Aula 17
Tabelas de Símbolos Ordenadas e Coleções em Java

Algoritmos e Estruturas de Dados

Tabelas de símbolos

Ordenadas

Tabelas de Símbolos Ordenadas

- Tabelas de símbolos onde existe uma ordenação explícita das chaves

  - Vantagens

    Para além de pesquisa eficiente

    Permitem a implementação fácil e eficiente de um conjunto de métodos adicionais

Tabelas de Símbolos Ordenadas

- Tabelas de símbolos onde existe uma ordenação explícita das chaves
  - A árvore binária de pesquisa é uma tabela de símbolos ordenada

  - Vantagens

    Para além de pesquisa eficiente

    Permitem a implementação fácil e eficiente de um conjunto de métodos adicionais

Tabelas de Símbolos Ordenadas

- Vantagens
  - Permitem a implementação eficiente de uma série de métodos de pesquisa

Tabelas de Símbolos Ordenadas

public class ST<Key extends Comparable<Key>, Value>	
ST()	*create an ordered symbol table*
void put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is null)*
Value get(Key key)	*value paired with* key *(null if* key *is absent)*
void delete(Key key)	*remove* key *(and its value) from table*
boolean contains(Key key)	*is there a value paired with* key*?*
boolean isEmpty()	*is the table empty?*
int size()	*number of key-value pairs*
Key min()	*smallest key*
Key max()	*largest key*
Key floor(Key key)	*largest key less than or equal to* key
Key ceiling(Key key)	*smallest key greater than or equal to* key
int rank(Key key)	*number of keys less than* key
Key select(int k)	*key of rank* k
void deleteMin()	*delete smallest key*
void deleteMax()	*delete largest key*
int size(Key lo, Key hi)	*number of keys in* [lo..hi]
Iterable<Key> keys(Key lo, Key hi)	*keys in* [lo..hi]*, in sorted order*
Iterable<Key> keys()	*all keys in the table, in sorted order*

API for a generic ordered symbol table

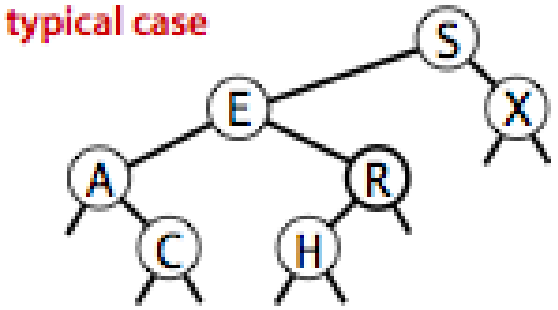
Operações

Baseadas na ordem

Min e Max

- Menor chave de uma BST

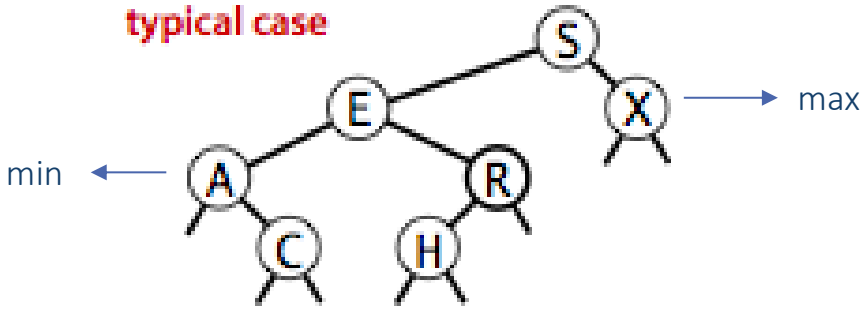
- Maior chave de uma BST



Min e Max

- Menor chave de uma BST

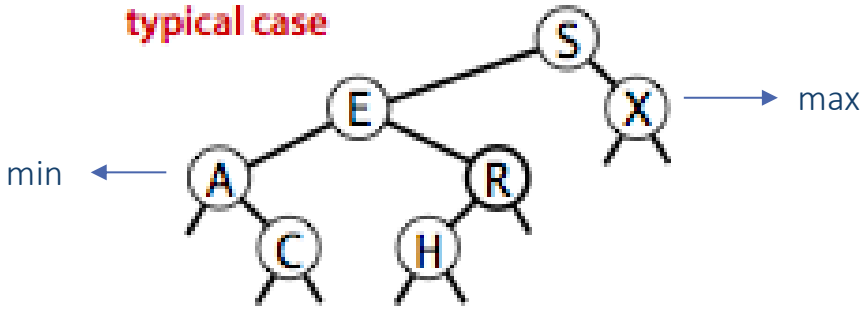
- Maior chave de uma BST



Min e Max

- Menor chave de uma BST
  - Elemento mais à esquerda
  - Avançar para a esquerda

    Até encontrar um nó sem filho esquerdo



- Maior chave de uma BST
  - Elemento mais à direita
  - Avançar para a direita

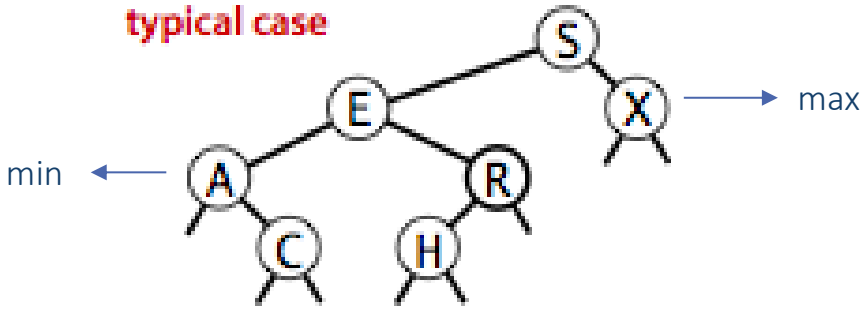
    Até encontrar um nó sem filho direito

- Nem sequer precisamos de fazer comparações!!!!!

Min e Max

```
public Key min()
{
   if(this.root == null) return null;
   return min(this.root);
}

private Key min(Node n)
{
   if(n.left == null) return n.key;
   return min(n.left);
}
```



Floor e *Ceiling*

- *Floor*(k)
  - Chave da árvore mais perto "por baixo"

    Qual a chave na árvore imediatamente antes de k, ou k?

    Maior chave *c*, tal que *c <= k*

- *Ceiling(k)*
  - *Chave da árvore mais perto "por* cima"

    Qual a chave na árvore imediatamente a seguir a k, ou k?

