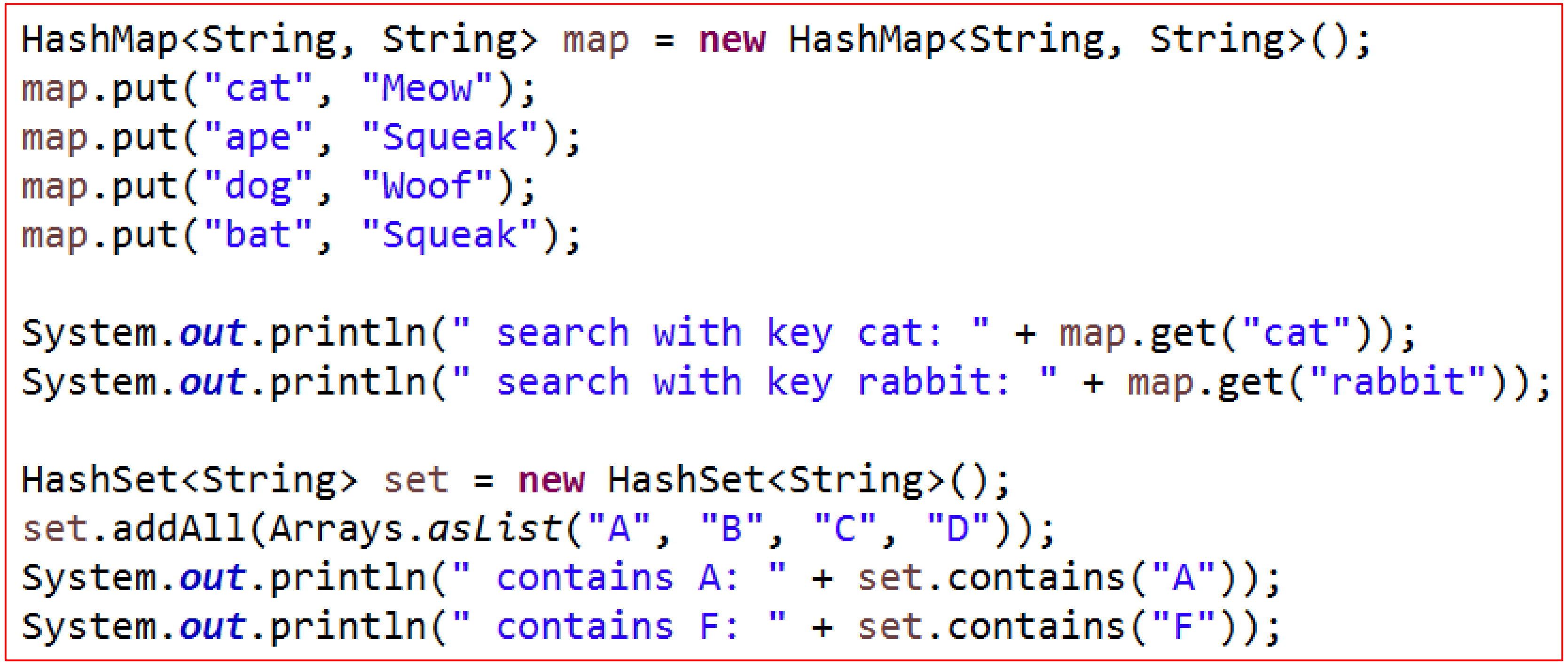
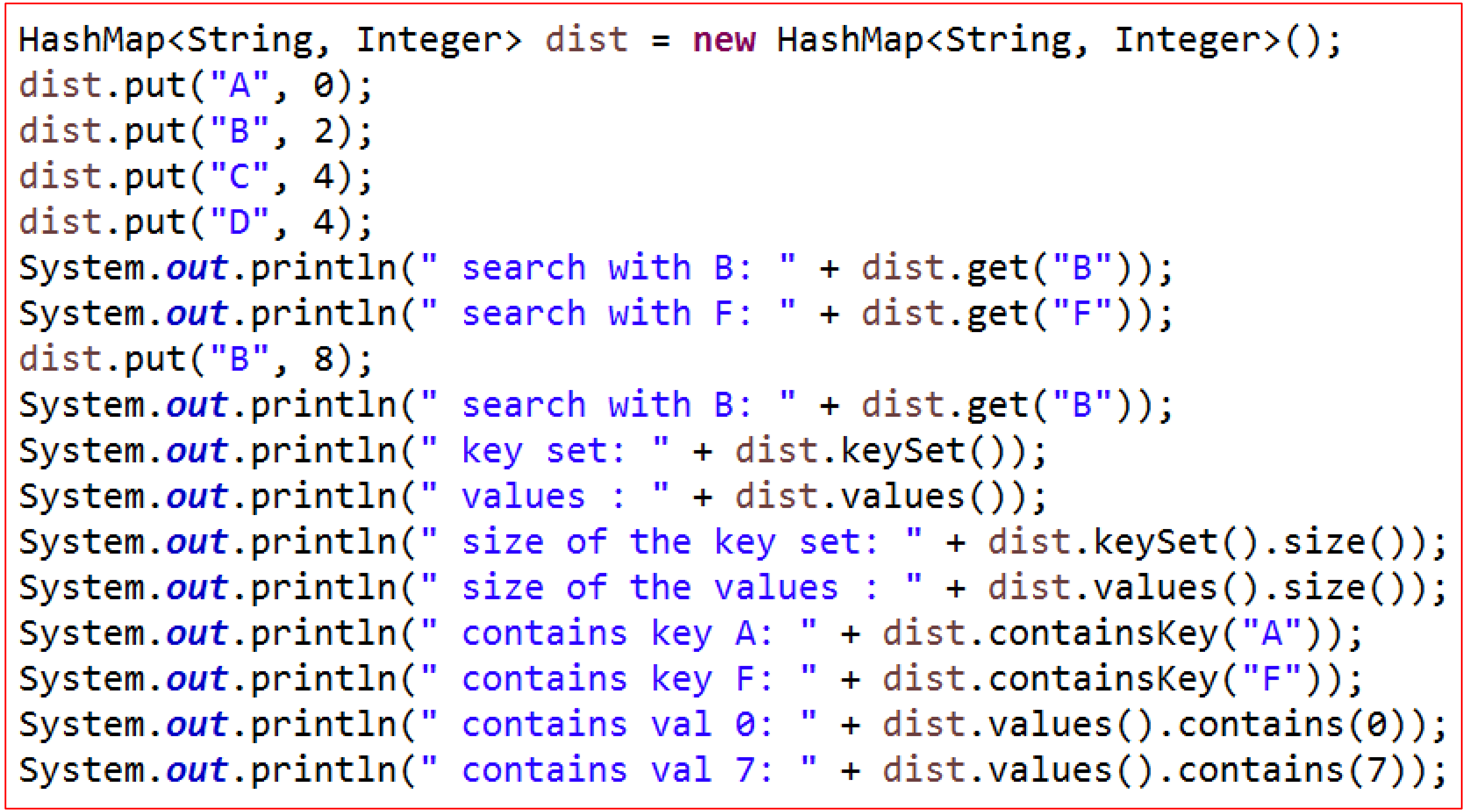
HashMap/Set

