



IPS Instituto  
Politécnico de Setúbal  
**Escola Superior de  
Tecnologia de Setúbal**



Programação Avançada 2021-22

**[3e] ADT Graph | Estruturas de dados e implementação**

Bruno Silva, Patrícia Macedo

# Sumário



- Grafos | Estruturas de dados
  - Lista de arestas;
  - Matriz de adjacências;
  - Lista de adjacências.
- ADT Graph
  - Abordagem das implementações;
  - Estruturas de dados em Java.
- Exercícios | em aula 
- Consolidação | trabalho autónomo 

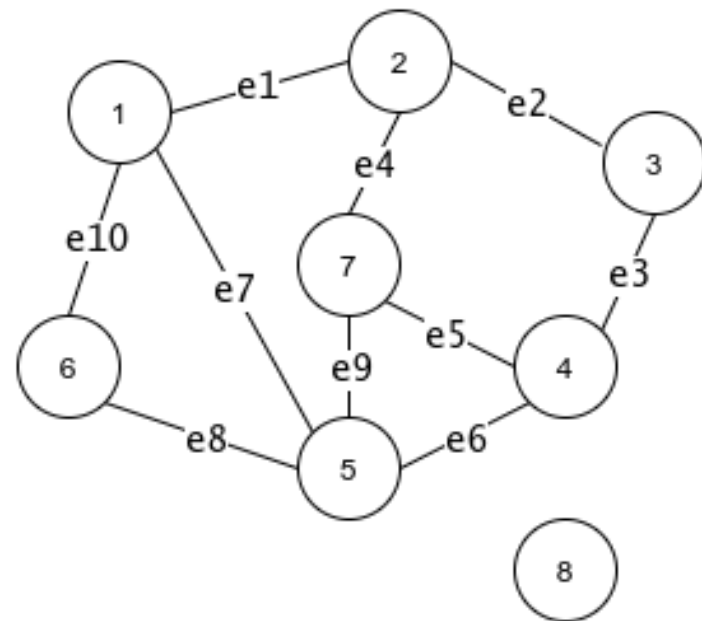
# Sobre implementar um grafo

Que informação tem de ser guardada para representar um grafo?

- sobre cada vértice?
- sobre cada aresta?

Que tipo de questões devem ser respondidas rapidamente:

- acerca de um vértice?
- acerca das suas arestas incidentes / vértices adjacentes?
- acerca de "caminhos"?
- acerca de que arestas existem no grafo?



# Grafos | Estruturas de dados

Existem 3 estruturas de dados tradicionais na implementação de grafos:

1. Lista de arestas
2. Matriz de adjacências
3. Lista de adjacências

🤔 Como sempre, a escolha da estrutura de dados terá impacto quer na **complexidade espacial** da implementação, quer na **complexidade algorítmica** das operações.