    Menor chave c, tal que c >= k

Floor(k)

- Maior chave c, tal que c <= k
  - Comparar k com chave do nó n
  - Se k = nó.c c <= k ✓

    Encontrámos a chave, retornar
  - *Se k* < nó.c c <= k ✗

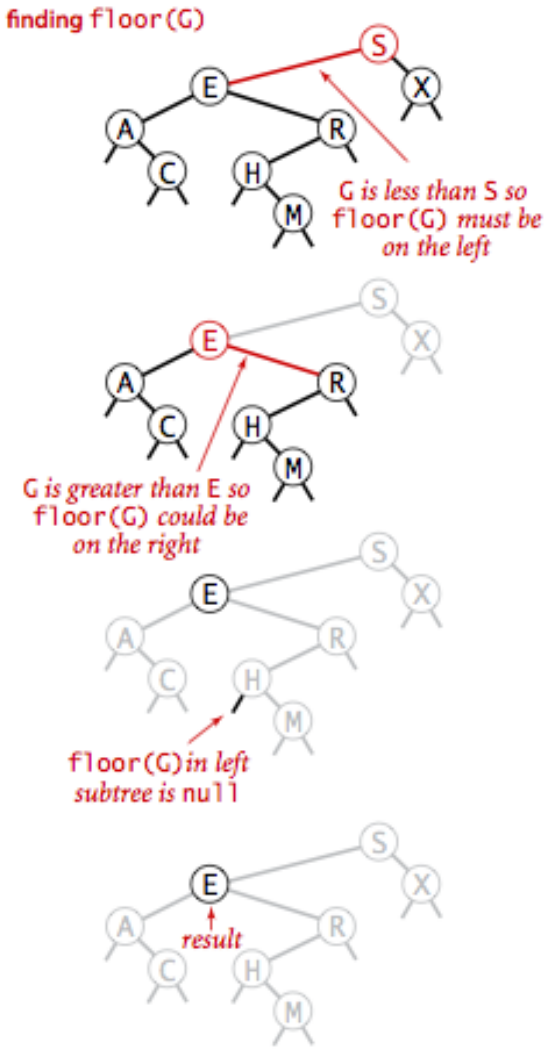
    Este não é candidato válido

    Continuar à procura para a esquerda
  - Se k > nó.c c <= k ✓

    Encontrámos um candidato possível!

    Mas pode haver outro melhor à direita

    Procurar à direita, mas se não encontrarmos, retornar nó.c



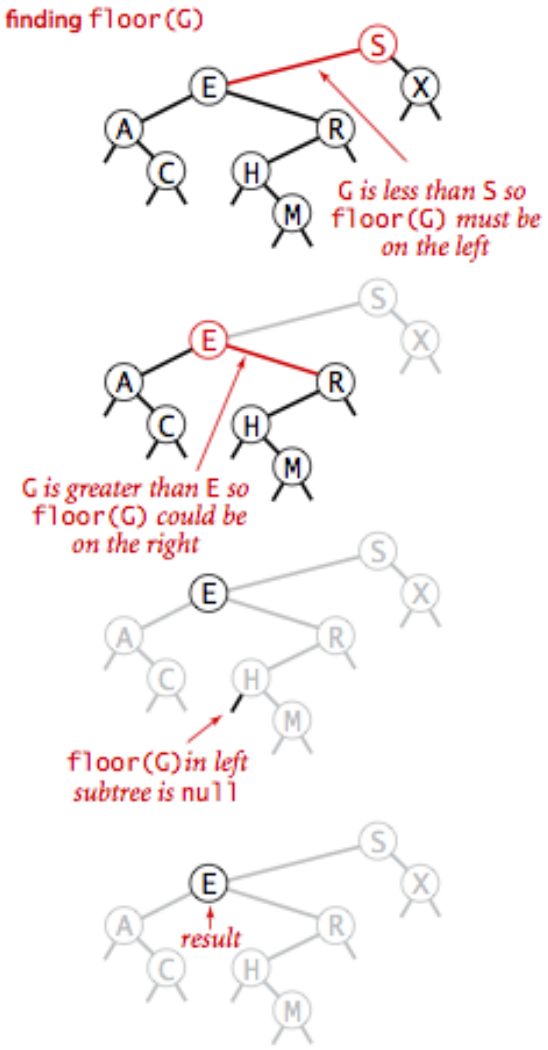
Computing the floor function

Floor(k)

```
public Key floor(Key k)
{
    Node n = floor(this.root, k);
    if(n == null) return null;
    return n.key;
}

private Node floor(Node n, Key k)
{
    if(n == null) return null;
    int cmp = k.compareTo(n.key);

    //found the key
    if(cmp == 0) return n;
    //not valid candidate, continue left
    if (cmp < 0) return floor(n.left, k);
    //else this node is a valid candidate
    //but we still need to check if there
    //is another option to the right
    Node otherOption = floor(n.right, k);
    //if there is another option, return it
    if(otherOption != null) return otherOption;
    //if not, return this node (valid candidate)
    return n;
}
```

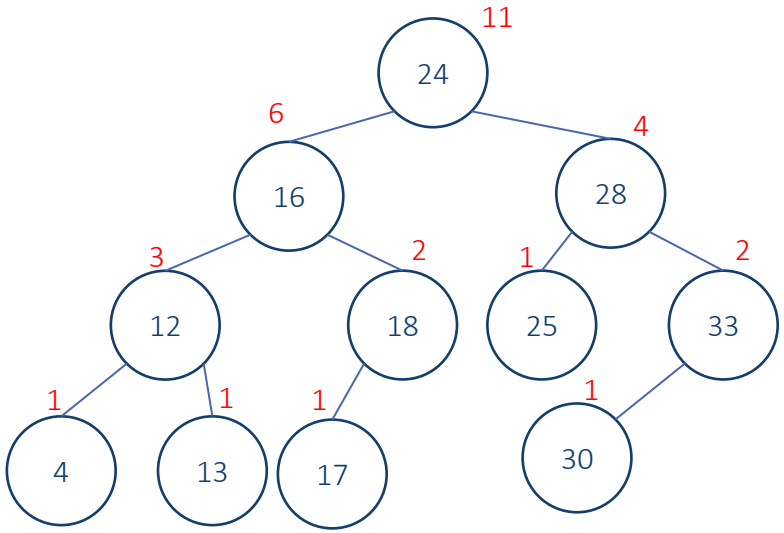


Computing the floor function

Rank(k)