Examples

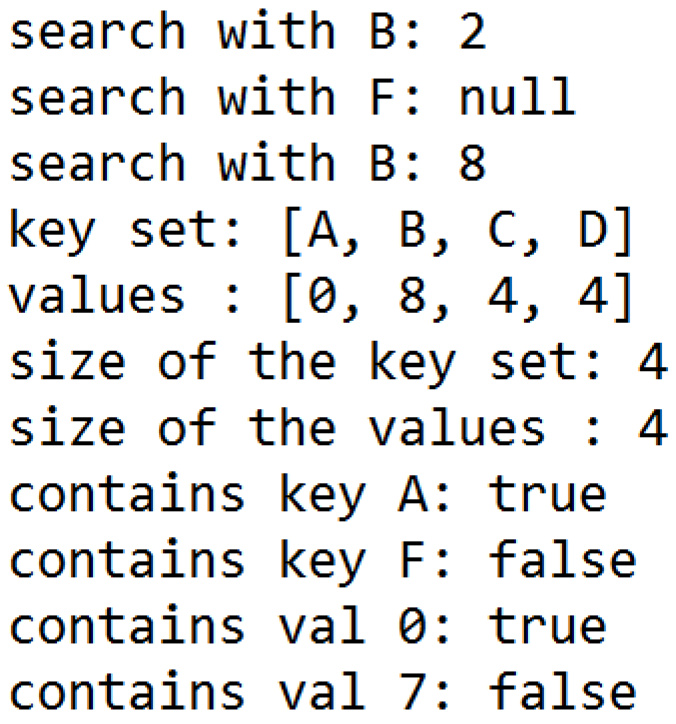


```java
HashMap<String, String> map = new HashMap<String, String>();
map.put("cat", "Meow");
map.put("ape", "Squeak");
map.put("dog", "Woof");
map.put("bat", "Squeak");

System.out.println(" search with key cat: " + map.get("cat"));
System.out.println(" search with key rabbit: " + map.get("rabbit"));

HashSet<String> set = new HashSet<String>();
set.addAll(Arrays.asList("A", "B", "C", "D"));
System.out.println(" contains A: " + set.contains("A"));
System.out.println(" contains F: " + set.contains("F"));
```

```
search with key cat: Meow
search with key rabbit: null
contains A: true
contains F: false
```



```java
HashMap<String, Integer> dist = new HashMap<String, Integer>();
dist.put("A", 0);
dist.put("B", 2);
dist.put("C", 4);
dist.put("D", 4);
System.out.println(" search with B: " + dist.get("B"));
System.out.println(" search with F: " + dist.get("F"));
dist.put("B", 8);
System.out.println(" search with B: " + dist.get("B"));
System.out.println(" key set: " + dist.keySet());
System.out.println(" values : " + dist.values());
System.out.println(" size of the key set: " + dist.keySet().size());
System.out.println(" size of the values : " + dist.values().size());
System.out.println(" contains key A: " + dist.containsKey("A"));
System.out.println(" contains key F: " + dist.containsKey("F"));
System.out.println(" contains val 0: " + dist.values().contains(0));
System.out.println(" contains val 7: " + dist.values().contains(7));
```

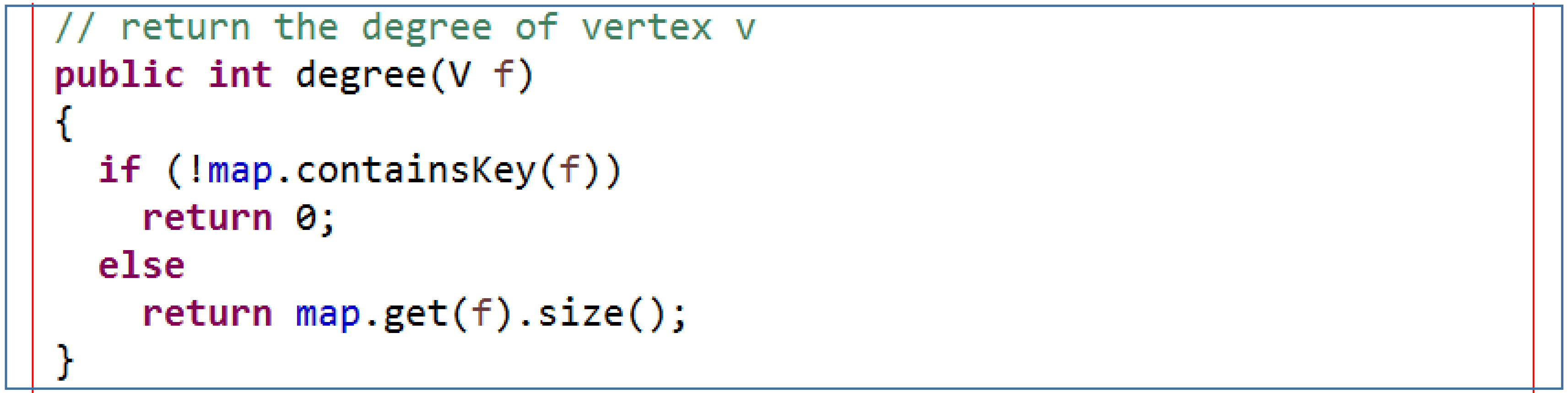


```
search with B: 2
search with F: null
search with B: 8
key set: [A, B, C, D]
values : [0, 8, 4, 4]
size of the key set: 4
size of the values : 4
contains key A: true
contains key F: false
contains val 0: true
contains val 7: false
```

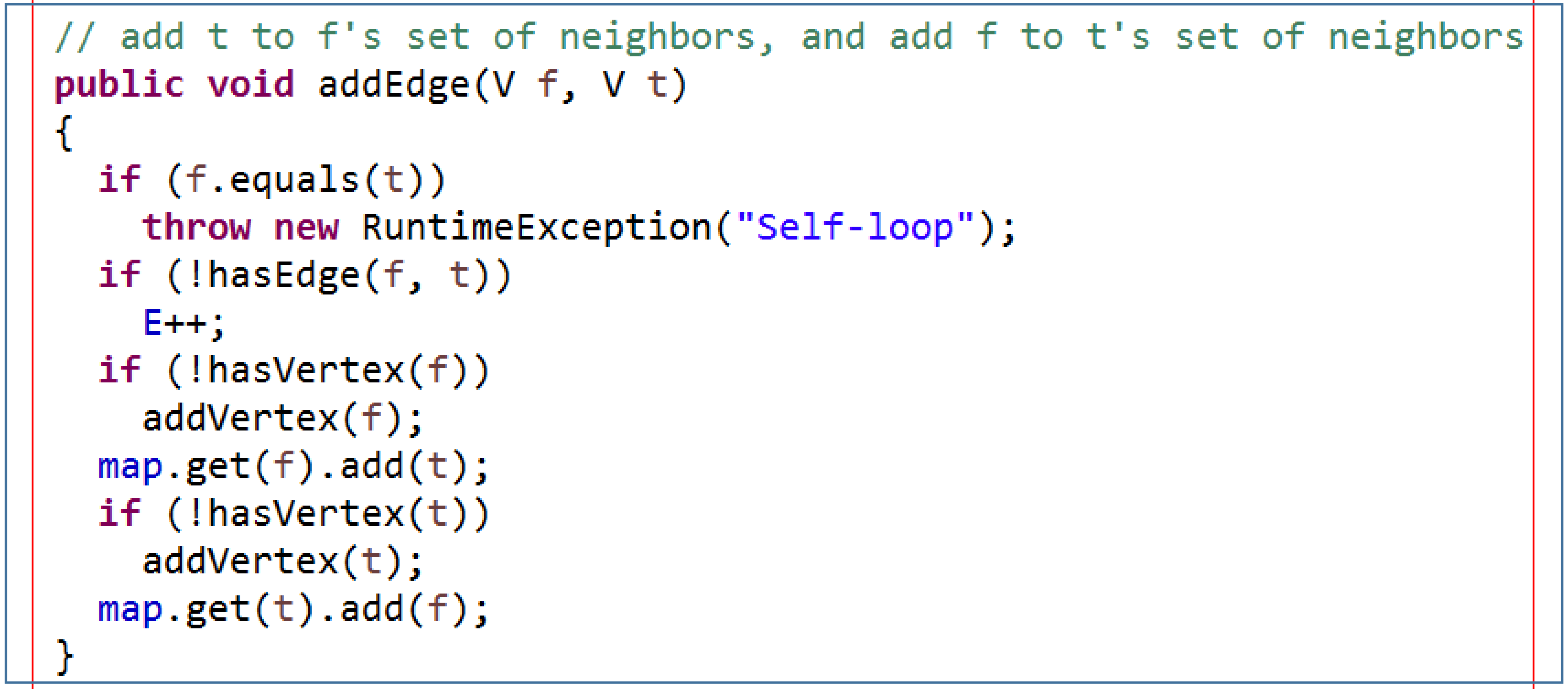
Graph implementation using HashMap/Set

Undirected, no self loops, no parallel edges

```java
import java.util.*;
public class Graph<V> // V is the vertex type
{
 // map : key = vertex, value = set of neighboring vertices
 private HashMap<V, HashSet<V>> map;

 // number of edges
 private int E;

 // create an empty graph
 public Graph()
 {
   map = new HashMap<V, HashSet<V>>();
 }

 public int numV()
 {
   return map.keySet().size();
 }

 public int numE()
 {
   return E;
 }
```



```java
// return the degree of vertex v
public int degree(V f)
{
  if (!map.containsKey(f))
    return 0;
  else
    return map.get(f).size();
}
```