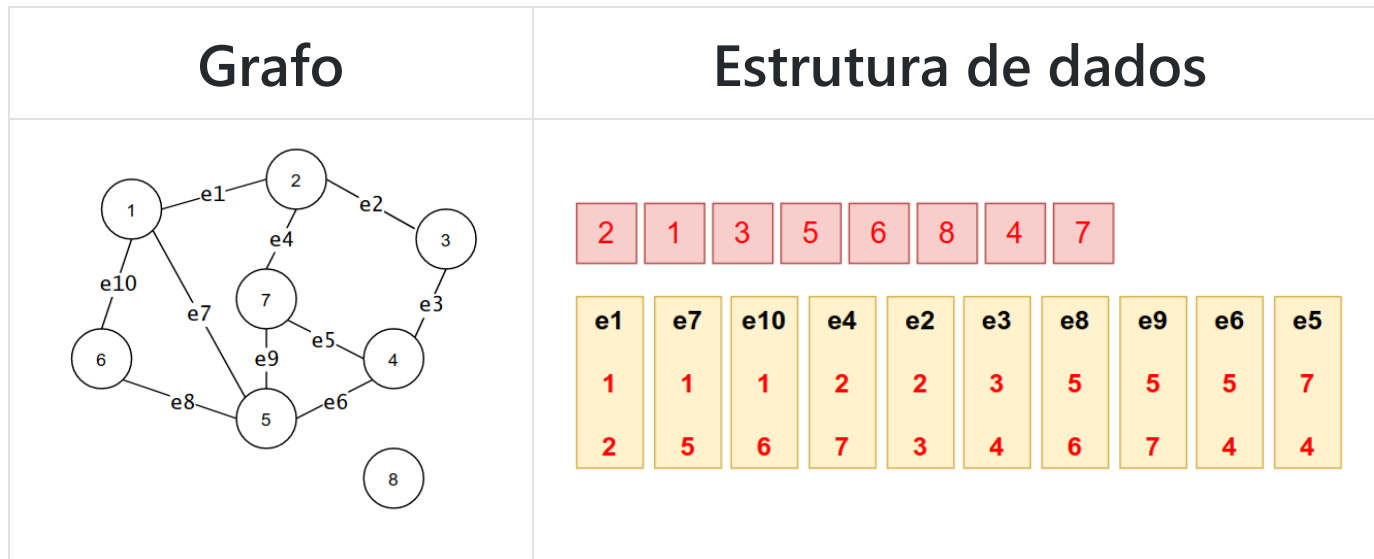


## ! Atenção:

- As estruturas de dados a seguir ilustradas podem carecer de adaptação para *grafos orientados* (dígrafos) e/ou com arestas paralelas.
- **Discuta sempre** que adaptações julga necessárias nestes casos.

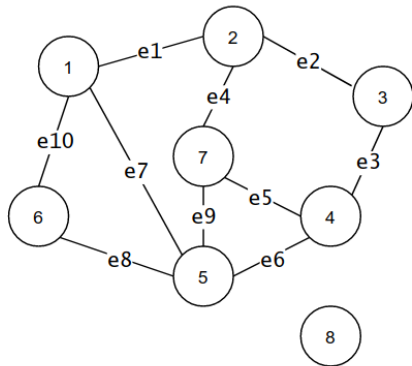
# Estrutura de dados | Lista de Arestas

🔧 Uma *coleção* de todas as arestas + uma coleção de todos os vértices do grafo (senão como repr. um vértice isolado?)



- 😊 fácil iterar sobre todas as arestas e vértices;
- 😞 difícil saber se existe uma aresta entre  $v$  e  $w$
- 😞 difícil saber as arestas incidentes de um vértice  $v$  (o seu grau)

# Estrutura de dados | Lista de Arestas

Grafo	Estrutura de dados																																						
	<table><tr><td>2</td><td>1</td><td>3</td><td>5</td><td>6</td><td>8</td><td>4</td><td>7</td></tr><tr><td>e1</td><td>e7</td><td>e10</td><td>e4</td><td>e2</td><td>e3</td><td>e8</td><td>e9</td><td>e6</td><td>e5</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>5</td><td>5</td><td>5</td><td>7</td></tr><tr><td>2</td><td>5</td><td>6</td><td>7</td><td>3</td><td>4</td><td>6</td><td>7</td><td>4</td><td>4</td></tr></table>	2	1	3	5	6	8	4	7	e1	e7	e10	e4	e2	e3	e8	e9	e6	e5	1	1	1	2	2	3	5	5	5	7	2	5	6	7	3	4	6	7	4	4
2	1	3	5	6	8	4	7																																
e1	e7	e10	e4	e2	e3	e8	e9	e6	e5																														
1	1	1	2	2	3	5	5	5	7																														
2	5	6	7	3	4	6	7	4	4																														

Em relação à informação na estrutura de dados:

- ? Como calcular o grau de um vértice?
- ? Como verificar se existe uma aresta entre  $v$  e  $w$  (adjacentes)?
- ? Como saber o conjunto de vértices existentes?

# Estrutura de dados | Matriz de Adjacências

🔧 Uma matriz  $n \times n$ , onde  $n$  é o número de vértices.

- a entrada  $a_{vw}$  indica se existe/que aresta liga o vértice  $v$  ao vértice  $w$ .