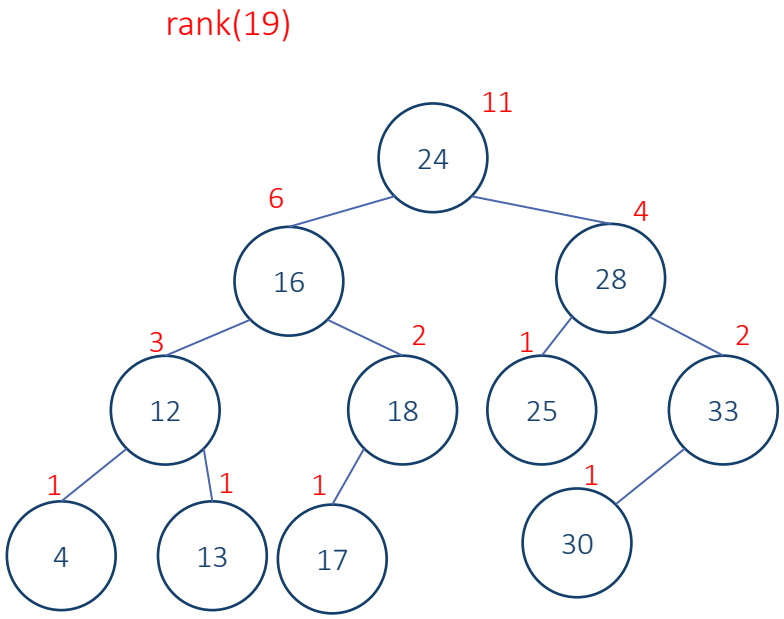
- Dada uma chave k, quantas chaves <k existem na árvore?

- Cada nó tem um contador para o número de nós na subárvore



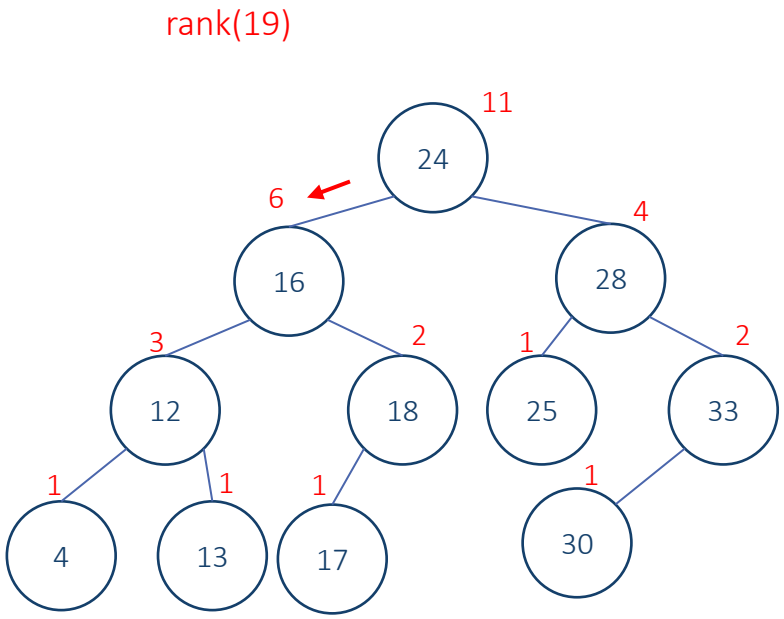
Rank(k)

- Dada uma chave k, quantas chaves <k existem na árvore?
- Se k = nó.chave
  - retornar o número de nós do filho esquerdo
- Se k < nó.chave
  - Continuar a procurar à esquerda
- Se k > nó.chave
  - Contar todos os da esquerda + 1 + procurar na direita



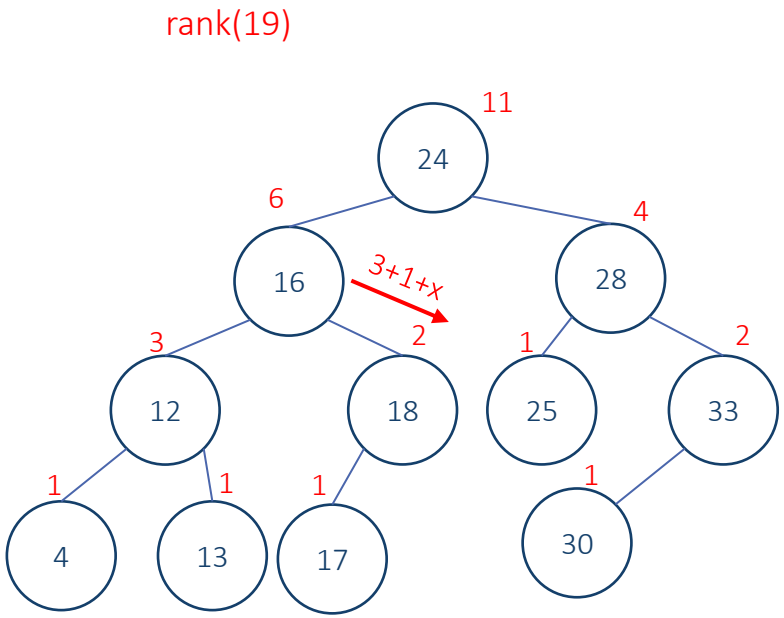
Rank(k)

- Dada uma chave k, quantas chaves <k existem na árvore?
- Se k = nó.chave
  - retornar o número de nós do filho esquerdo
- Se k < nó.chave
  - Continuar a procurar à esquerda
- Se k > nó.chave
  - Contar todos os da esquerda + 1 + procurar na direita



