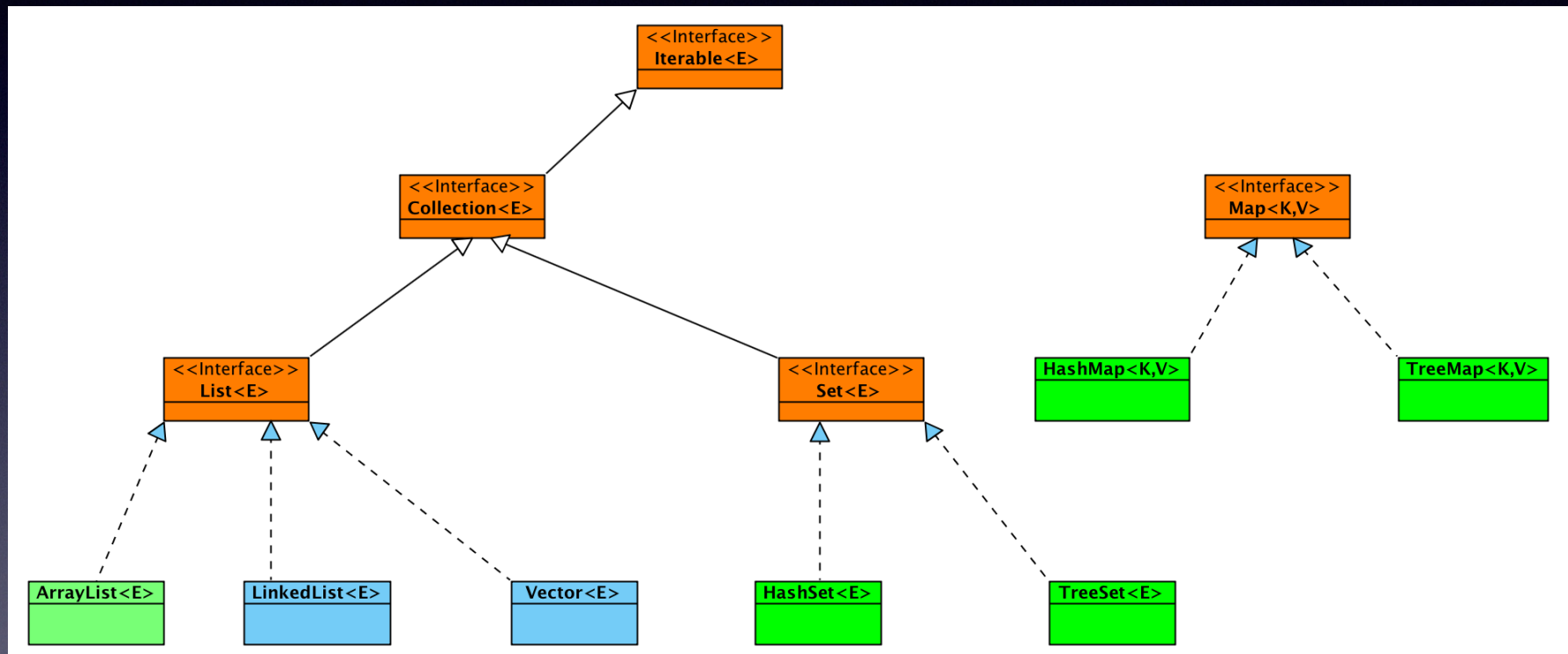
```
public double media() {
    double tot = 0.0;

    for(Aluno a: lstAlunos)
        tot += a.getNota();

    return tot/lstAlunos.size();
}
```



# Coleções e Maps





# Set<E>

Adicionar elementos	<code>boolean add(E e)</code> <code>boolean addAll(Collection c)</code>
Alterar o Set	<code>void clear()</code> <code>boolean remove(Object o)</code> <code>boolean removeAll(Collection c)</code> <code>boolean retainAll(Collection c)</code> <code>boolean removeIf(Predicate p)</code>
Consultar	<code>boolean contains(Object o)</code> <code>boolean containsAll(Collection c)</code> <code>boolean isEmpty()</code> <code>int size()</code>
Iteradores externos	<code>Iterator&lt;E&gt; iterator()</code>
Iteradores internos	<code>Stream&lt;E&gt; stream()</code> <code>void forEach(Consumer c)</code>
Outros	<code>boolean equals(Object o)</code> <code>int hashCode()</code>