# Grafo

```

graph LR
    1 ---|e1| 2
    2 ---|e2| 3
    3 ---|e3| 4
    4 ---|e6| 5
    5 ---|e8| 6
    6 ---|e10| 1
    1 ---|e7| 7
    7 ---|e4| 2
    7 ---|e5| 4
    7 ---|e9| 5
    8
  
```

# Estrutura de dados

	1	2	3	4	5	6	7	8
1		e1			e7	e10		
2	e1		e2				e4	
3		e2		e3				
4			e3		e6		e5	
5	e7			e6		e8	e9	
6	e10				e8			
7		e4		e5	e9			
8								

😊 fácil saber se existe uma aresta entre  $v$  e  $w$

😞 consome muita memória em grafos com poucas arestas

😞 difícil de "gerir" (e.g., adicionar/remover vértices)



# Estrutura de dados | Matriz de Adjacências

# Grafo

```
graph LR; 1 ---|e1| 2; 2 ---|e2| 3; 3 ---|e3| 4; 4 ---|e6| 5; 5 ---|e8| 6; 6 ---|e10| 1; 1 ---|e7| 7; 7 ---|e4| 2; 7 ---|e5| 4; 7 ---|e9| 5;
```

# Estrutura de dados

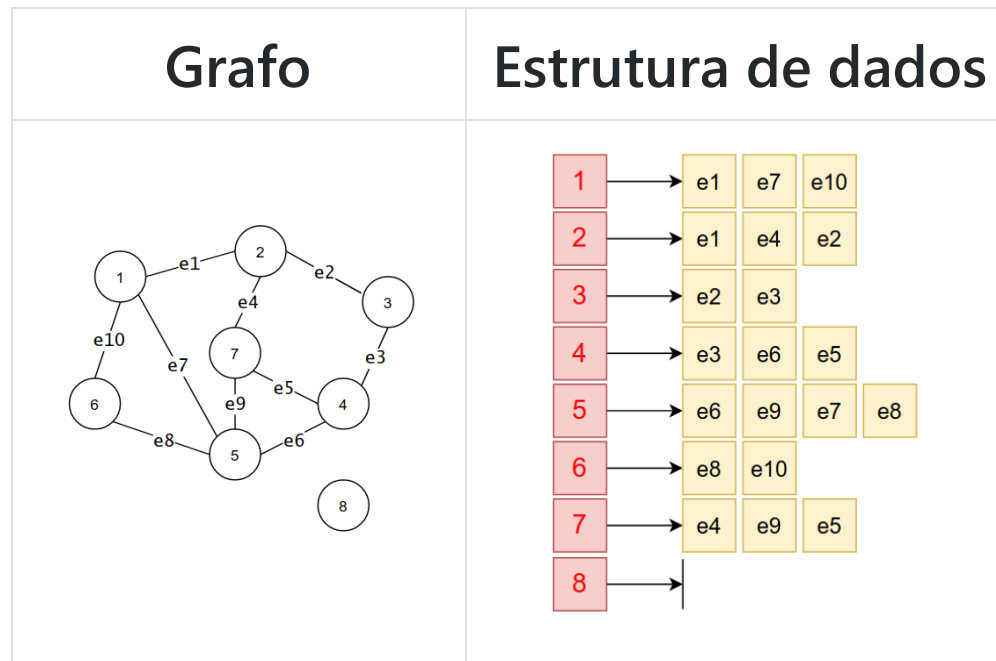
	1	2	3	4	5	6	7	8
1		e1			e7	e10		
2	e1		e2				e4	
3		e2		e3				
4			e3		e6		e5	
5	e7			e6		e8	e9	
6	e10				e8			
7		e4		e5	e9			
8								

Em relação à informação na estrutura de dados:

- ? Como calcular o grau de um vértice?
- ? Como verificar se existe uma aresta entre  $v$  e  $w$  (adjacentes)?
- ? Como saber o conjunto de arestas existentes?

# Estrutura de dados | Lista de Adjacências

🔧 Uma *coleção* de vértices  $\rightarrow$  cada vértice guarda a sua lista de arestas incidentes.