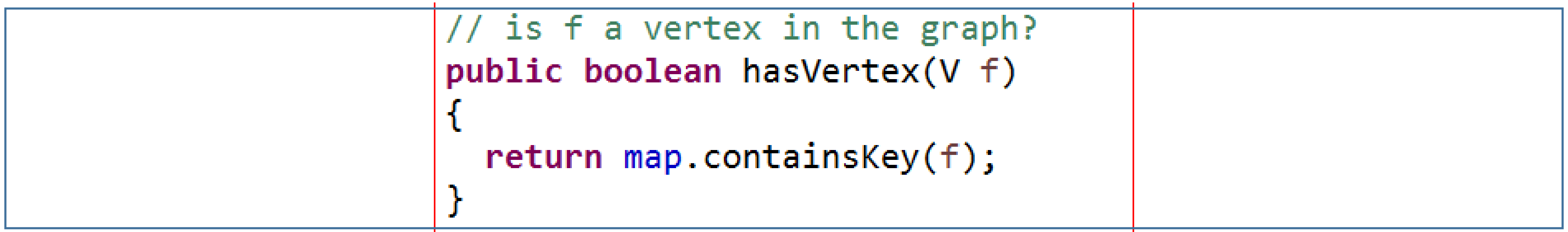


```java
// add t to f's set of neighbors, and add f to t's set of neighbors
public void addEdge(V f, V t)
{
  if (f.equals(t))
    throw new RuntimeException("Self-loop");
  if (!hasEdge(f, t))
    E++;
  if (!hasVertex(f))
    addVertex(f);
  map.get(f).add(t);
  if (!hasVertex(t))
    addVertex(t);
  map.get(t).add(f);
}
```

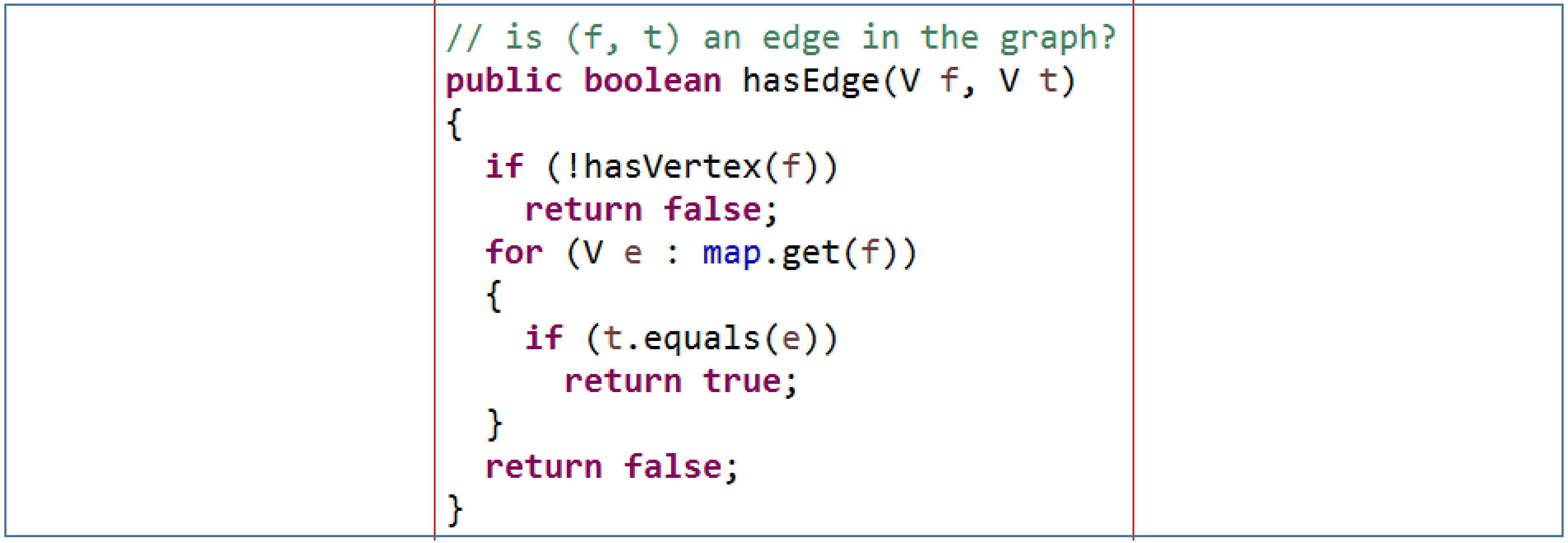
```
// add a new vertex f with no neighbors (if vertex does not yet exist)
public void addVertex(V f)
{
  if (!hasVertex(f))
    map.put(f, new HashSet<V>());
}
```

```
// return iterator over all vertices in graph
public Iterable<V> vertices()
{
  return map.keySet();
}
```

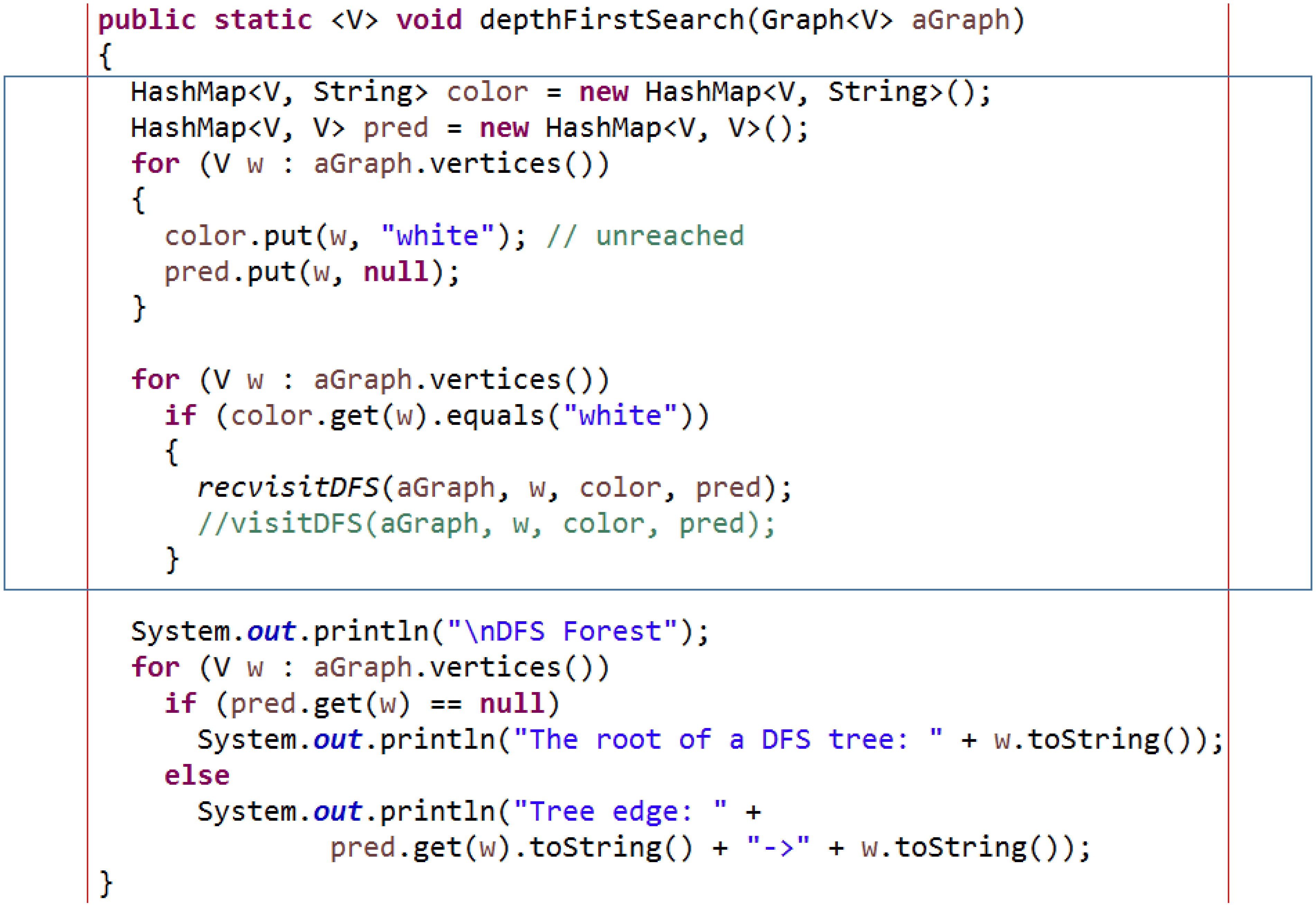
```
// return an iterator over the neighbors of vertex f
public Iterable<V> adjacentTo(V f)
{
  // return empty set if vertex isn't in graph
  if (!hasVertex(f))
    return new HashSet<V>();
  else
    return map.get(f);
}
```