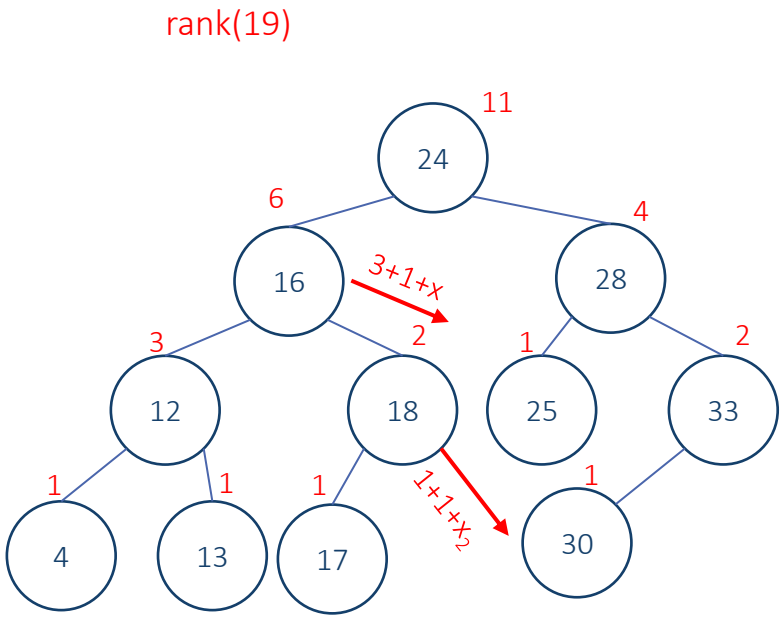
Rank(k)

- Dada uma chave k, quantas chaves <k existem na árvore?
- Se k = nó.chave
  - retornar o número de nós do filho esquerdo
- Se k < nó.chave
  - Continuar a procurar à esquerda
- Se k > nó.chave
  - Contar todos os da esquerda + 1 + procurar na direita



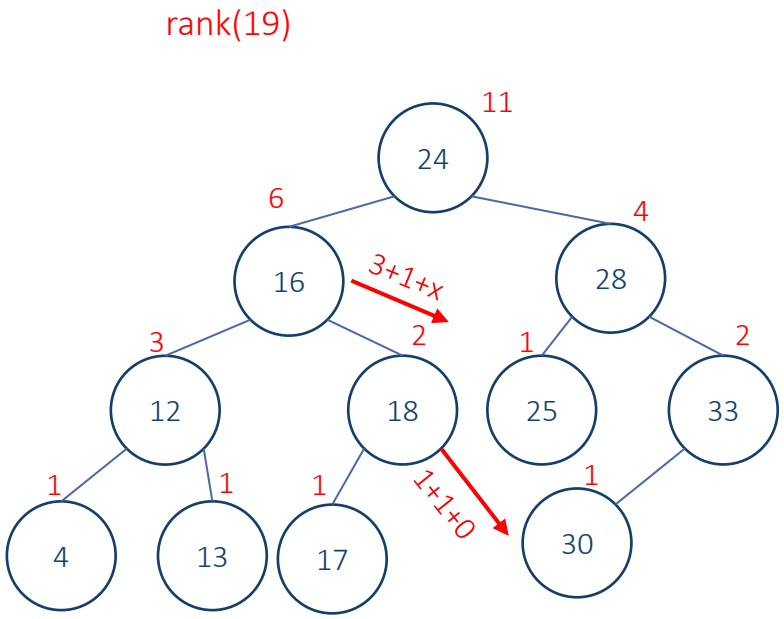
Rank(k)

- Dada uma chave k, quantas chaves <k existem na árvore?
- Se k = nó.chave
  - retornar o número de nós do filho esquerdo
- Se k < nó.chave
  - Continuar a procurar à esquerda
- Se k > nó.chave
  - Contar todos os da esquerda + 1 + procurar na direita



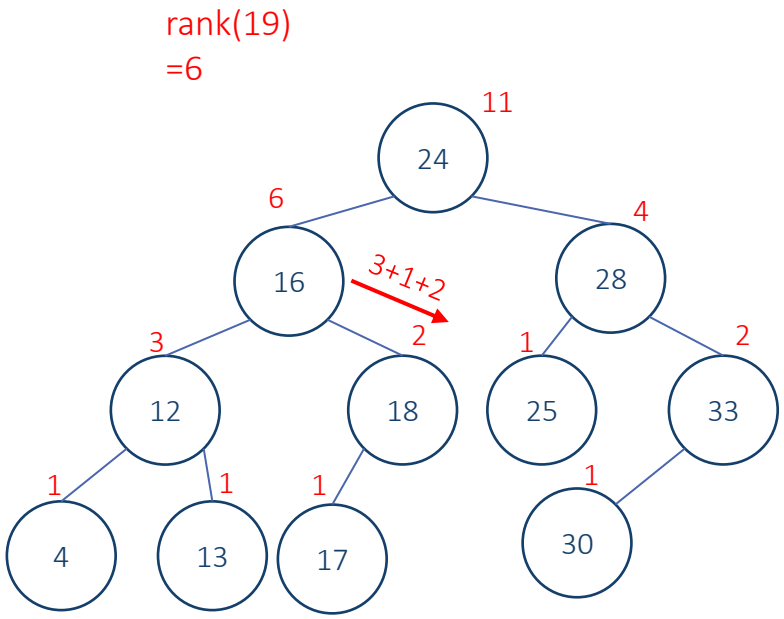
Rank(k)

- Dada uma chave k, quantas chaves <k existem na árvore?
- Se k = nó.chave
  - retornar o número de nós do filho esquerdo
- Se k < nó.chave
  - Continuar a procurar à esquerda
- Se k > nó.chave
  - Contar todos os da esquerda + 1 + procurar na direita



Rank(k)

- Dada uma chave k, quantas chaves <k existem na árvore?
- Se k = nó.chave
  - retornar o número de nós do filho esquerdo
- Se k < nó.chave
  - Continuar a procurar à esquerda
- Se k > nó.chave
  - Contar todos os da esquerda + 1 + procurar na direita



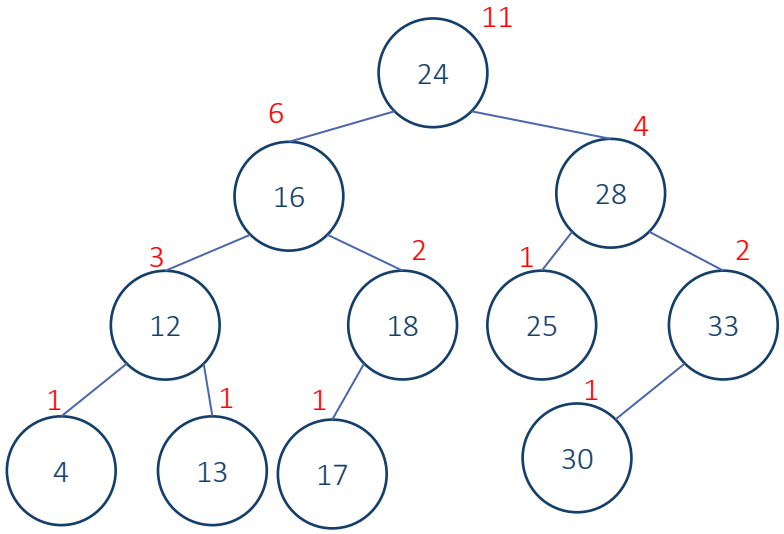
Rank(k)

```
private int size(Node n)
{
   if(n==null) return 0;
   else return n.size;
}

public int rank(Key k)
{
   return rank(this.root,k);
}

private int rank(Node n, Key k)
{
   if(n == null) return 0;
   int cmp = k.compareTo(n.key);
   if(cmp == 0) return size(n.left);
   if(cmp < 0) return rank(n.left, k);
   return 1 + size(n.left) + rank(n.right, k);
}
```

rank(19)



