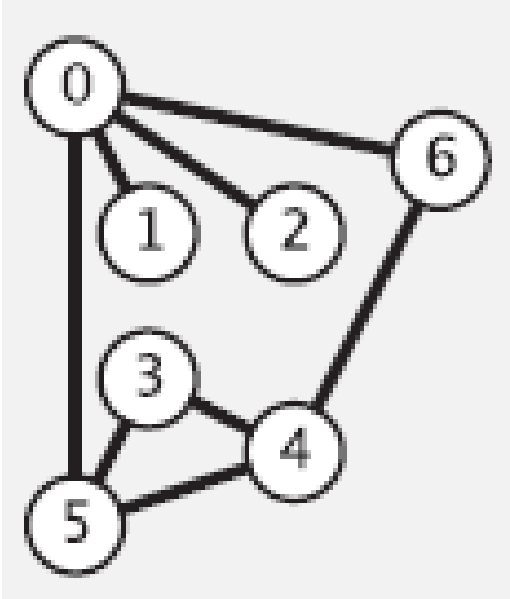
Grafy

Grafy nieskierowane i skierowane

Zagadnienia

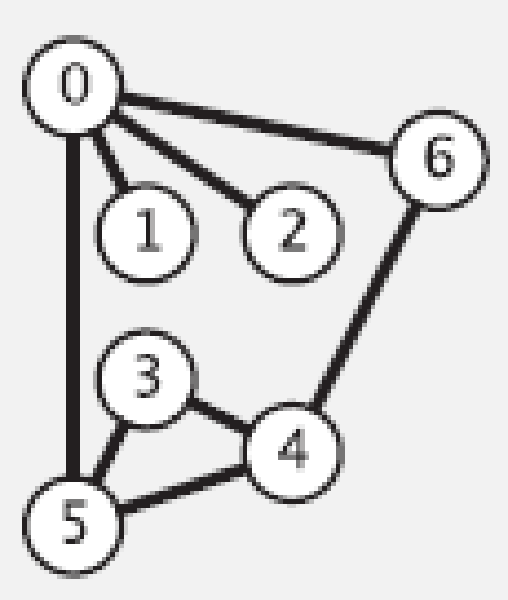
Graf $G = (V, E)$, gdzie V – zbiór wierzchołków, a E – zbiór krawędzi.



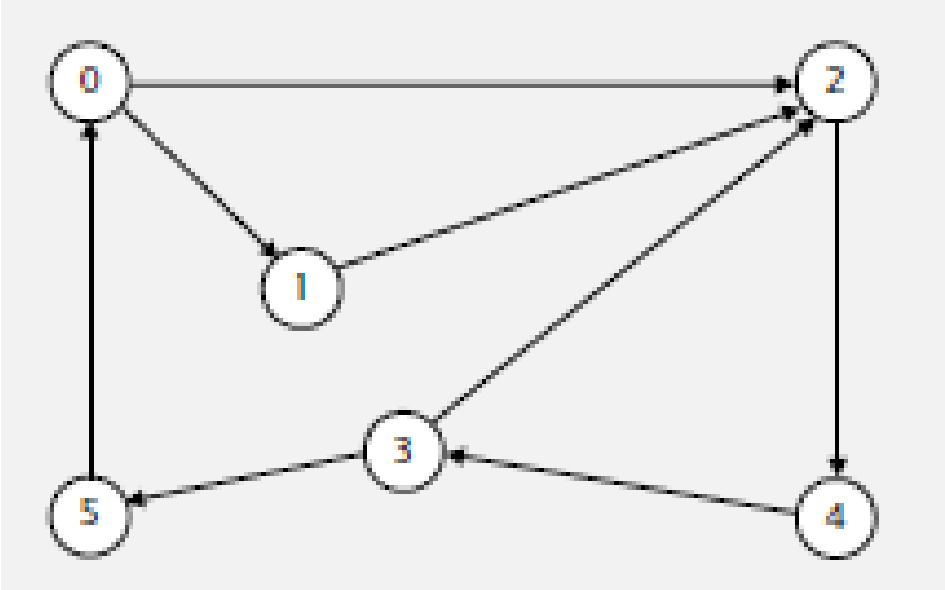
Zagadnienia

Graf może być:

nieskierowany



skierowany

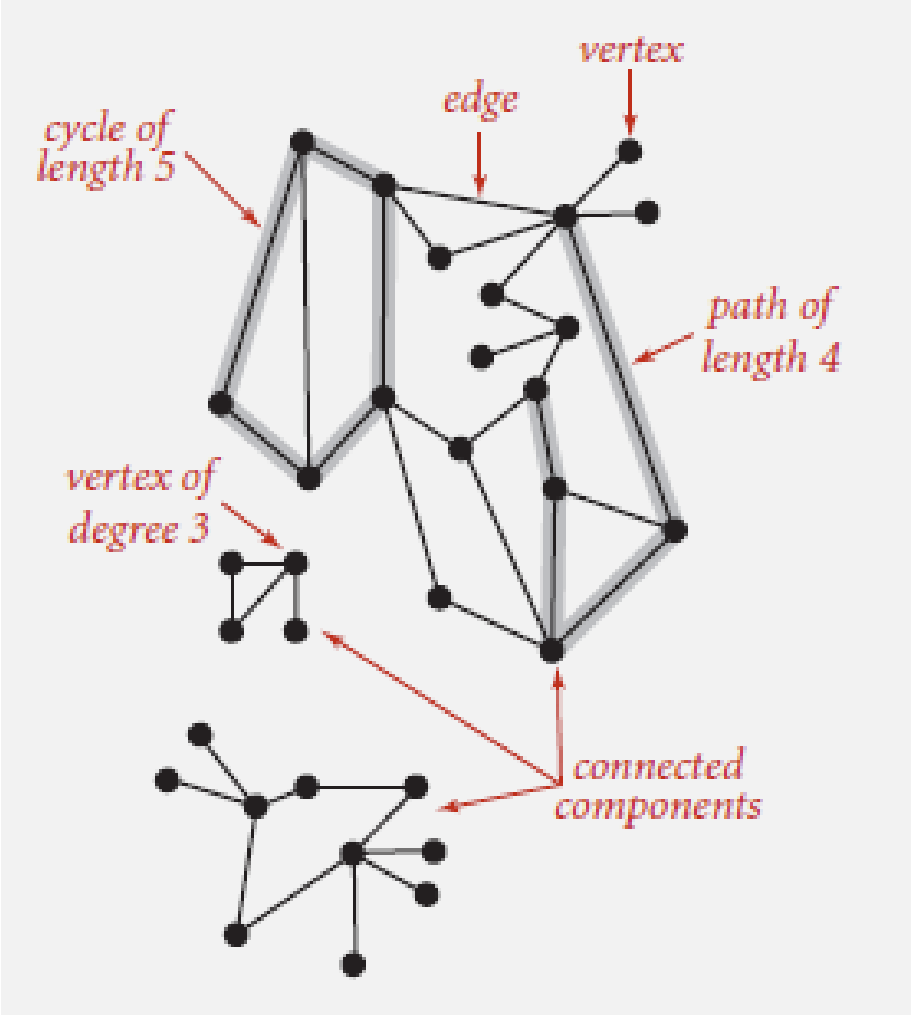


Graf nieskierowany

Ścieżka – sekwencja wierzchołków połączonych krawędziami.

Cykl – ścieżka, której początkiem i końcem jest ten sam wierzchołek.

Dwa wierzchołki są **połączone** jeśli istnieje ścieżka pomiędzy nimi.



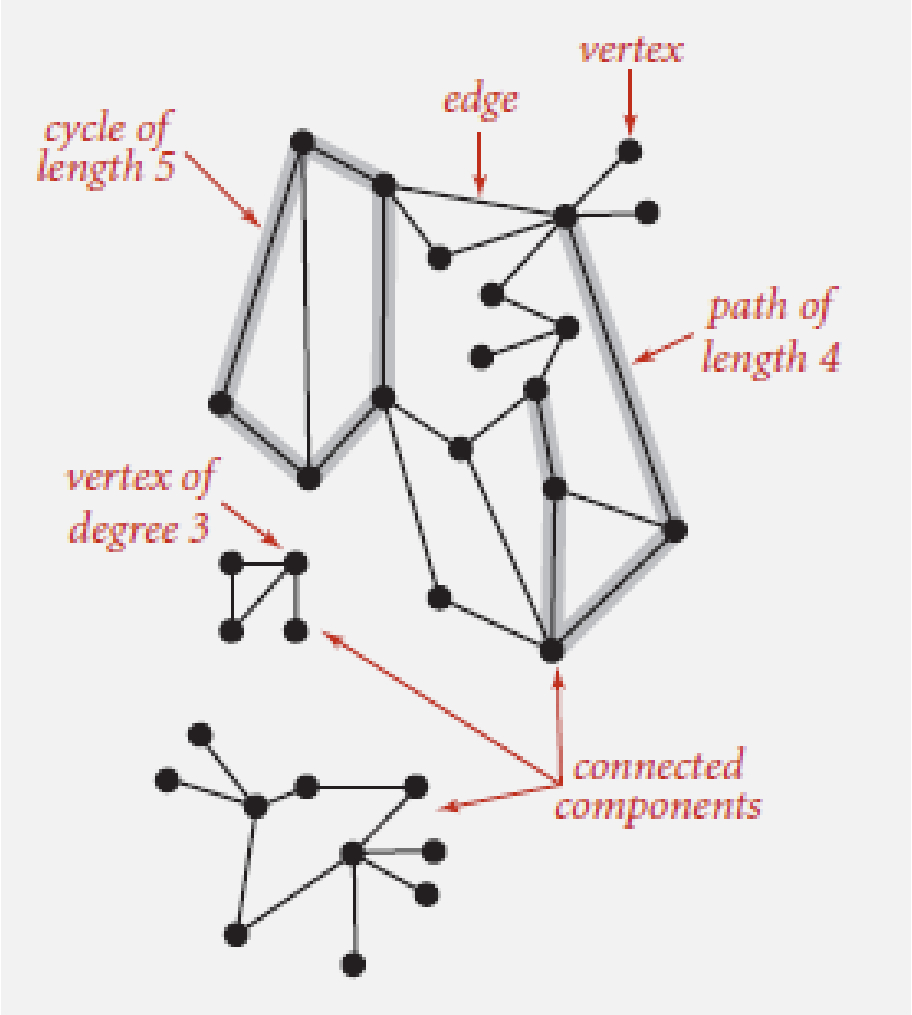
[SW-2018]

Graf nieskierowany

Długość ścieżki lub cyklu – liczba krawędzi wchodzących w ich skład.

Graf jest **spójny** jeśli istnieje ścieżka z każdego wierzchołka do każdego innego wierzchołka grafu.