# Set<E>

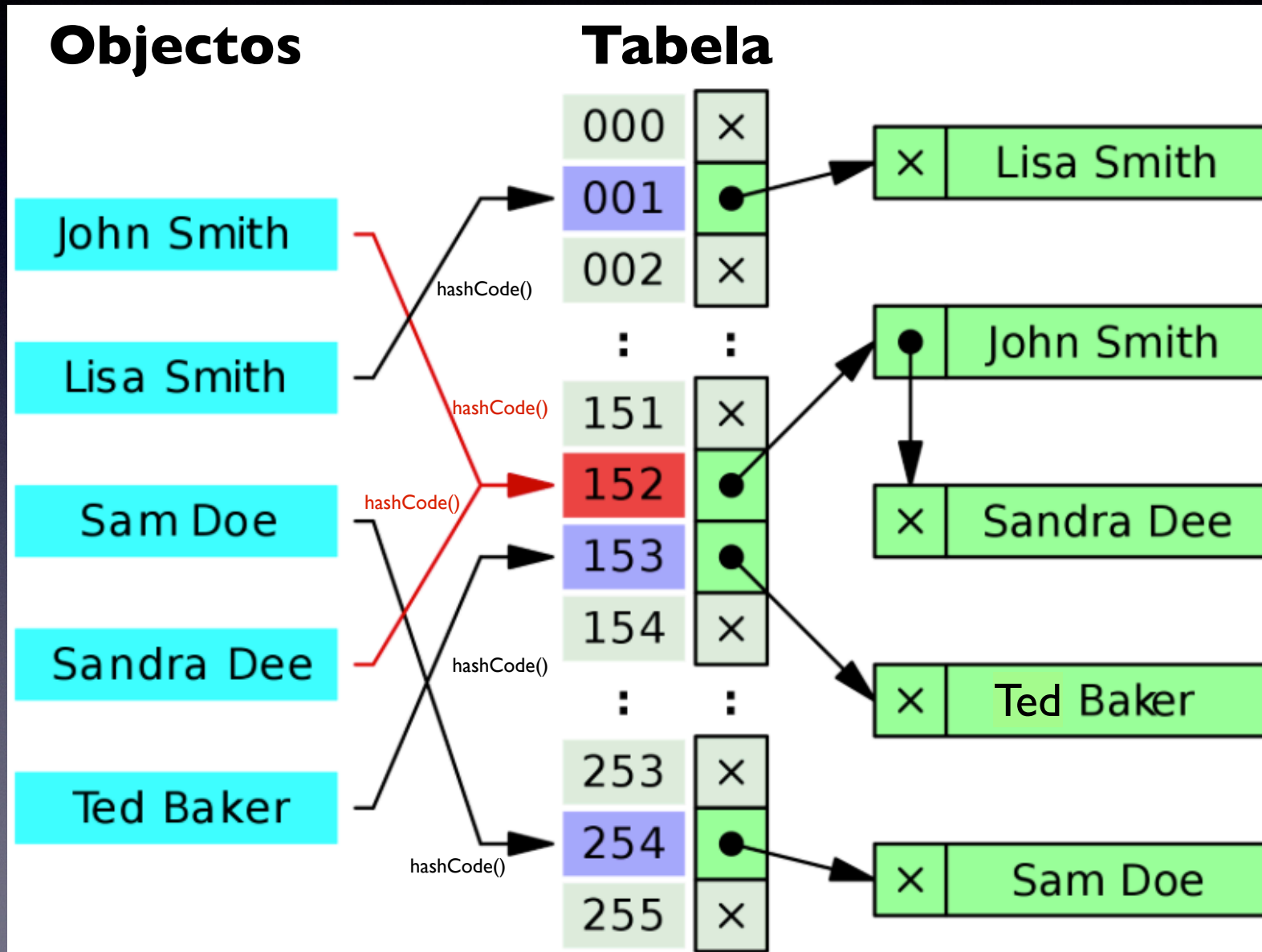
- Utilizar sempre que se quer garantir ausência de elementos repetidos
- O método add testa se o objecto existe
- O método contains utiliza a lógica do equals, mas não só...
- Duas implementações: **HashSet<E>** e **TreeSet<E>**