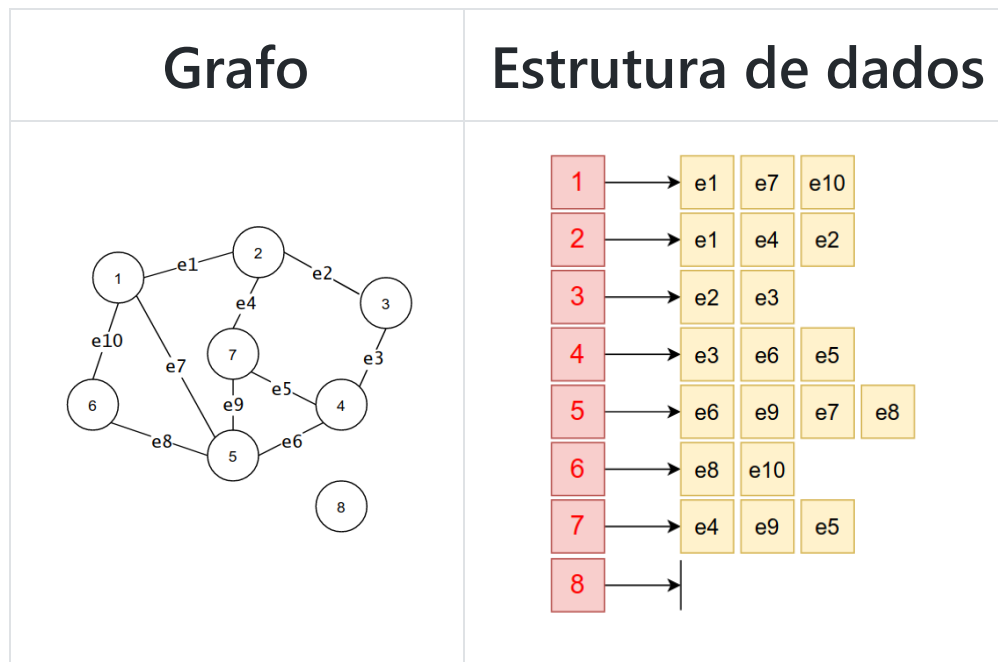


😊 fácil adicionar vértices e arestas

😊 fácil saber arestas incidentes a  $v$

😞 difícil saber se existe uma aresta entre  $v$  e  $w$

# Estrutura de dados | Lista de Adjacências



Em relação à informação na estrutura de dados:

- ? Como calcular o grau de um vértice?
- ? Como verificar se existe uma aresta entre  $v$  e  $w$  (adjacentes)?
- ? Como saber o conjunto de arestas existentes?

# Estrutura de dados | Lista de Adjacências

! Nota acerca do nome desta estrutura de dados:

- se não for requerido uma aresta guardar informação, pode guardar-se em alternativa a lista de vértices adjacentes - daí o nome "original" *lista de adjacências*.
  - a estrutura de dados ilustrada é uma variação desta e permite associar informação a uma aresta.
- neste caso as complexidades a seguir apresentadas para esta estrutura de dados seriam melhores; pode fazer este exercício autónomo de análise.

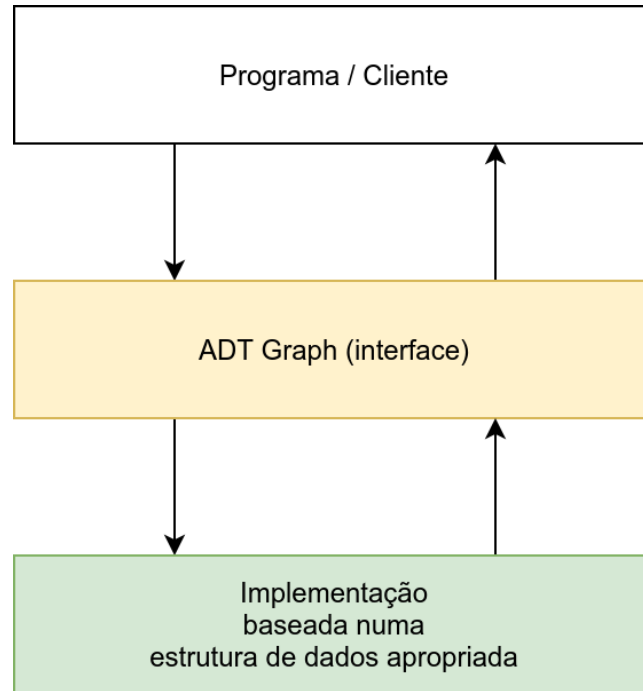
# Estruturas de dados | Comparação

Tabela de Complexidades  $\mathcal{O}()$ , sendo  $n$  e  $m$  o núm. de vértices e arestas, respetivamente:

	Lista de arestas	Lista de adj.	Matriz de adj.
<b>Complexidade espacial</b>	$n + m$	$n + m$	$n^2$
Encontrar todos os vértices adjacentes a $v$	$m$	$\deg(v) * m$	$n$
Det. se $v$ é adjacente a $w$	$m$	$\deg(v) * \deg(w)$	$1$
Inserir um vértice	$1$	$1$	$n^2$
Inserir uma aresta	$1$	$1$	$1$
Remover um vértice $v$	$m$	$\deg(v) * n$	$n^2$
Remover uma aresta	$m$	$n$	$1$

💡  $\deg(x)$  - grau do vértice  $x$

# ADT Graph (recap.) | *tipo abstrato*



💡 (1) O cliente apenas conhece e trabalha com a *interface*; não conhece detalhes da implementação - apenas que respeita a especificação da interface. (2) Pode haver implementações diferentes disponíveis, o programa não necessita de modificações.

# ADT Graph (recap.) | *interface*

```
public interface Graph<V, E> {  
    public int numVertices();  
    public int numEdges();  
    public Collection<Vertex<V>> vertices();  
    public Collection<Edge<E, V>> edges();  
    public Collection<Edge<E, V>> incidentEdges(Vertex<V> v)  
        throws InvalidVertexException;  
    public Vertex<V> opposite(Vertex<V> v, Edge<E, V> e)  
        throws InvalidVertexException, InvalidEdgeException;  
    public boolean areAdjacent(Vertex<V> u, Vertex<V> v)  
        throws InvalidVertexException;  
    public Vertex<V> insertVertex(V vElement)  
        throws InvalidVertexException;  
    public Edge<E, V> insertEdge(Vertex<V> u, Vertex<V> v, E edgeElement)  
        throws InvalidVertexException, InvalidEdgeException;  
    public V removeVertex(Vertex<V> v)  
        throws InvalidVertexException;  
    public E removeEdge(Edge<E, V> e)  
        throws InvalidEdgeException;  
    //...  
}
```

! Note o uso dos *tipos/interfaces* `Vertex<V>` e `Edge<E,V>` .

## ADT Graph (recap.) | *interface*

⚠ Sendo `Vertex<V>` e `Edge<E,V>` *interfaces*, estas terão que ter uma "expressão concreta" - através de **tipos de dados** conhecidos apenas pela implementação.

- E.g, um **vértice** numa qualquer implementação será representado por uma instância de uma classe apropriada que guarda a informação respetiva desse vértice.



# ADT Graph | *implementação*

- Uma classe que implementa a interface `Graph<V,E>` ;
- A estrutura de dados será codificada (conjuntamente):
  - **nos atributos da classe**, e;
  - **nos tipos concretos que descrevem um vértice e uma aresta**, definidos através de *inner classes* - privadas e desconhecidas no "exterior"; implementam as interface `Vertex<V>` e `Edge<E,V>`
    - É uma forma de não se expor ao "exterior" a representação interna destes tipos, mas permitir que as *referências* desses objetos/instâncias sejam comunicadas entre o cliente e a implementação na invocação de operações.
- As implementações das operações, determinadas pela interface `Graph` , serão feitas à custa da manipulação da estrutura de dados.



## ! Observação:

Imaginando que quero guardar uma *coleção de vértices*...

As *coleções* mencionadas anteriormente podem ser:

- Listas simples (e.g., `List< Vertex<V> > vertices;` ), ou então;
  - 🙅 Para verificar a existência de um determinado vértice tenho de "varrer" a lista;
- Dicionários (e.g., `Map< V, Vertex<V> > vertices;` ).
  - Assumindo que não existem "vértices duplicados", mantemos registo de qual vértice guarda que elemento;
  - 👍 Sabendo o *elemento* a pesquisar, obtenho imediatamente o vértice associado.