Stopień wierzchołka *v* – liczba krawędzi połączonych z wierzchołkiem *v*

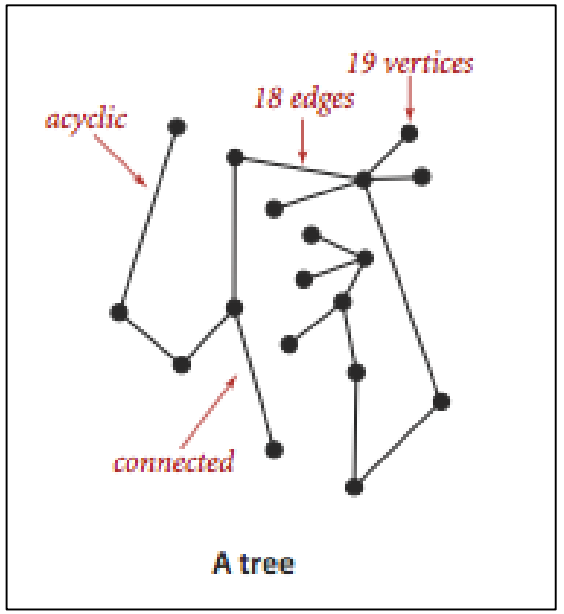


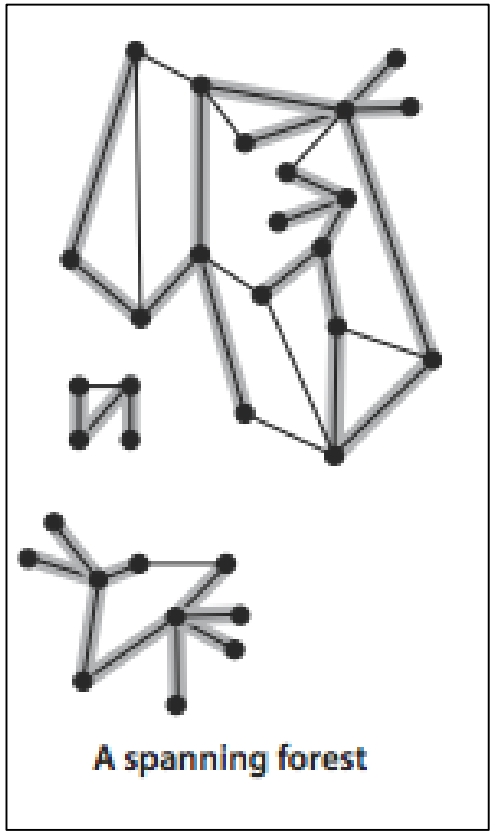
[SW-2018]

Graf nieskierowany

Drzewo – to acykliczny graf spójny.

Drzewo rozpinające dla grafu spójnego to podgraf składający się z wszystkich wierzchołków grafu i będący jednym drzewem.





[SW-2018]

Graf nieskierowany

Opis	Metoda
Konstruktor, konstruuje graf o liczbie wierzchołków V	`Graph(int V)`
Dodaje do grafu krawędź łączącą wierzchołki v i w	`void addEdge(int v, int w)`
Zwraca wierzchołki połączone z v	`Iterable<Integer> adj(int v)`
Zwraca liczbę wierzchołków	`int V()`
Zwraca liczbę krawędzi	`int E()`

Reprezentacje grafu

Przerywnik: Iteratory, Kolekcje z możliwością iterowania

W wielu aplikacjach zależy nam na przejściu (iterowaniu) przez całą kolekcję, czyli dotarciu po kolei do każdego elementu i wykonaniu pewnych operacji, niezależnie jak taka kolekcja jest zorganizowana.

Interfejs `Iterable`

(ma metodę, która zwraca `Iterator`)

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();
}
```

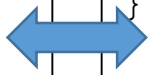
Reprezentacje grafu

Przerywnik: Iteratory, Kolekcje z możliwością iterowania

```
import java.util.Iterator;
public class Stack<Item>
        implements Iterable<Item>
{
…
}

Stack<String> stack = new Stack<String>();

for (String s : stack)
   System.out.println(s);
```



```
import java.util.Iterator;
public class Stack<Item>
        implements Iterable<Item>
{
…
}

Stack<String> stack = new Stack<String>();

Iterator<String> i = stack.iterator();