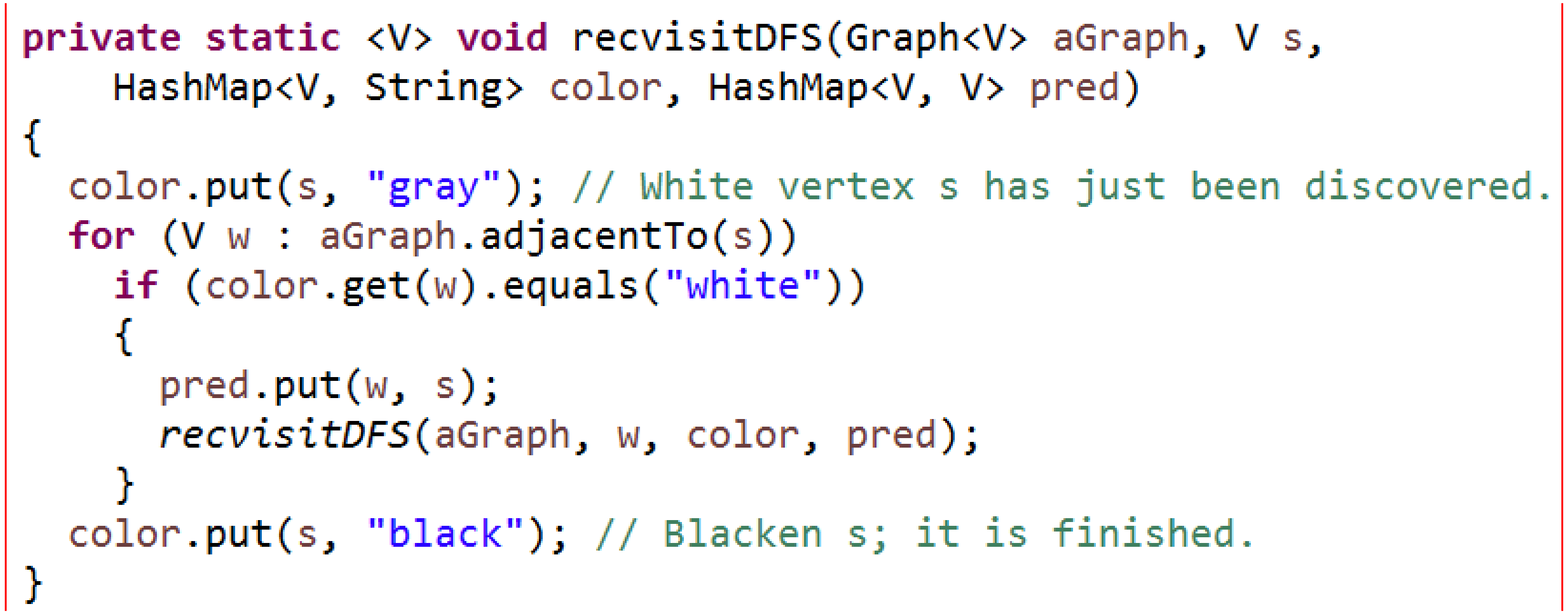
```java
// is f a vertex in the graph?
public boolean hasVertex(V f)
{
  return map.containsKey(f);
}
```



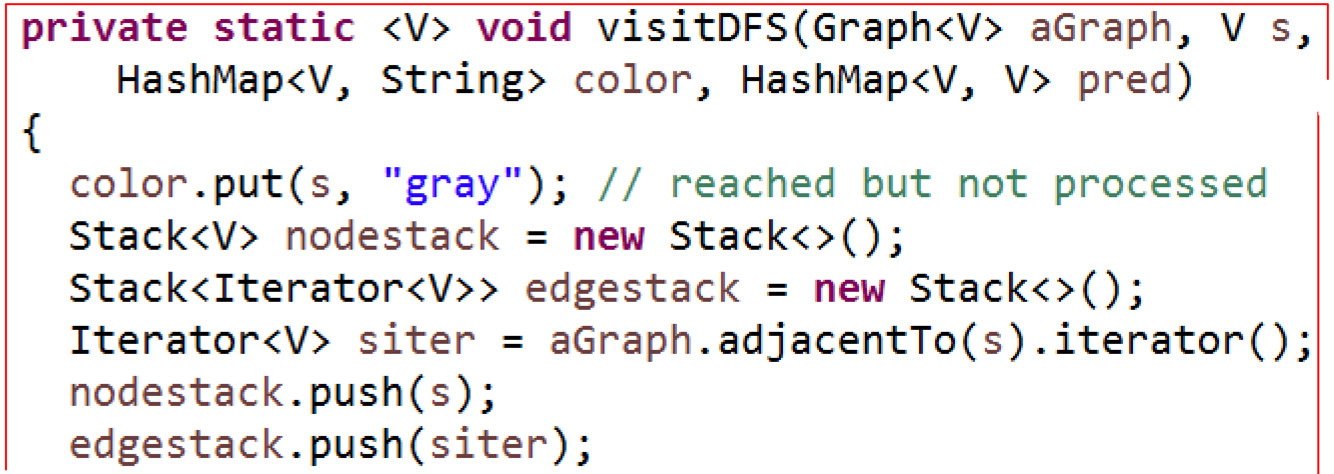
```java
// is (f, t) an edge in the graph?
public boolean hasEdge(V f, V t)
{
  if (!hasVertex(f))
    return false;
  for (V e : map.get(f))
  {
    if (t.equals(e))
      return true;
  }
  return false;
}
```



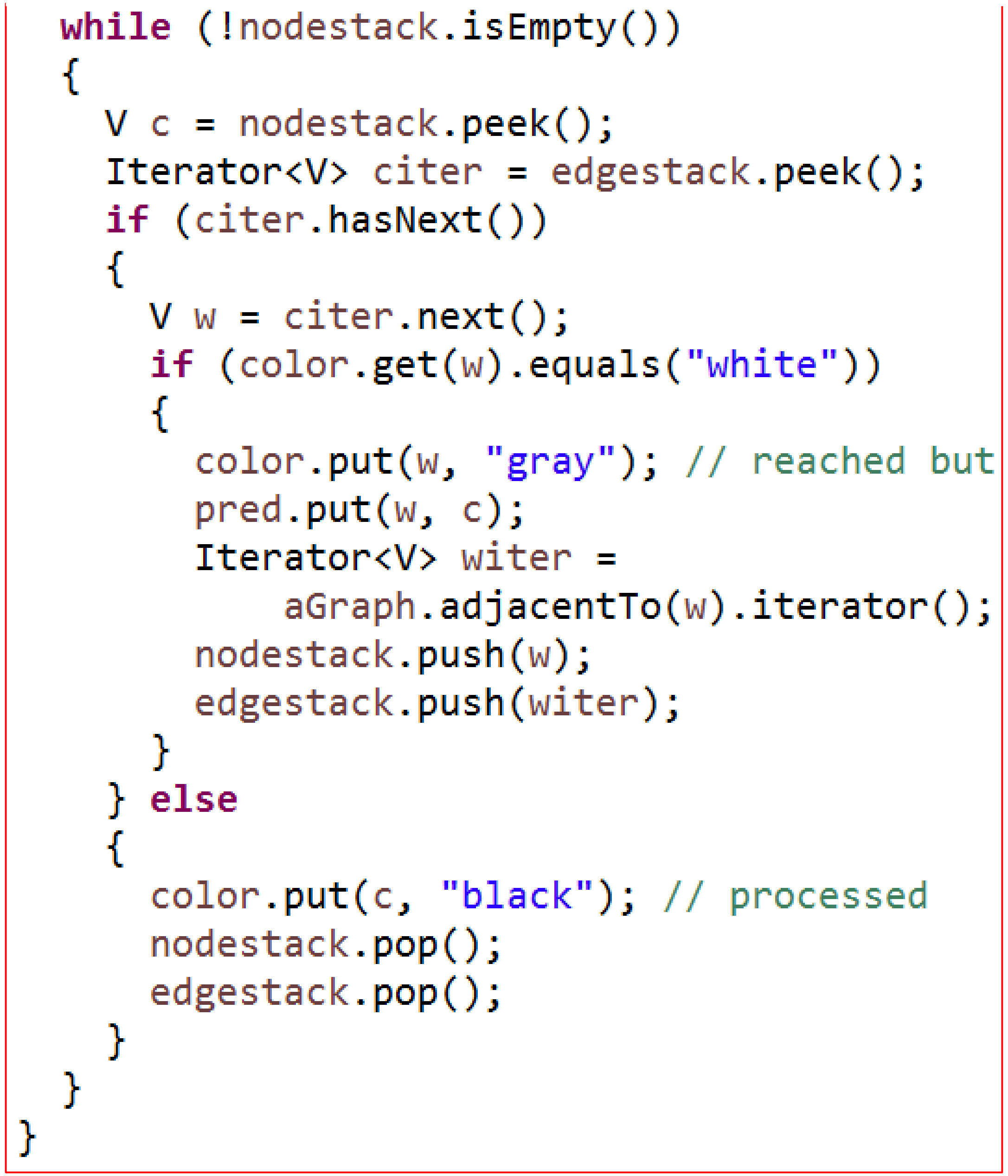
```java
public static <V> void depthFirstSearch(Graph<V> aGraph)
{
  HashMap<V, String> color = new HashMap<V, String>();
  HashMap<V, V> pred = new HashMap<V, V>();
  for (V w : aGraph.vertices())
  {
    color.put(w, "white"); // unreached
    pred.put(w, null);
  }

  for (V w : aGraph.vertices())
    if (color.get(w).equals("white"))
    {
      recvisitDFS(aGraph, w, color, pred);
      //visitDFS(aGraph, w, color, pred);
    }

  System.out.println("\nDFS Forest");
  for (V w : aGraph.vertices())
    if (pred.get(w) == null)
      System.out.println("The root of a DFS tree: " + w.toString());
    else
      System.out.println("Tree edge: " +
              pred.get(w).toString() + "->" + w.toString());
}
```



```java
private static <V> void recvisitDFS(Graph<V> aGraph, V s,
    HashMap<V, String> color, HashMap<V, V> pred)
{
  color.put(s, "gray"); // White vertex s has just been discovered.
  for (V w : aGraph.adjacentTo(s))
    if (color.get(w).equals("white"))
    {
      pred.put(w, s);
      recvisitDFS(aGraph, w, color, pred);
    }
  color.put(s, "black"); // Blacken s; it is finished.
}
```



```
private static <V> void visitDFS(Graph<V> aGraph, V s,
    HashMap<V, String> color, HashMap<V, V> pred)
{
  color.put(s, "gray"); // reached but not processed
  Stack<V> nodestack = new Stack<>();
  Stack<Iterator<V>> edgestack = new Stack<>();
  Iterator<V> siter = aGraph.adjacentTo(s).iterator();
  nodestack.push(s);
  edgestack.push(siter);
```



```
  while (!nodestack.isEmpty())
  {
    V c = nodestack.peek();
    Iterator<V> citer = edgestack.peek();
    if (citer.hasNext())
    {
      V w = citer.next();
      if (color.get(w).equals("white"))
      {
        color.put(w, "gray"); // reached but
        pred.put(w, c);
        Iterator<V> witer =
            aGraph.adjacentTo(w).iterator();
        nodestack.push(w);
        edgestack.push(witer);
      }
    } else
    {
      color.put(c, "black"); // processed
      nodestack.pop();
      edgestack.pop();
    }
  }
}
```