# HashSet<E>

- Utiliza uma tabela de Hash para guardar os elementos.
- O método **add** calcula o valor de hash do objecto a adicionar para determinar a sua posição na estrutura de dados
- O método **contains** necessita de saber o valor de hash do objecto para determinar a posição em que o encontra
- Logo, não chega ter o **equals** definido
  - é necessário ter o método **hashCode()**

# Tabelas de hash





# Método hashCode()

- Sempre que se define o método **equals**, deve definir-se também o método **hashCode()**
  - objectos iguais devem ter o mesmo código de hash
- Se **hashCode()** não for definido é utilizada a implementação por omissão, logo:
  - recorre à referência do objecto
  - objectos iguais (em valor) podem ter códigos diferentes!



# Método hashCode()

- Exemplo
  - nome é String
  - número é int
  - nota é double

```
public int hashCode() {  
    int hash = 7;  
    long aux;  
  
    hash = 31*hash + nome.hashCode();  
    hash = 31*hash + numero;  
    aux = Double.doubleToLongBits(nota);  
    hash = 31*hash + (int)(aux^(aux >>> 32));  
    return hash;  
}
```



# Implementar o hashCode()

(exemplo!)

1. Definir **int hash = x ;** //(x diferente de 0)
2. Calcular o código de hash de cada var. instância **v** conforme o seu tipo:
  - boolean: **(v ? 0 : 1);**
  - byte, char, short ou int: **(int)v;**
  - long: **(int)(v ^ (v >>> 32));**
  - float: **Float.floatToIntBits(v);**
  - double: calcular **Double.doubleToLongBits(v)** e usar a regra dos long no resultado
  - objectos: **v.hashCode()**, ou **0** se **v == null**;
  - arrays: tratar cada elemento do array como uma variável de inst.
3. Combinar cada um dos valores calculados acima no resultado do seguinte modo: **hash = 37 \* hash + valor;**
4. **return result;**