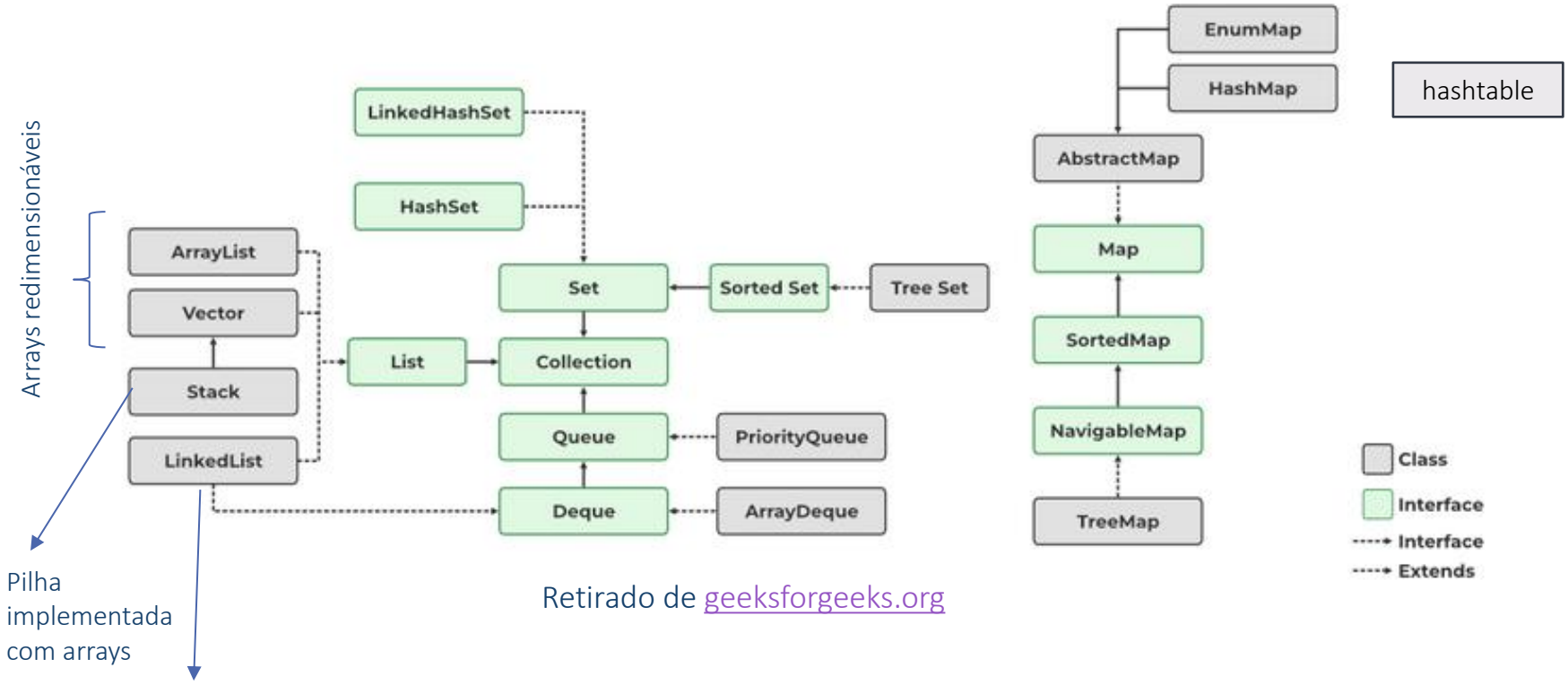
Coleções em Java

Coleções em Java

- Agora que já temos um conhecimento muito mais aprofundado sobre coleções

- Podemos perceber que tipos de coleções diferentes existem em java, e quais as vantagens e desvantagens de cada uma dela