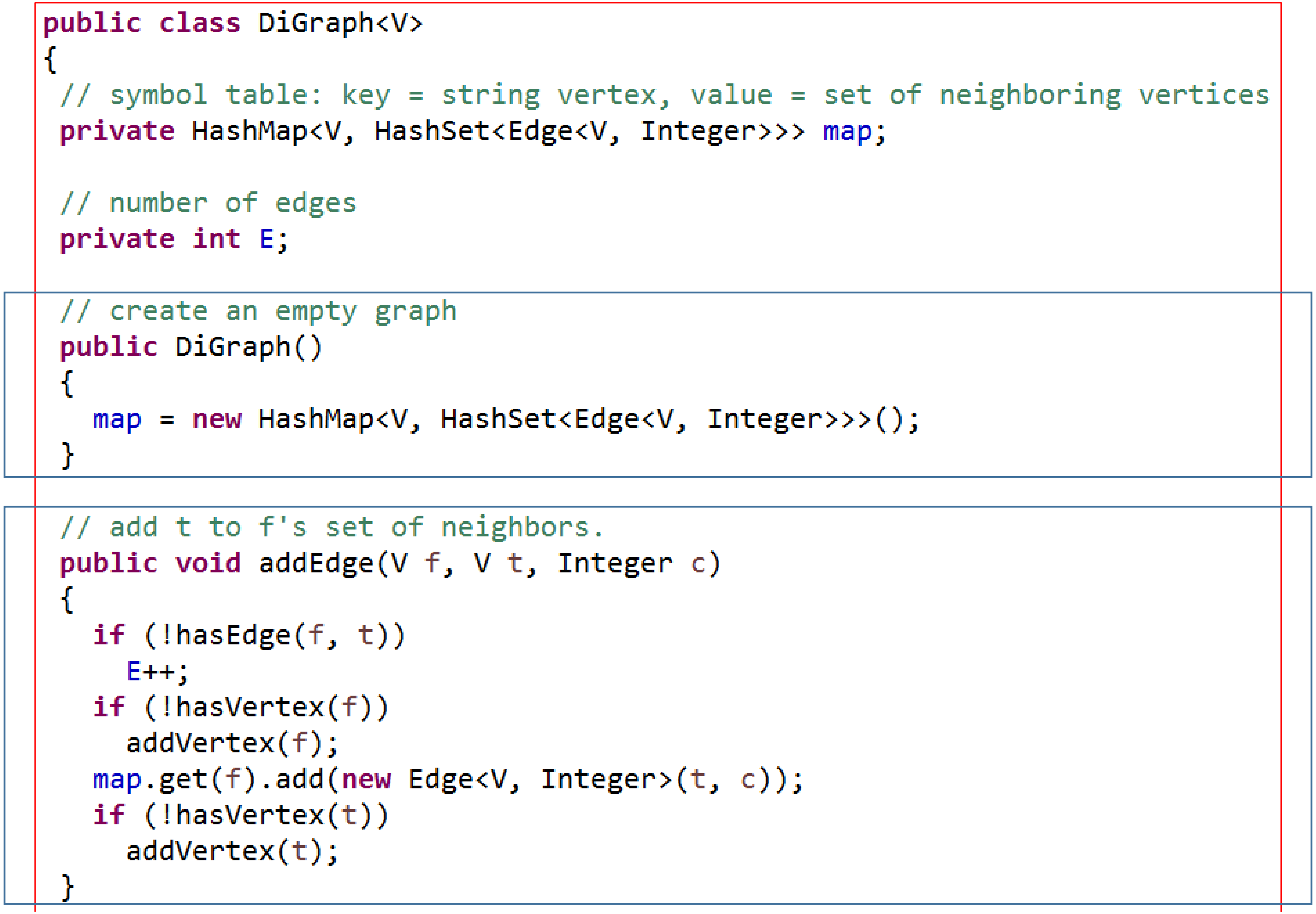
DiGraph implementation using HashMap/Set

No parallel edges in one direction

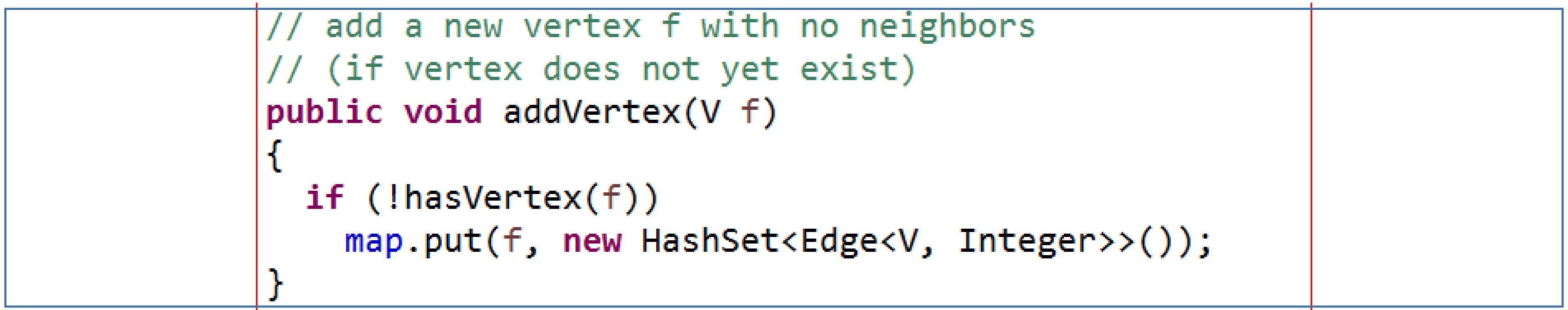
```java
public class Edge<V, C extends Comparable<? super C>>
       implements Comparable<Edge<V, C>>
{
 private V node;
 private C cost;

 Edge(V n, C c)
 {
   node = n;
   cost = c;
 }

 public V getVertex()
 {
   return node;
 }

 public C getCost()
 {
   return cost;
 }

 public int compareTo(Edge<V, C> other)
 {
   return cost.compareTo(other.getCost());
 }
 public String toString()
 {
   return "<" + node.toString() + ", " + cost.toString() + ">";
 }
```

```java
 public int hashCode()
 {
   return node.hashCode();
 }
```