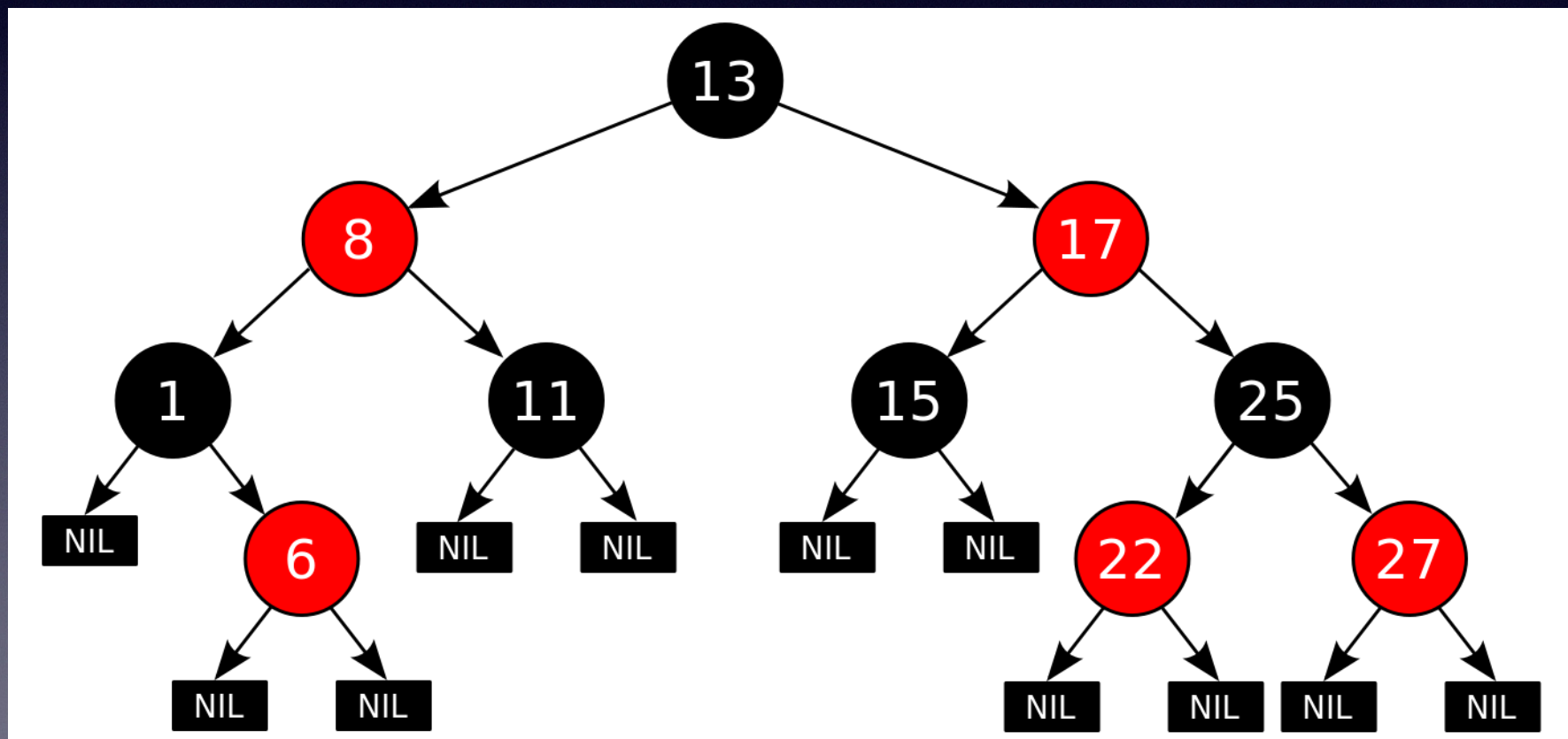


# TreeSet<E>

- Utiliza uma árvore binária auto-balanceada do tipo *Red-Black* para guardar os elementos.
- É necessário fornecer um método de comparação dos objectos
  - **compareTo()** - na classe **E**
  - **compare()** - num **Comparator**
- sem este método de comparação não é possível utilizar o TreeSet, a não ser para tipos de dados simples (String, Integer, etc.)



# Red-black self-balancing binary search tree



# Método compareTo()

- Define a ordem “natural” das instâncias de uma dada classe
- Definido como um método de instância
  - Compara o objecto receptor com outro passado como parâmetro
  - Se objectos são iguais
    - resultado: 0
  - Se objecto receptor é “maior”
    - resultado > 0 (neste caso 1)
  - Se objecto receptor é “menor”
    - resultado < 0 (neste caso -1)

```
public int compareTo(Aluno a) {  
    int numA = a.getNumero();  
    int res;  
  
    if (this.numero==numA)  
        res = 0;  
    else if (numero>numA)  
        res = 1;  
    else  
        res = -1;  
    return res;  
}
```



# Método compareTo()

- Classe deve implementar **Comparable<T>**
  - **public class Aluno implements Comparable<Aluno>**
- Ordem natural com base no número (versão alternativa)

```
public int compareTo(Aluno a) {  
    if (this.getNumero() == a.getNumero())  
        return 0;  
    if (this.getNumero() > a.getNumero())  
        return 1;  
    return -1;  
}
```

- Ordem natural com base no nome

```
public int compareTo(Aluno a) {  
    return this.nome.compareTo(a.getNome());  
}
```

- No entanto, só pode existir uma ordem natural (um método **compareTo()**) em cada classe.