Coleções em Java

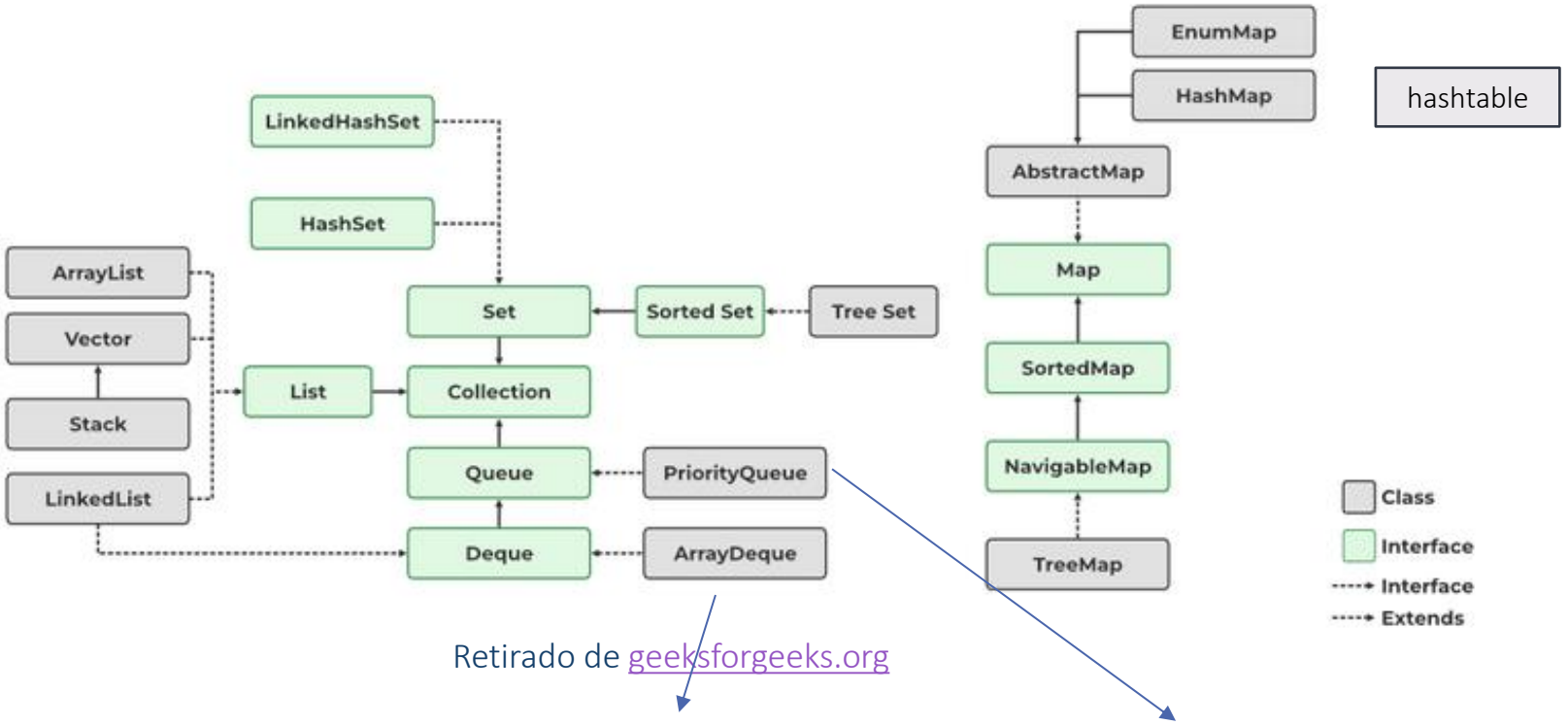


Retirado de geeksforgeeks.org

Pilha implementada com arrays

Lista ligada (nós com ponteiros)

Coleções em Java



Retirado de geeksforgeeks.org

Fila implementada com array

Um dequeue é uma double queue, uma fila que nos permite adicionar ou remover no fim ou no início da fila.
Eu pessoalmente não gosto disto.

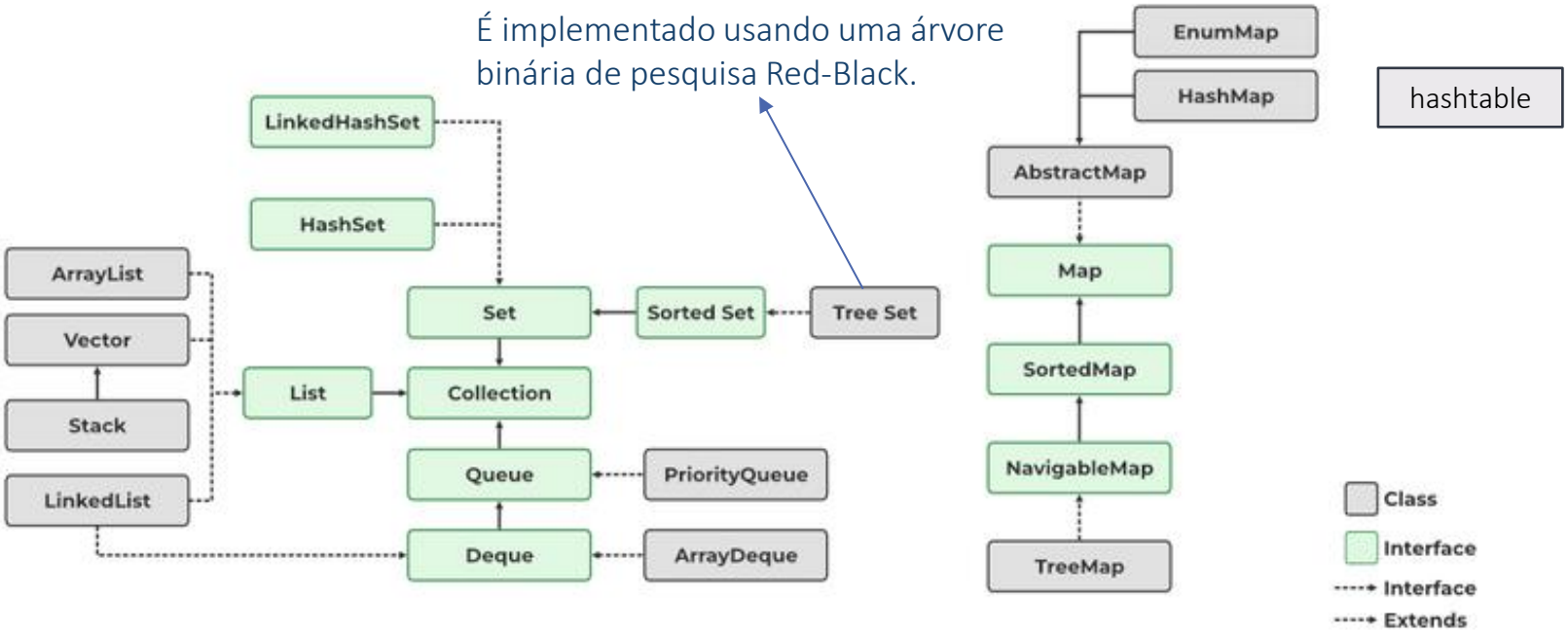
Fila prioritária implementada através de um amontoado (heap).

A implementação é feita usando um array para guardar os valores do heap.