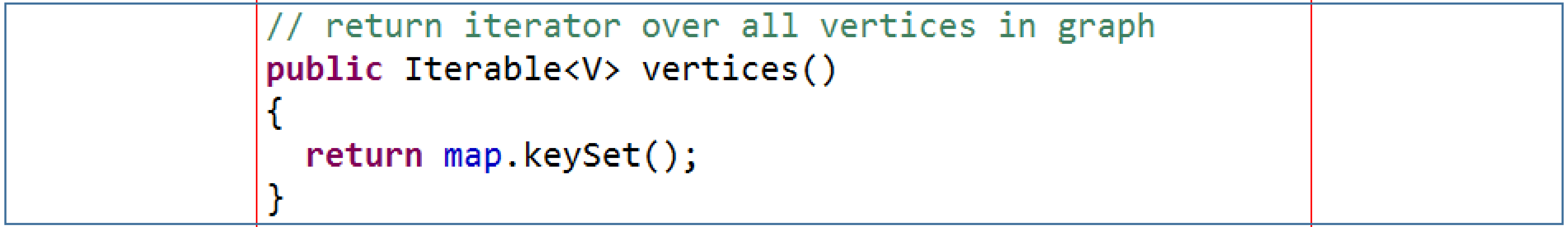
```java
 public boolean equals(Object obj)
 {
   if (this == obj)
     return true;
   if ((obj == null) || (obj.getClass() != this.getClass()))
     return false;
   Edge<?, ?> test = (Edge<?, ?>) obj;
   return (node == test.node || (node != null && node.equals(test.node)));
   // (node == test.node || (node != null && node.equals(test.node))) &&
   // (cost == test.cost || (cost != null && cost.equals(test.cost)));
 }
}
```



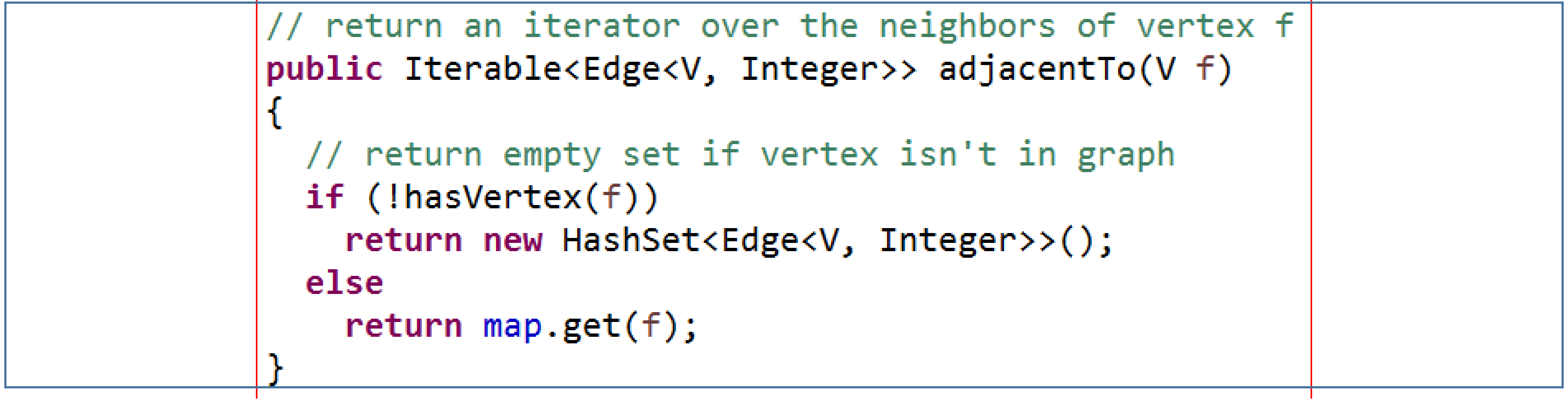
```
public class DiGraph<V>
{
 // symbol table: key = string vertex, value = set of neighboring vertices
 private HashMap<V, HashSet<Edge<V, Integer>>> map;

 // number of edges
 private int E;

 // create an empty graph
 public DiGraph()
 {
   map = new HashMap<V, HashSet<Edge<V, Integer>>>();
 }

 // add t to f's set of neighbors.
 public void addEdge(V f, V t, Integer c)
 {
   if (!hasEdge(f, t))
     E++;
   if (!hasVertex(f))
     addVertex(f);
   map.get(f).add(new Edge<V, Integer>(t, c));
   if (!hasVertex(t))
     addVertex(t);
 }
```



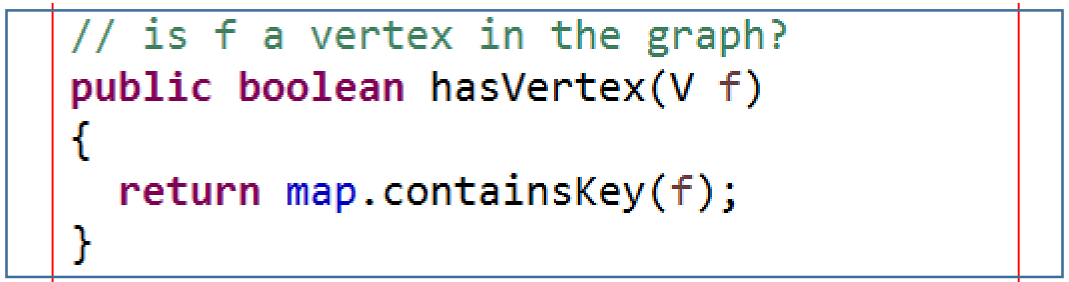
```
// add a new vertex f with no neighbors
// (if vertex does not yet exist)
public void addVertex(V f)
{
  if (!hasVertex(f))
    map.put(f, new HashSet<Edge<V, Integer>>());
}
```



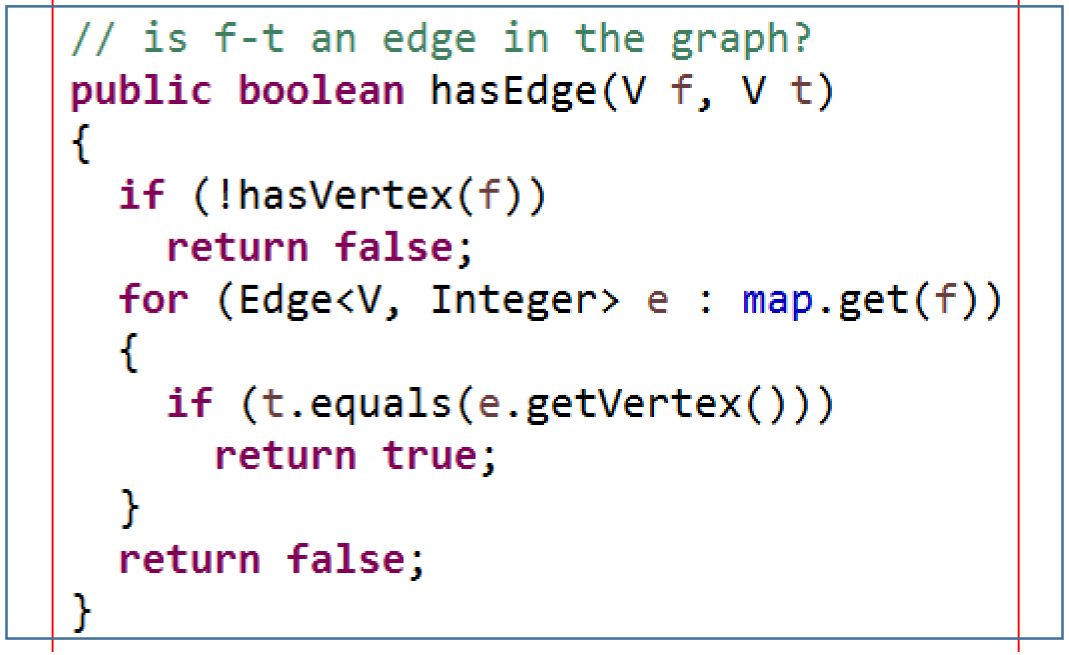
```
// return iterator over all vertices in graph
public Iterable<V> vertices()
{
  return map.keySet();
}
```