# Estruturas de dados em Java + Implementação

Para cada uma das estruturas de dados apresentada vamos:

- Ver como a codificar na linguagem *Java*;
- Exemplificar a implementação do método `boolean areAdjacent(Vertex<V> v, Vertex<V> w)`.

# Estruturas de dados em Java | Lista de arestas

2	1	3	5	6	8	4	7		
e1	e7	e10	e4	e2	e3	e8	e9	e6	e5
1	1	1	2	2	3	5	5	5	7
2	5	6	7	3	4	6	7	4	4

Estrutura de dados definida por:

- Coleção de **vértices**;
- Coleção de **arestas**;
  - Cada aresta conhece os vértices que liga;

# Estruturas de dados em Java | Lista de arestas

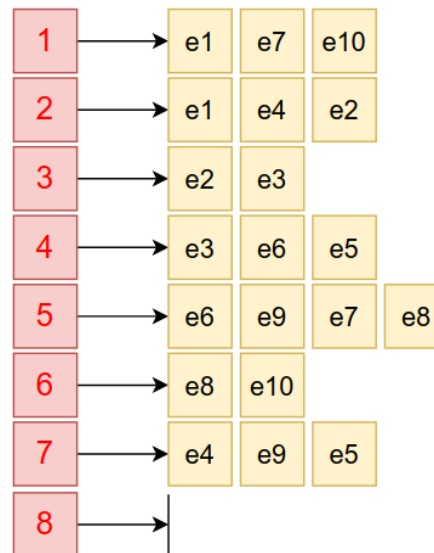
```
public class GraphEdgeList<V,E> implements Graph<V,E> {  
  
    private Map<V, Vertex<V>> vertices; //coleção dos vértices  
    private Map<E, Edge<E,V>> edges; //coleção das arestas  
  
    public GraphEdgeList() {  
        this.vertices = new HashMap<>();  
        this.edges = new HashMap<>();  
    }  
  
    private class MyVertex implements Vertex<V> {  
        private V element;  
        //...  
    }  
  
    private class MyEdge implements Edge<E, V> {  
        private E element;  
        private MyVertex v1, v2; //vértices que liga  
        //...  
    }  
    //...  
}
```

# Implementação em Java | Lista de arestas

```
public class GraphEdgeList<V,E> implements Graph<V,E> {  
  
    //...  
  
    @override  
    public boolean areAdjacent(Vertex<V> v, Vertex<V> w) {  
        /*TODO: validações */  
  
        /* Existe alguma aresta que ligue v e w ?*/  
        for(Edge<E, V> e : edges.values()) {  
            MyEdge edge = (MyEdge)e;  
            if(edge.v1 == v && edge.v2 == w || edge.v1 == w && edge.v2 == v) {  
                return true;  
            }  
        }  
  
        return false;  
    }  
}
```

? Complexidade?  $\rightarrow \mathcal{O}(m)$ , sendo  $m$  o número de arestas.

# Estruturas de dados em Java | Lista de adjacências



Estrutura de dados definida por:

- Coleção de **vértices**;
  - Cada vértice possui a sua **lista de arestas** incidentes;

# Estruturas de dados em Java | Lista de adjacências

```
public class GraphAdjacencyList<V,E> implements Graph<V,E> {  
  
    private Map<V, Vertex<V>> vertices; //coleção dos vértices  
  
    public GraphAdjacencyList() {  
        this.vertices = new HashMap<>();  
    }  
  
    private class MyVertex implements Vertex<V> {  
        private V element;  
        private List<Edge<E,V>> incidentEdges; //lista incidentes  
        //...  
    }  
  
    private class MyEdge implements Edge<E, V> {  
        private E element;  
        //...  
    }  
    //...  
}
```



# Implementação em Java | Lista de adjacências

```
public class GraphAdjacencyList<V,E> implements Graph<V,E> {  
  
    //...  
  
    @Override  
    public boolean areAdjacent(Vertex<V> v, Vertex<V> w) {  
        /*TODO: validações */  
  
        MyVertex myV = (MyVertex)v;  
        MyVertex myW = (MyVertex)w;  
  
        /* Existe uma aresta em comum nas listas de v e w ?*/  
        Set<Edge<E,V>> intersection = new HashSet<>(myV.incidentEdges);  
        intersection.retainAll(myW.incidentEdges);  
  
        return !intersection.isEmpty();  
    }  
}
```

? Complexidade?  $\rightarrow \mathcal{O}(\deg(v) \times \deg(w))$ , sendo  $\deg(v)$  e  $\deg(w)$  o número de arestas incidentes a  $v$  e  $w$ , respectivamente.