Coleções em Java

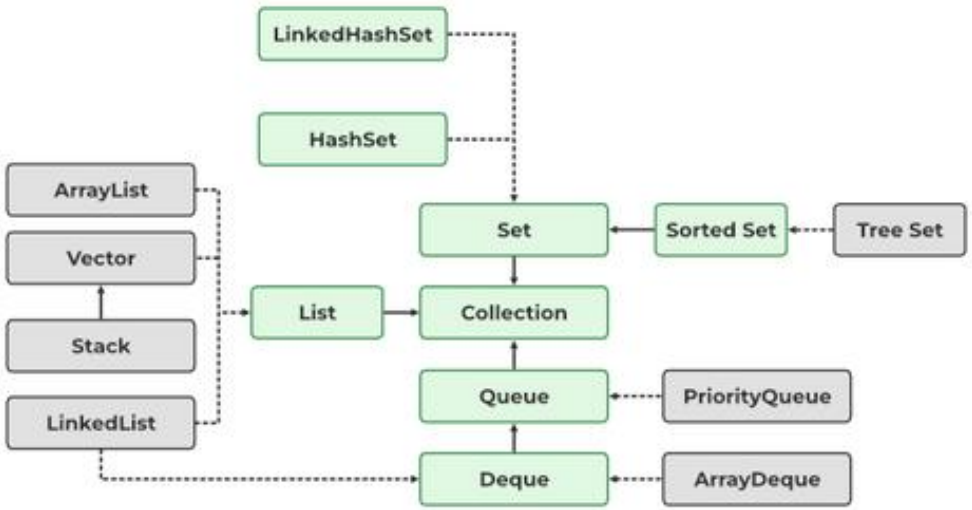
TreeSet<T> é uma árvore binária de pesquisa que só guarda chaves.

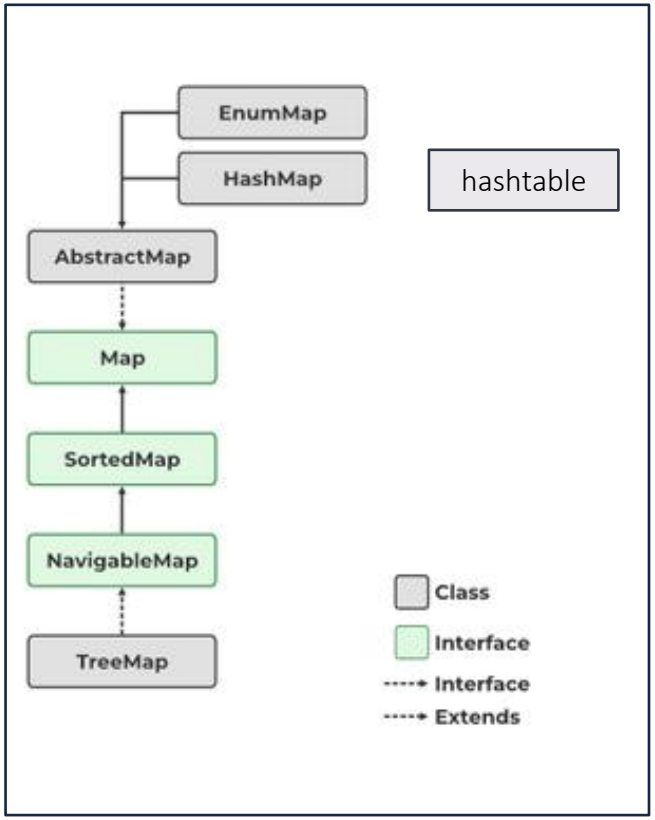
É implementado usando uma árvore binária de pesquisa Red-Black.



Retirado de geeksforgeeks.org

Coleções em Java

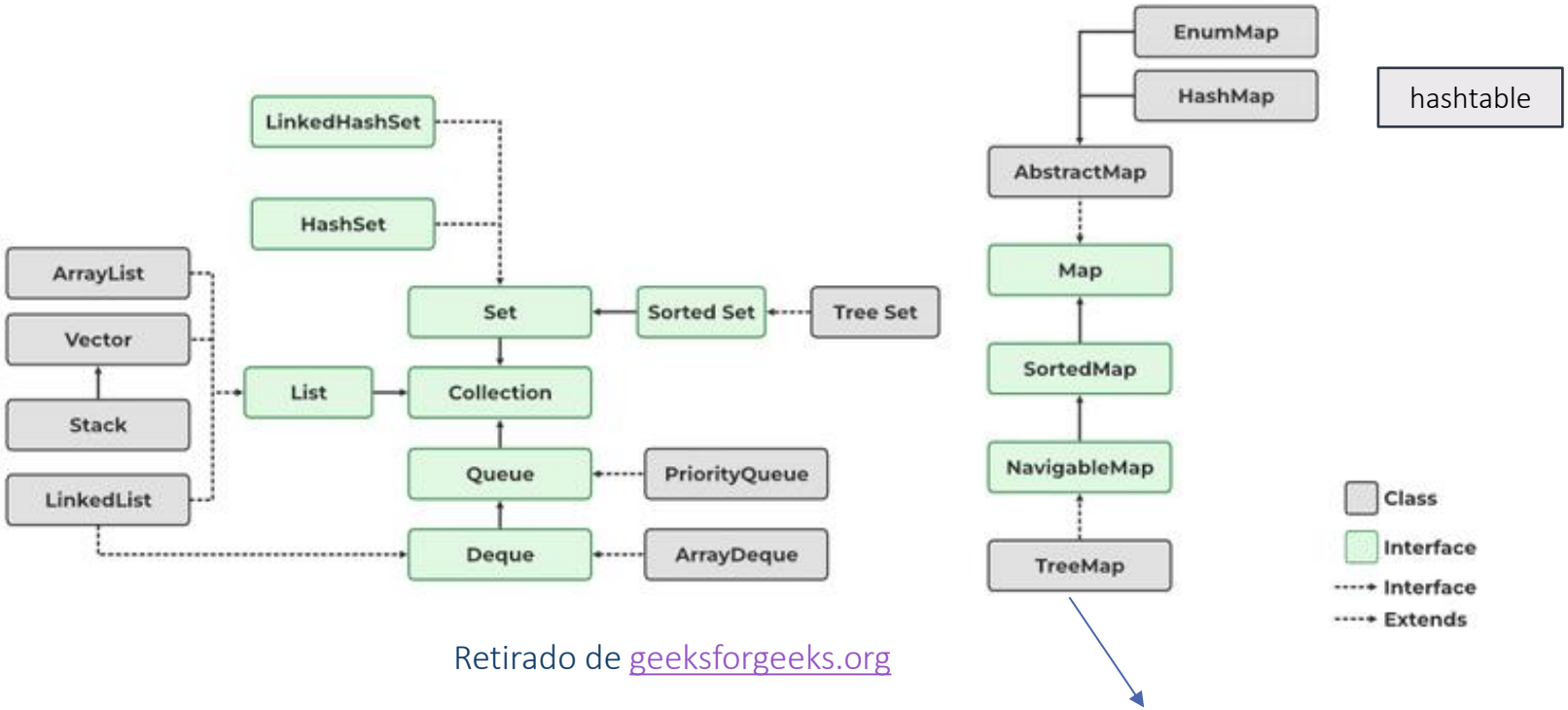




Retirado de geeksforgeeks.org

Observação:
A interface Map em Java corresponde ao conceito de Tabela de Símbolos:
Uma estrutura de dados que guarda pares de <chave,valor> e que usa as chaves para guardar e procurar.