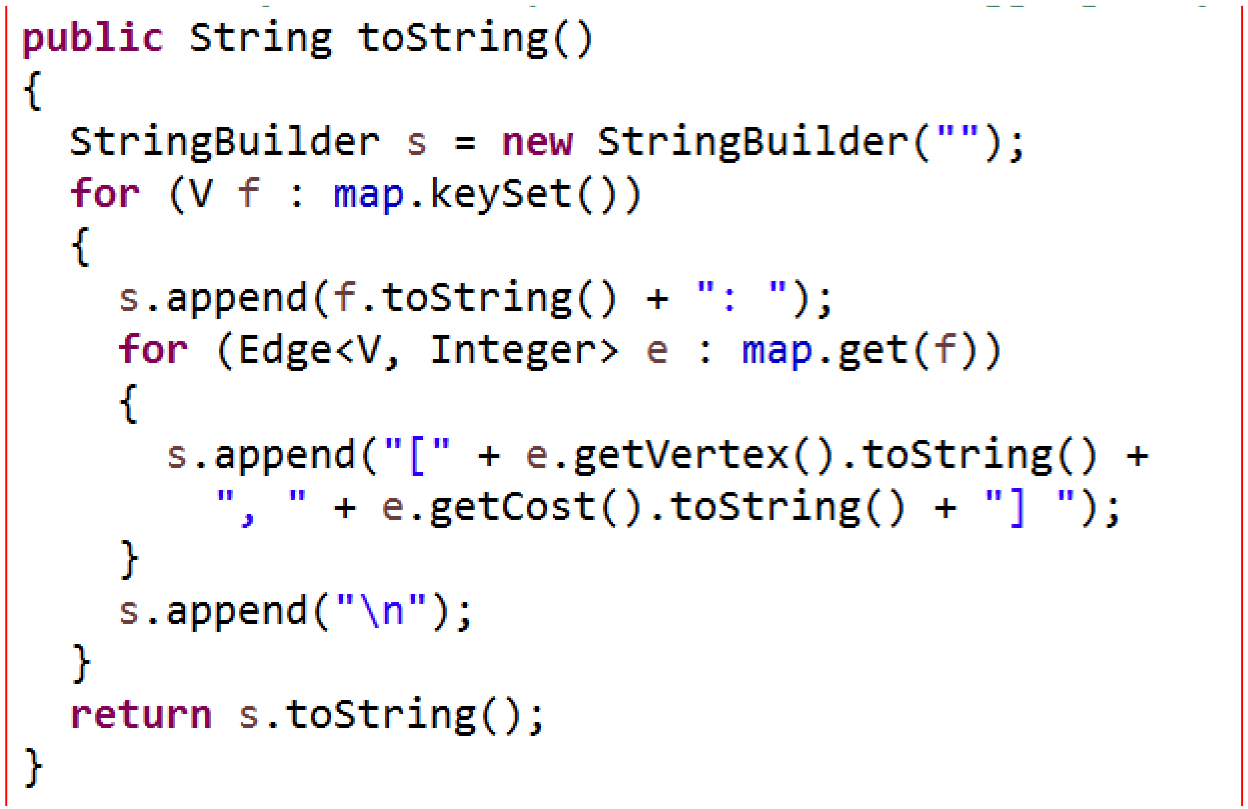
```
// return an iterator over the neighbors of vertex f
public Iterable<Edge<V, Integer>> adjacentTo(V f)
{
  // return empty set if vertex isn't in graph
  if (!hasVertex(f))
    return new HashSet<Edge<V, Integer>>();
  else
    return map.get(f);
}
```



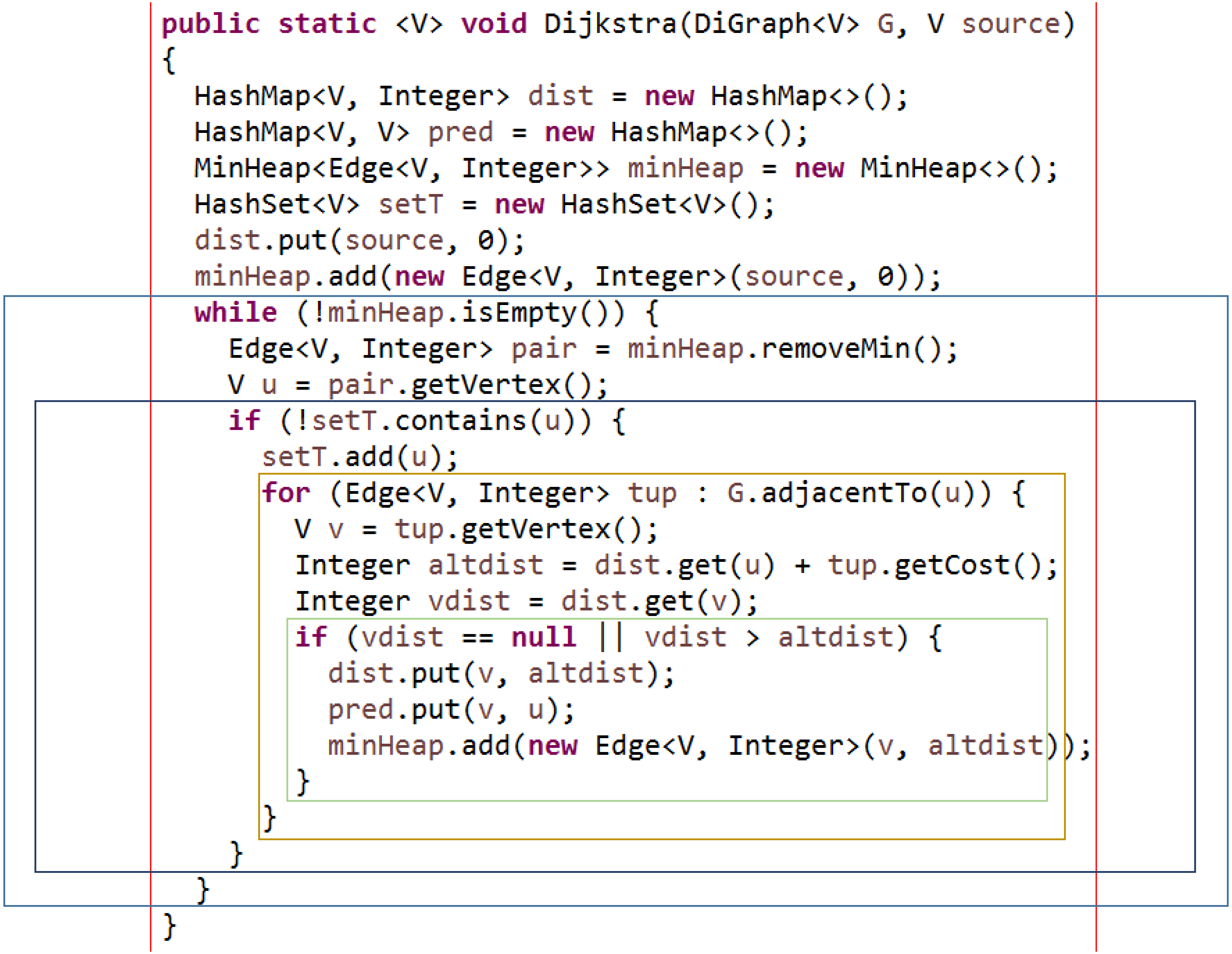
```java
// is f a vertex in the graph?
public boolean hasVertex(V f)
{
  return map.containsKey(f);
}
```



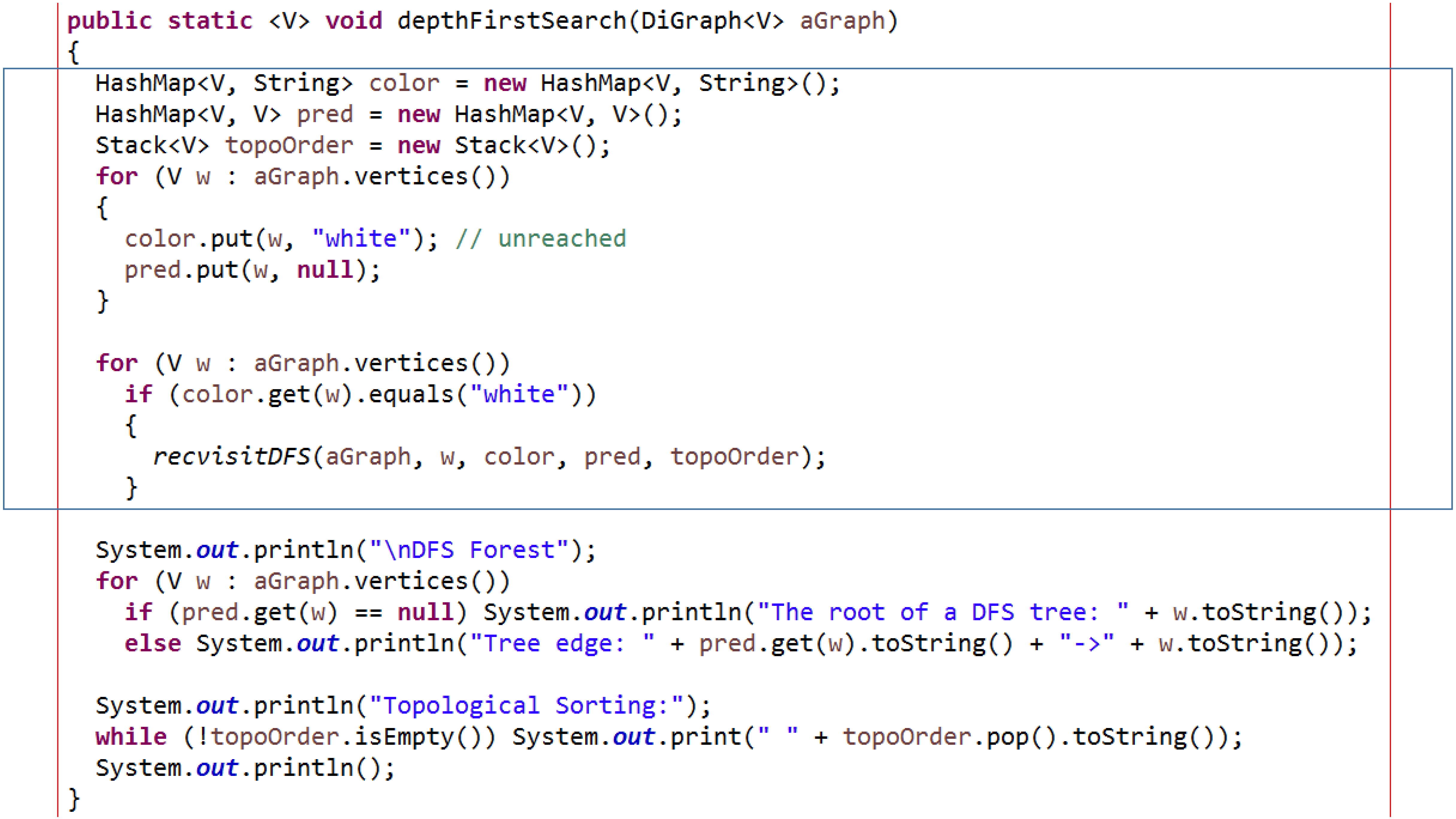
```java
// is f-t an edge in the graph?
public boolean hasEdge(V f, V t)
{
  if (!hasVertex(f))
    return false;
  for (Edge<V, Integer> e : map.get(f))
  {
    if (t.equals(e.getVertex()))
      return true;
  }
  return false;
}
```