while (i.hasNext())
{
   String s = i.next();
   System.out.println(s);
}
```

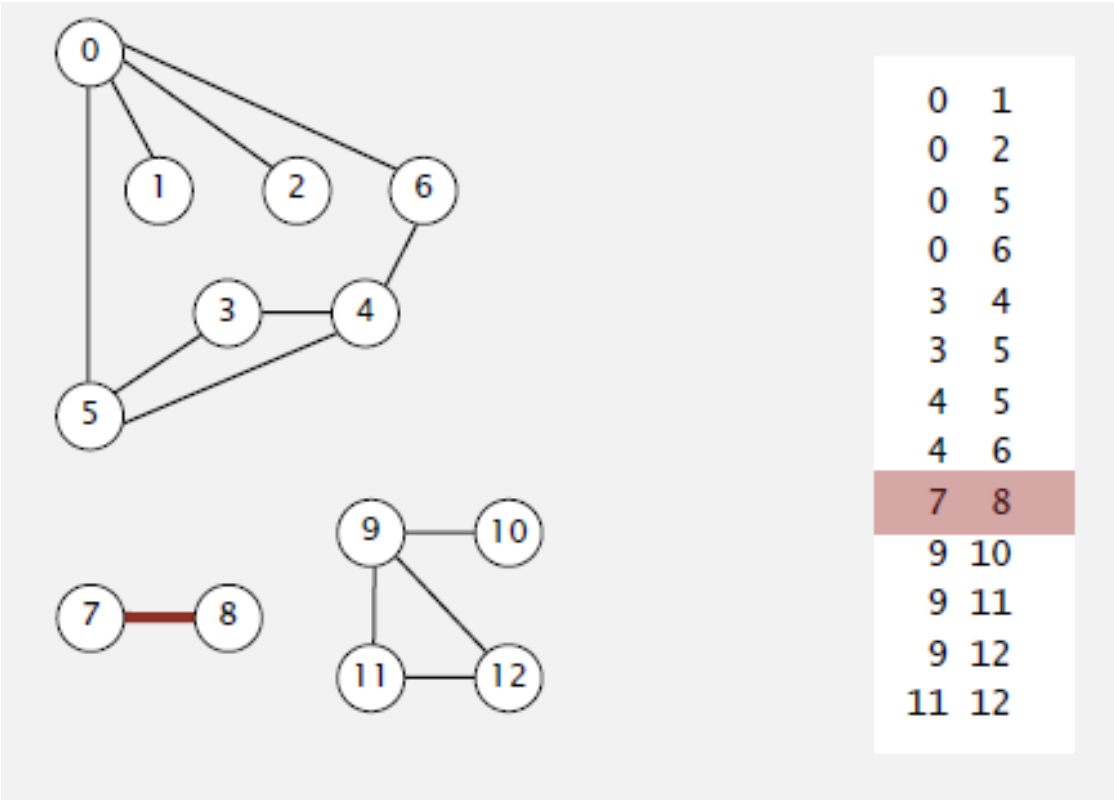
Reprezentacje grafu

- W postaci tablicy krawędzi
- W postacie macierzy sąsiedztwa (*ang. adjacency matrix*)
- W postaci tablicy list sąsiedztwa

Reprezentacje grafu

W postaci tablicy krawędzi

Oznacza to, że pamiętamy wszystkie pary: $(i, j) \in E$



[SW-2018]

Reprezentacje grafu

W postaci macierzy sąsiedztwa

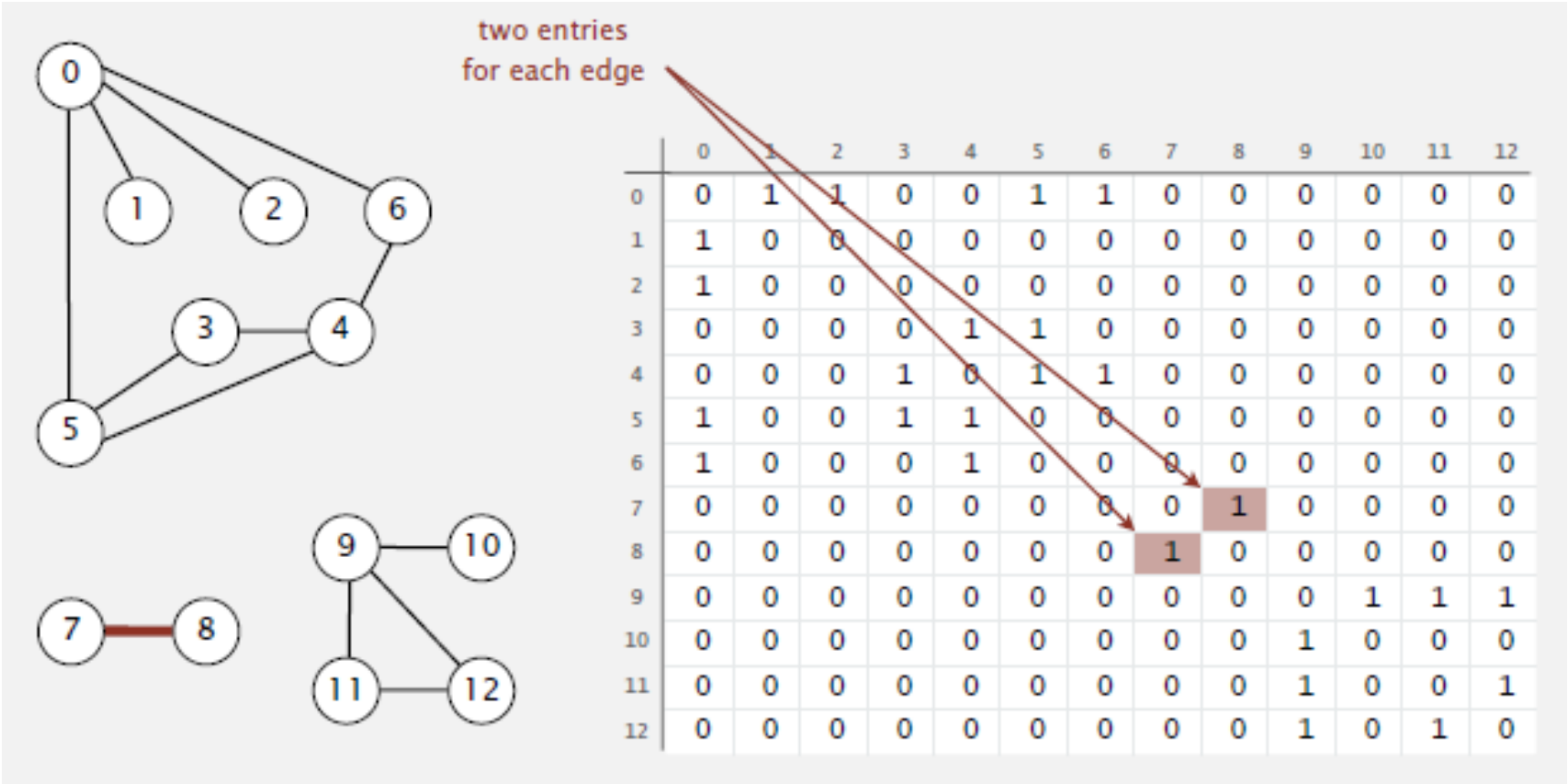
Tworzymy dwuwymiarową tablicę *n* x *n*, gdzie *n* – liczba wierzchołków. Wartościami tablicy są:

- 1 (istnieje krawędź łącząca wierzchołki odpowiadające numerowi wiersza i kolumny, odpowiednio)