Coleções em Java



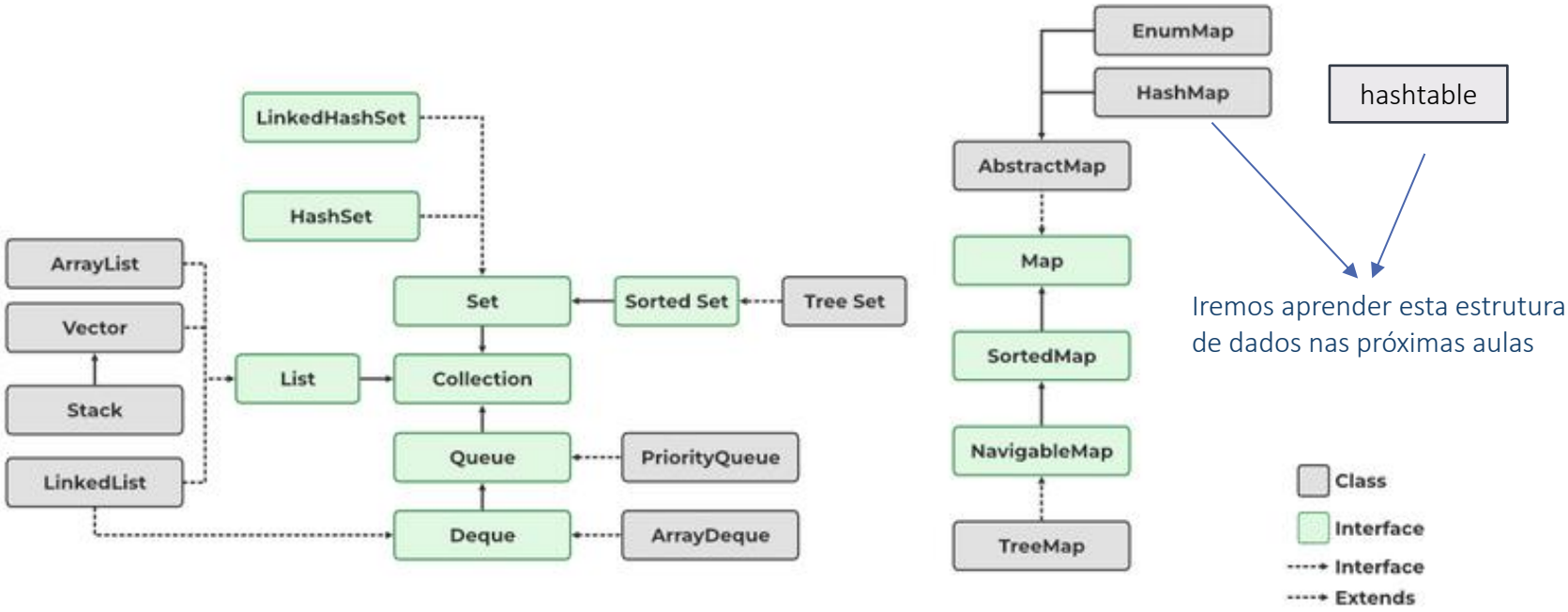
Retirado de geeksforgeeks.org

TreeMap<Key,Value> é uma tabela de símbolos ordenada.

É implementada usando uma árvore binária de pesquisa Red-Black.

É igual a um TreeSet, mas guarda um valor para além da chave.

Coleções em Java



Retirado de geeksforgeeks.org

Utilização de coleções

De forma eficiente

- O conhecimento sobre o funcionamento das coleções
  - Conhecimento sobre a sua complexidade temporal

- Permite-nos implementar código mais eficiente

Exemplo

```
public static List<String> getTeams(List<Submission> submissions)
{
    List<String> teams = new ArrayList<String>();

    for(Submission s : submissions)
    {
        if(!teams.contains(s.getNomeEquipa()))
        {
            teams.add(s.getNomeEquipa());
        }
    }

    teams.sort(null);
    return teams;
}
```

Dada uma lista de submissões (que entre outros dados contém o nome da equipa que fez a submissão),

este método devolve uma lista com todos os nomes de equipas, sem nomes duplicados, e a lista devolvida está ordenada.

Esta implementação não é muito eficiente.

Exemplo Projeto

```
public static List<String> getTeams(List<Submission> submissions)
{
    List<String> teams = new ArrayList<String>();
    for(Submission s : submissions)
    {
        if(!teams.contains(s.getNomeEquipa()))
        {
            teams.add(s.getNomeEquipa());
        }
    }
    teams.sort(null);
    return teams;
}
```

O(1)

O(n)

O(n)

O(1)

$O(n^2)$

O(n log$_2$ n)

Esta é na realidade uma estimativa grosseira,

Uma aproximação tilde mais correta seria: $\sim\frac{n^2}{2}$

$T(n) = O(n^2)$

Versão com TreeSet

```
public static List<String> getTeams(List<Submission> submissions)
{
    TreeSet<String> teams = new TreeSet<>();

    for(Submission s : submissions)
    {
        teams.add(s.getNomeEquipa());
    }

    return teams.stream().toList();
}
```

Não preciso de verificar se existe, porque pela definição de conjunto, o elemento só é adicionado se não existir

Não preciso de **ordenar**, porque o **TreeSet já está ordenado**.

Versão com TreeSet

```
public static List<String> getTeams2(List<Submission> submissions)
{
    TreeSet<String> teams = new TreeSet<>();

    for(Submission s : submissions)
    {
        teams.add(s.getNomeEquipa());
    }

    return teams.stream().toList();
}
```

O(1)

O(n)

$O(\log_2 n)$

$O(n \log_2 n)$

O(n)

Se isto estiver bem implementado, é possível ser feito em tempo constante, "mascarando" uma árvore como lista

Alternativamente, se declarasse este método como retornando uma Collection<String> não precisaria sequer de transformar numa lista, e bastava retornar a árvore.

De qualquer forma, não tem impacto na complexidade assimptótica do método

$$T(n) = O(n \log_2 n)$$