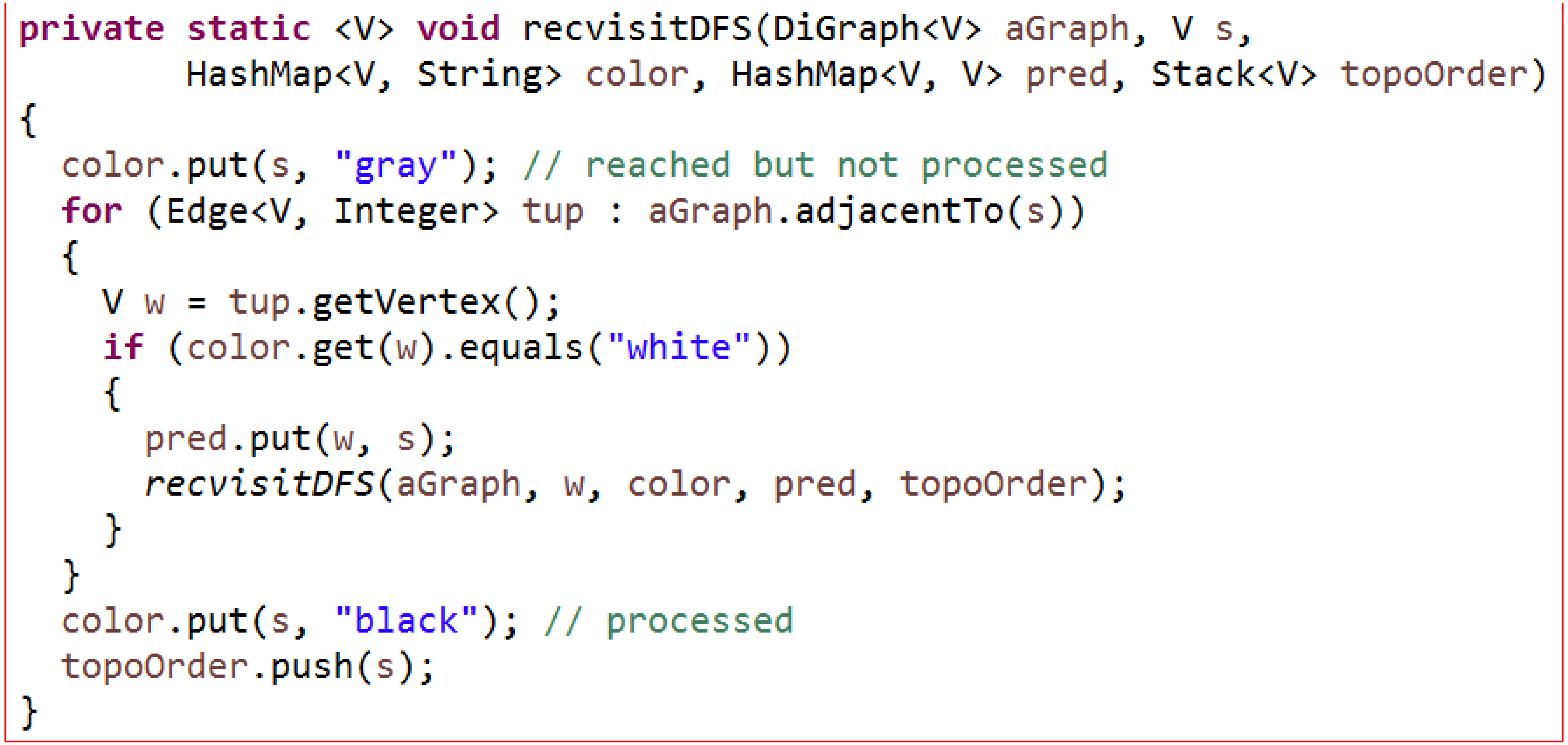
```java
public String toString()
{
  StringBuilder s = new StringBuilder("");
  for (V f : map.keySet())
  {
    s.append(f.toString() + ": ");
    for (Edge<V, Integer> e : map.get(f))
    {
      s.append("[" + e.getVertex().toString() +
        ", " + e.getCost().toString() + "] ");
    }
    s.append("\n");
  }
  return s.toString();
}
```



```
public static <V> void Dijkstra(DiGraph<V> G, V source)
{
  HashMap<V, Integer> dist = new HashMap<>();
  HashMap<V, V> pred = new HashMap<>();
  MinHeap<Edge<V, Integer>> minHeap = new MinHeap<>();
  HashSet<V> setT = new HashSet<V>();
  dist.put(source, 0);
  minHeap.add(new Edge<V, Integer>(source, 0));
  while (!minHeap.isEmpty()) {
    Edge<V, Integer> pair = minHeap.removeMin();
    V u = pair.getVertex();
    if (!setT.contains(u)) {
      setT.add(u);
      for (Edge<V, Integer> tup : G.adjacentTo(u)) {
        V v = tup.getVertex();
        Integer altdist = dist.get(u) + tup.getCost();
        Integer vdist = dist.get(v);
        if (vdist == null || vdist > altdist) {
          dist.put(v, altdist);
          pred.put(v, u);
          minHeap.add(new Edge<V, Integer>(v, altdist));
        }
      }
    }
  }
}
```
]



```
public static <V> void depthFirstSearch(DiGraph<V> aGraph)
{
 HashMap<V, String> color = new HashMap<V, String>();
 HashMap<V, V> pred = new HashMap<V, V>();
 Stack<V> topoOrder = new Stack<V>();
 for (V w : aGraph.vertices())
 {
  color.put(w, "white"); // unreached
  pred.put(w, null);
 }

 for (V w : aGraph.vertices())
  if (color.get(w).equals("white"))
  {
   recvisitDFS(aGraph, w, color, pred, topoOrder);
  }

 System.out.println("\nDFS Forest");
 for (V w : aGraph.vertices())
  if (pred.get(w) == null) System.out.println("The root of a DFS tree: " + w.toString());
  else System.out.println("Tree edge: " + pred.get(w).toString() + "->" + w.toString());

 System.out.println("Topological Sorting:");
 while (!topoOrder.isEmpty()) System.out.print(" " + topoOrder.pop().toString());
 System.out.println();
}
```



```java
private static <V> void recvisitDFS(DiGraph<V> aGraph, V s,
        HashMap<V, String> color, HashMap<V, V> pred, Stack<V> topoOrder)
{
  color.put(s, "gray"); // reached but not processed
  for (Edge<V, Integer> tup : aGraph.adjacentTo(s))
  {
    V w = tup.getVertex();
    if (color.get(w).equals("white"))
    {
      pred.put(w, s);
      recvisitDFS(aGraph, w, color, pred, topoOrder);
    }
  }
  color.put(s, "black"); // processed
  topoOrder.push(s);
}
```