- 0 (krawędź taka nie istnieje)

Ew. zamiast 1 i 0 możemy pamiętać wartość typu boolean.

Reprezentacje grafu

W postaci macierzy sąsiedztwa



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

[SW-2018]

Reprezentacje grafu

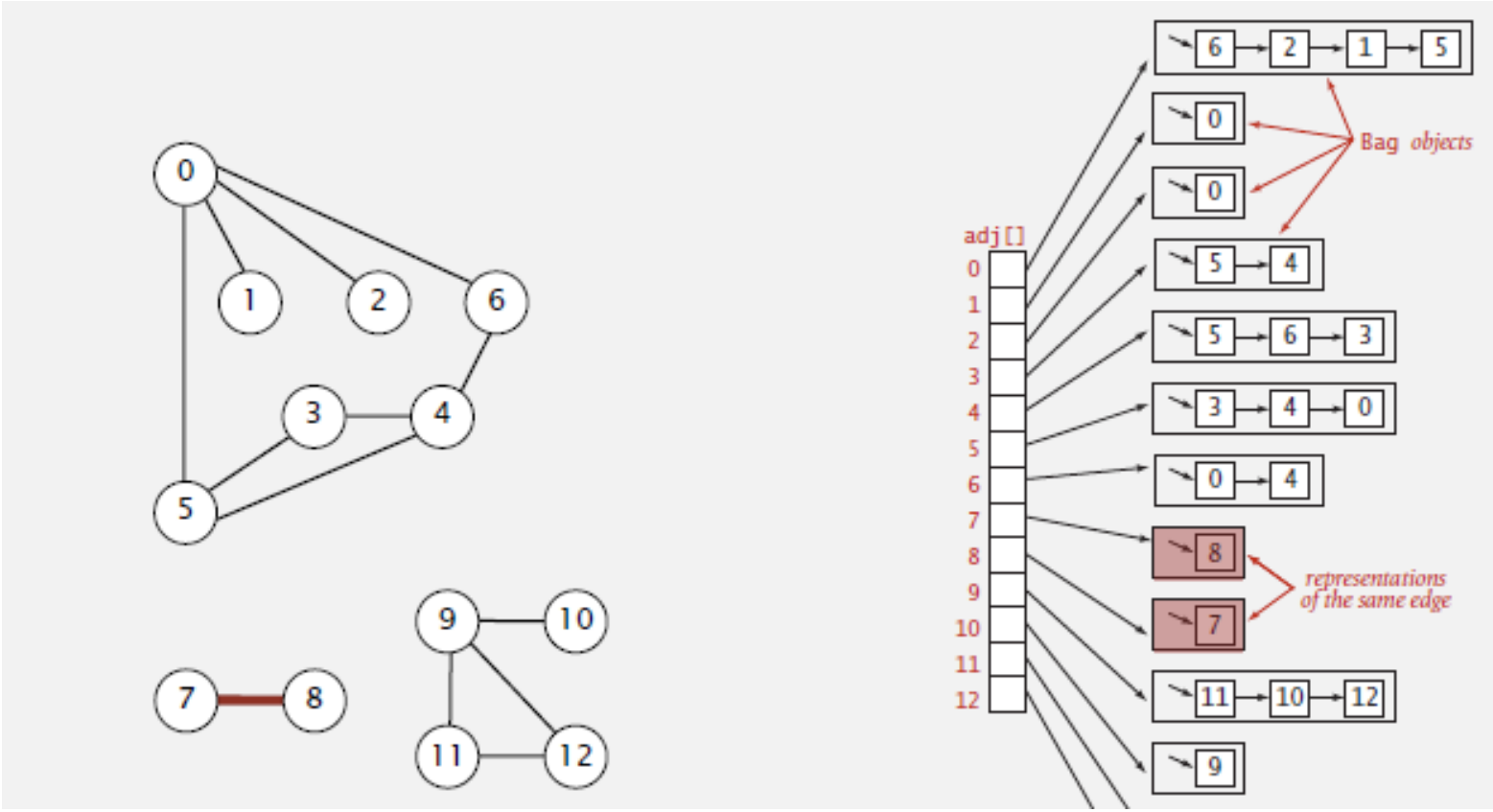
W postaci tablicy list sąsiedztwa

Pamiętamy tablicę jednowymiarową o wymiarze równym liczbie wierzchołków.

Dla każdego wierzchołka $v \in V$ pamiętamy listę wierzchołków, z którymi połączony jest wierzchołek v.

Reprezentacje grafu

W postaci tablicy list sąsiedztwa



[SW-2018]

Reprezentacje grafu

W postaci tablicy list sąsiedztwa