# Estruturas de dados em Java | Matriz de adjacências

	1	2	3	4	5	6	7	8
1		e1			e7	e10		
2	e1		e2				e4	
3		e2		e3				
4			e3		e6		e5	
5	e7			e6		e8	e9	
6	e10				e8			
7		e4		e5	e9			
8								

⚠ Para utilizar uma **matriz**, teríamos de "associar" *índices*  $\rightarrow$  vértices;

- Podemos "simplificar" e até poupar memória..

Estrutura de dados definida por:

- Dicionário de **vértices**  $\rightarrow$  { **vértice** : **aresta** } | "mapa de adjacências"
  - e.g., `adjacencyMap.get(V2).entrySet()` dá-nos os tuplos  
[ {V1,e1}, {V3,e2}, {V7,e4} ]

# Estruturas de dados em Java | Matriz de adjacências

```
public class GraphAdjacencyMatrix<V,E> implements Graph<V,E> {  
    private Map<Vertex<V>, Map<Vertex<V>, Edge<E,V>>> adjacencyMap;  
  
    public GraphAdjacencyMatrix() {  
        this.adjacencyMap = new HashMap<>();  
    }  
  
    private class MyVertex implements Vertex<V> {  
        private V element;  
        //...  
    }  
  
    private class MyEdge implements Edge<E, V> {  
        private E element;  
        //...  
    }  
    //...  
}
```

⚠ Esta codificação não permite arestas paralelas; para tal teria de ser:

```
Map<Vertex<V>, Map<Vertex<V>, List<Edge<E,V>>>> adjacencyMap;
```

# Implementação em Java | Matriz de adjacências

```
public class GraphAdjacencyList<V,E> implements Graph<V,E> {  
  
    //...  
  
    @override  
    public boolean areAdjacent(Vertex<V> v, Vertex<V> w) {  
        /*TODO: validações */  
  
        return adjacencyMap.get(v).containsKey(w);  
    }  
}
```

? Complexidade?  $\rightarrow \approx \mathcal{O}(1)$ , utilizando tabelas de dispersão (e.g., `HashMap` ).

# Exercícios | em aula

Faça o *clone* do projeto IntelliJ IDEA:

[https://github.com/estsetubal-pa-geral/ADTGraph\\_Implementation.git](https://github.com/estsetubal-pa-geral/ADTGraph_Implementation.git)

1. Para cada uma das 3 implementações, forneça o código de:

- `Vertex<V> insertVertex(V vElement)`
- `Edge<E, V> insertEdge(Vertex<V> u, Vertex<V> v, E edgeElement)`
- `Collection<Edge<E, V>> incidentEdges(Vertex<V> v)`

2. Teste o programa, variando a utilização das implementações.

---

**!** Os restantes métodos serão alvo de trabalho de consolidação (ver slide seguinte)

# Consolidação | trabalho autónomo

1. Termine a implementação da classe `GraphEdgeList` e finalize o programa ( `main` ) do repositório (i.e., os *TODO*)
  - Tem como referência a implementação completa desta classe no repositório do projeto ou repositório `SmartGraphJavaFX`  
<https://github.com/brunomnsilva/JavaFXSmartGraph>
2. Termine a implementação da classe `GraphEdgeList` → requerida pela *milestone* do projeto de época normal. Teste o programa.
3. [Opcional/desafio] Termine a implementação da classe `GraphAdjacencyMatrix` . Teste o programa.
4. Adicione os métodos seguintes à interface `Graph` e implemente-os:
  - `int degreeOf(Vertex<V> v) throws InvalidVertexException`
  - `Collection<Vertex<V>> adjacentVertices(Vertex<V> v) throws InvalidVertexException`