```
public class Graph
{
   private int V;
   private Bag<Integer>[] adj;

   public Graph(int V) {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w) {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v) {
      return adj[v];
   }
}
```

Przeszukiwanie grafu

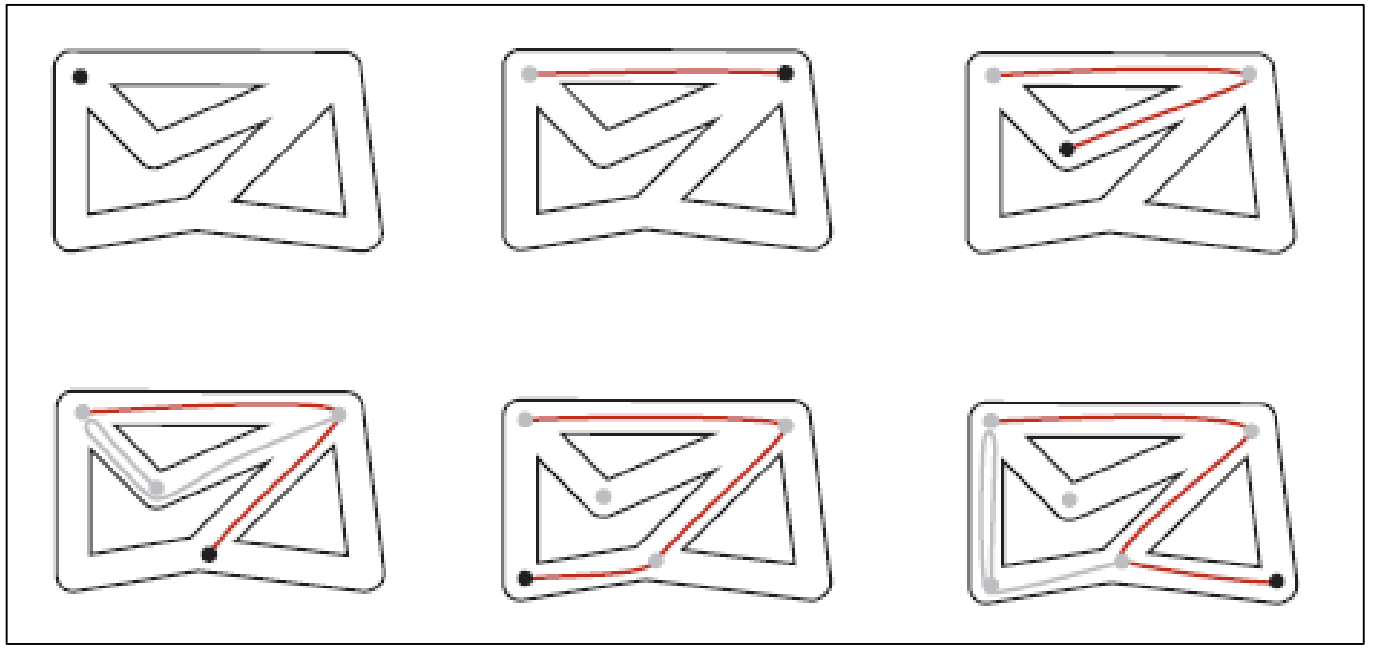
Jak przeszukać graf?

- **Przeszukiwanie w głąb** (ang. *depth-first search* - DFS)
- **Przeszukiwanie wszerz** (ang. *breadth-first search* - BFS)

Przeszukiwanie grafu

Przeszukiwanie w głąb

Idea: zaczerpnięta ze strategii wędrowania po labiryncie, gdzie wierzchołki odpowiadają rozwidleniom dróg, a krawędzie - przejściom między rozwidleniami dróg.



[SW-2018]

Przeszukiwanie grafu

Przeszukiwanie w głąb

Szkic algorytmu:

Startując z pewnego wierzchołka *s*, aby odwiedzić dowolny wierzchołek v:

- Zaznacz wierzchołek v jako odwiedzony
- Rekurencyjnie odwiedź wszystkie niezaznaczone wierzchołki połączone z *v*

Przeszukiwanie grafu

Przeszukiwanie w głąb

- `marked[v] == true` – gdy v połączone jest z s
- `edgeTo[v]` – poprzedni wierzchołek na ścieżce z *s* do *v*

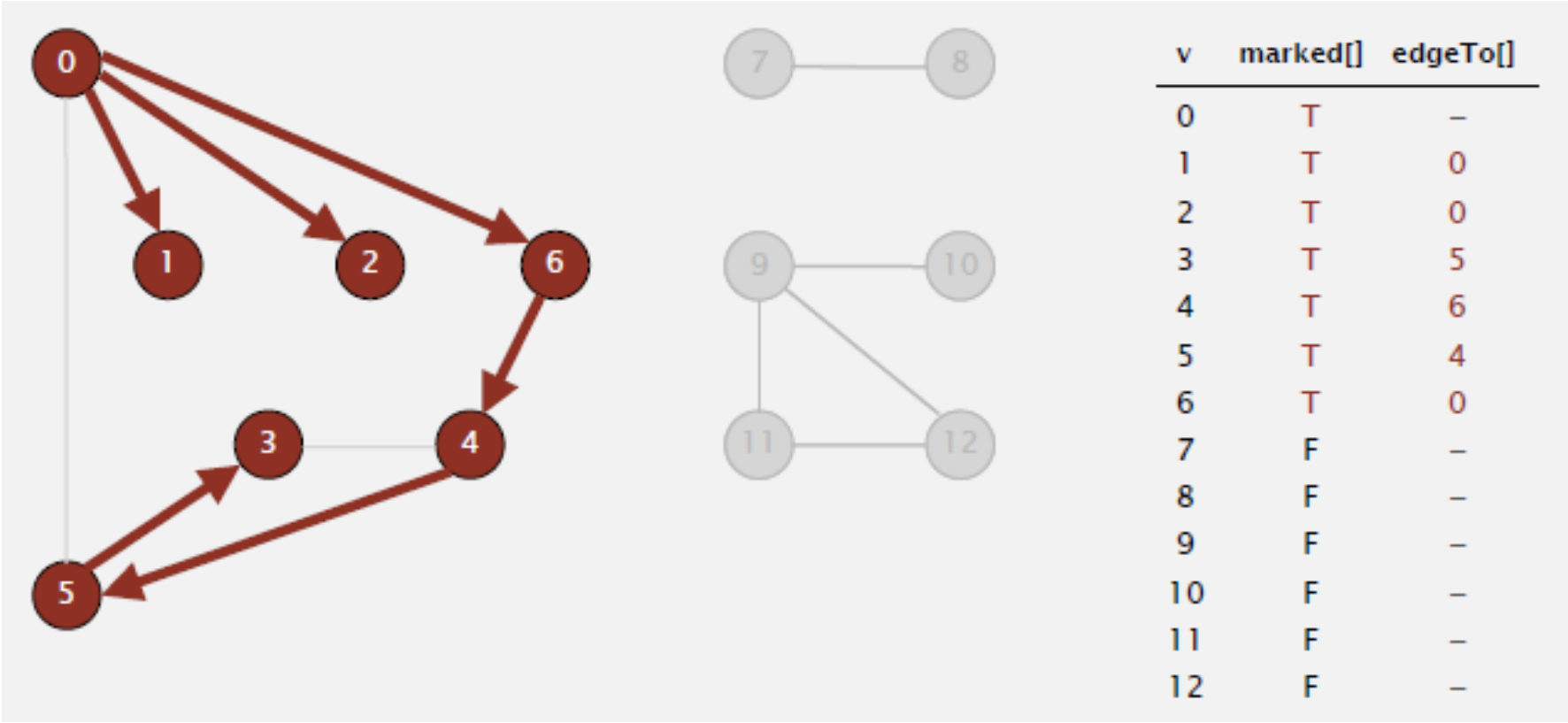
```
public class DepthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int s;

   public DepthFirstPaths(Graph G, int s) {
      // znajdz wierzchołki połączone z s
      dfs(G, s);
   }

   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
         {
            dfs(G, w);
            edgeTo[w] = v;
         }
   }
}
```

Przeszukiwanie grafu

Przeszukiwanie w głąb



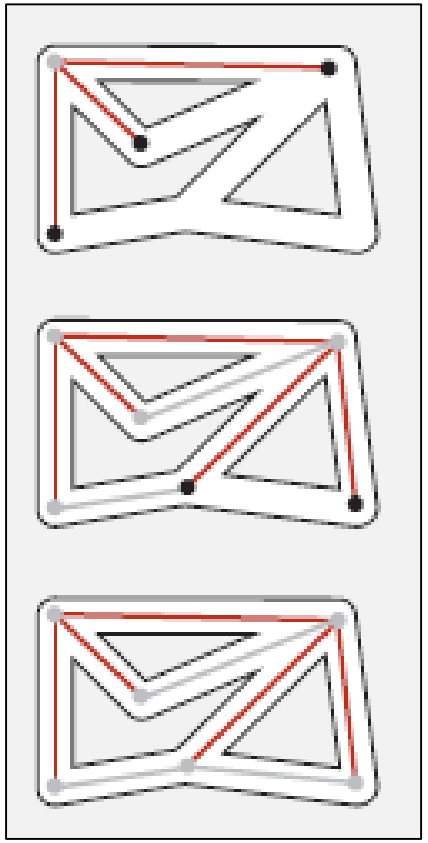
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

[SW-2018]

Przeszukiwanie grafu

Przeszukiwanie wszerz

Idea: wędrowanie po labiryncie, ale nie przez jednego człowieka (jak w DFS), ale przez grupę osób wyruszających w różnych kierunkach.



[SW-2018]

Przeszukiwanie grafu

Przeszukiwanie wszerz

Pytanie:

Czy dla danego grafu i źródłowego wierzchołka *s* istnieje ścieżka z *s* do danego wierzchołka? Jeśli tak, znajdź najkrótszą ścieżkę.

Aby znaleźć najkrótszą ścieżkę z *s* do *v*, należy zacząć w *s* i sprawdzić, czy *v* znajduje się wśród wierzchołków, do których można dotrzeć poprzez jedną krawędź, następnie poszukać *v* wśród wierzchołków dostępnych z *s* przez dwie krawędzie, itd.

Przeszukiwanie grafu

Przeszukiwanie wszerz

Szkic algorytmu:

Startując z pewnego wierzchołka *s*, aby odwiedzić dowolny wierzchołek *v*:

- Dodaj *s* do kolejki (FIFO) i zaznacz *s* jako odwiedzony.
- Powtarzaj aż kolejka stanie się pusta:
  - Usuń najwcześniej dodany wierzchołek *v*
  - Dodaj każdy z nieodwiedzonych sąsiadów *v* do kolejki i zaznacz je jako odwiedzone

Przeszukiwanie grafu

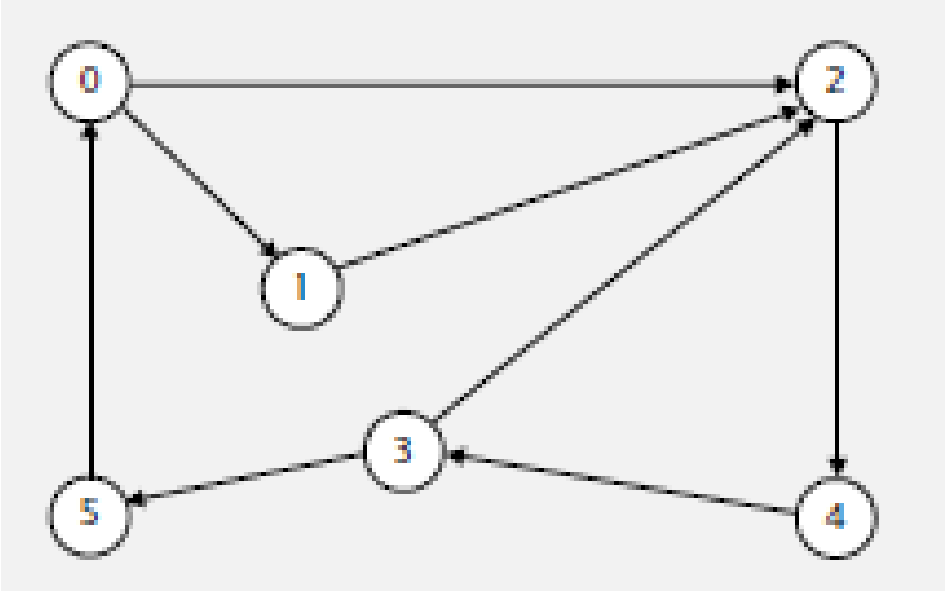
Przeszukiwanie wszerz

Kolejność przeglądania wierzchołków?

```
public class BreadthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int[] distTo;
   …
   private void bfs(Graph G, int s) {
      Queue<Integer> q = new Queue<Integer>();
      q.enqueue(s);
      marked[s] = true;
      distTo[s] = 0;
      while (!q.isEmpty()) {
         int v = q.dequeue();
         for (int w : G.adj(v)) {
            if (!marked[w]) {
               q.enqueue(w);
               marked[w] = true;
               edgeTo[w] = v;
               distTo[w] = distTo[v] + 1;
            }
         }
      }
   }
}
```

Graf skierowany

Graf, w którym krawędzie mają charakter skierowany.



Graf skierowany

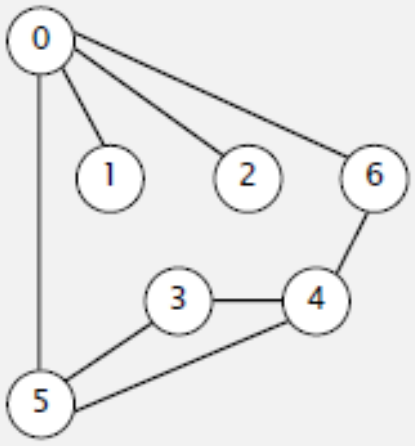
Opis	Metoda
Konstruktor, konstruuje graf o liczbie wierzchołków V	`Digraph(int V)`
Dodaje do grafu krawędź skierowaną łączącą prowadzącą od v do w	`void addEdge(int v, int w)`
Zwraca wierzchołki wychodzące z v	`Iterable<Integer> adj(int v)`
Zwraca liczbę wierzchołków	`int V()`
Zwraca liczbę krawędzi	`int E()`

Reprezentacje grafu skierowanego

- W postaci tablicy krawędzi
- W postacie macierzy sąsiedztwa (*ang. adjacency matrix*)
- W postaci tablicy list sąsiedztwa

Reprezentacje grafu skierowanego

W postaci tablicy krawędzi



0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Tak było w przypadku grafu nieskierowanego

A jak będzie w przypadku grafu skierowanego ???

???

[SW-2018]

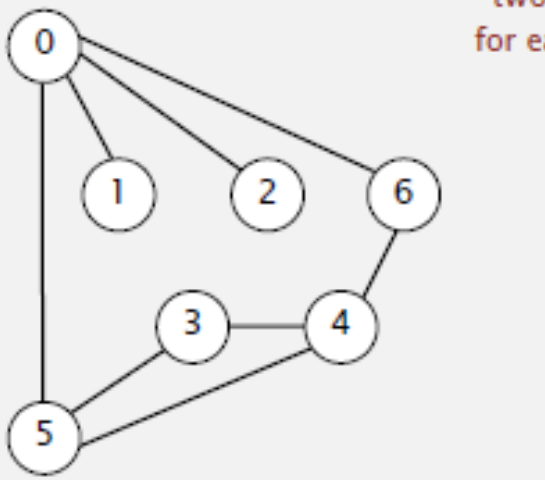
Reprezentacje grafu skierowanego

W postaci macierzy sąsiedztwa

Tak było w przypadku grafu nieskierowanego

A jak będzie w przypadku grafu skierowanego ???

???



two entries for each edge

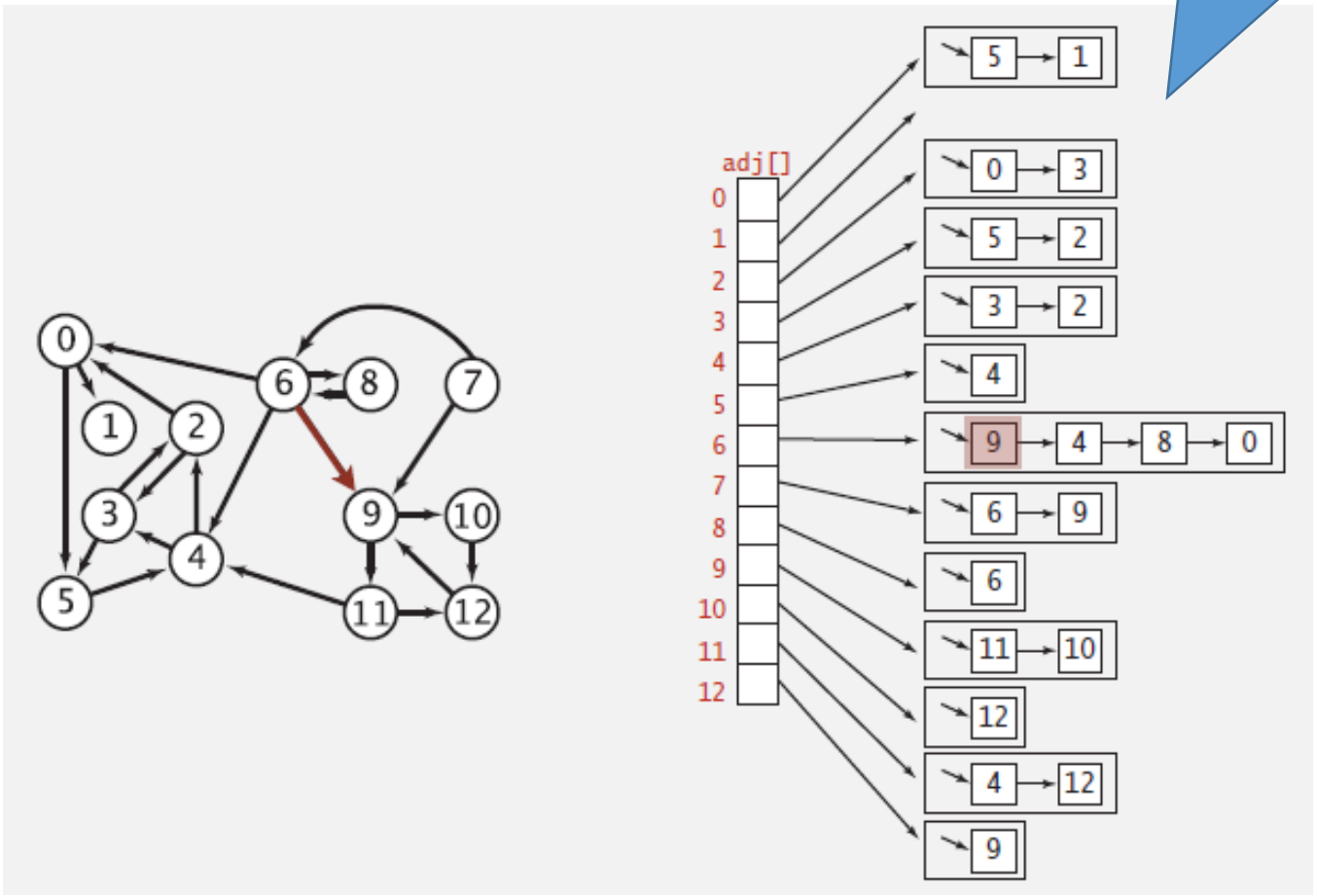
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

[SW-2018]

Reprezentacje grafu skierowanego

W postaci tablicy list sąsiedztwa

Przypadek grafu skierowanego



[SW-2018]

Reprezentacje grafu

W postaci tablicy list sąsiedztwa

```
public class Digraph
{
   private final int V;
   private Bag<Integer>[] adj;

   public Graph(int V) {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w) {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v) {
      return adj[v];
   }
}
```

Przeszukiwanie grafu skierowanego

Przeszukiwanie w głąb

Dokładnie tak samo jak dla grafu nieskierowanego

(dlaczego?)

Przeszukiwanie grafu skierowanego

Przeszukiwanie wszerz

Dokładnie tak samo jak dla grafu nieskierowanego

(dlaczego?)