18: GRAPH DATA STRUCTURES

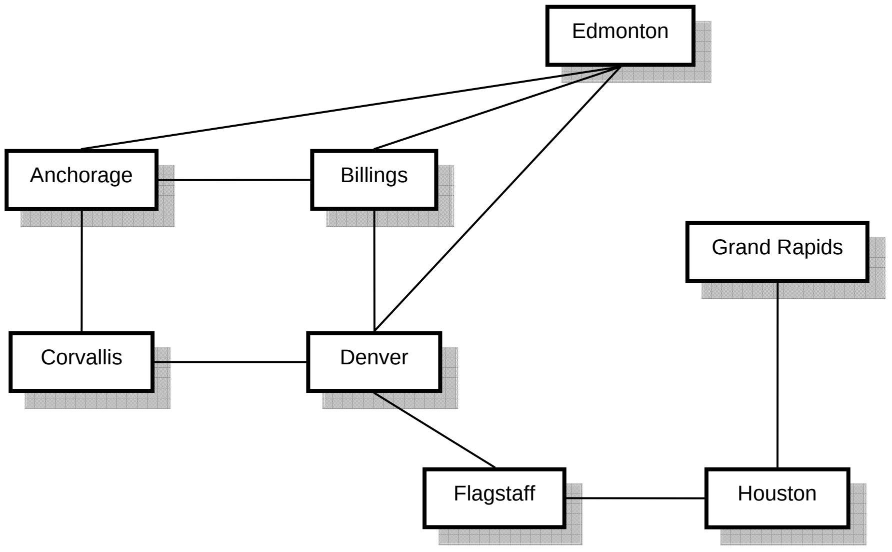
Introduction

We looked previously at the binary tree data structure, which provides a useful way of storing data for efficient searching. In a binary tree, each node can have up to two child nodes. More general tree structures can be created in which different numbers of child nodes are allowed for each node.

All tree structures are **hierarchical**. This means that each node can **only have one parent node**. Trees can be used to store data which has a definite hierarchy; for example a family tree or a computer file system.

Some data need to have connections between items which **do not fit into a hierarchy** like this. **Graph** data structures can be useful in these situations. A graph consists of a number of data items, each of which is called a **vertex**. Any vertex may be connected to any other, and these connections are called **edges**.

The following figure shows a graph in which the vertices are the names of cities in North America. The edges could represent flights between these cities, or possibly Wide Area Network links between them.



Describing graphs

The graph in the figure above is known as an **undirected graph**.

An undirected graph is **complete** if it has as many edges as possible – in other words, if every vertex is joined to every other vertex. The graph in the figure is not complete. For a complete graph with n vertices, the number of edges is $n(n-1)/2$. A graph with 6 vertices needs 15 edges to be complete.

Two vertices in a graph are **adjacent** if the form an edge. For example, Anchorage and Corvallis are adjacent, while Anchorage and Denver are not. Adjacent vertices are called **neighbours**.

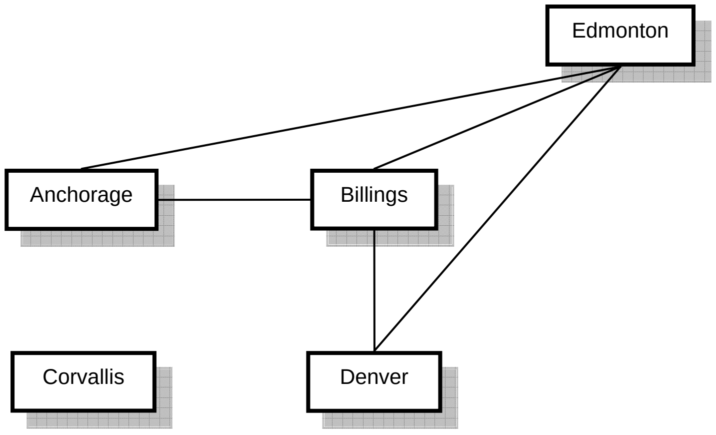
A **path** is a sequence of vertices in which each successive pair is an edge. For example:

Anchorage-Billings-Denver-Edmonton-Anchorage

A **cycle** is a path in which the first and last vertices are the same and there are no repeated edges. For example:

Anchorage-Billings-Denver-Flagstaff

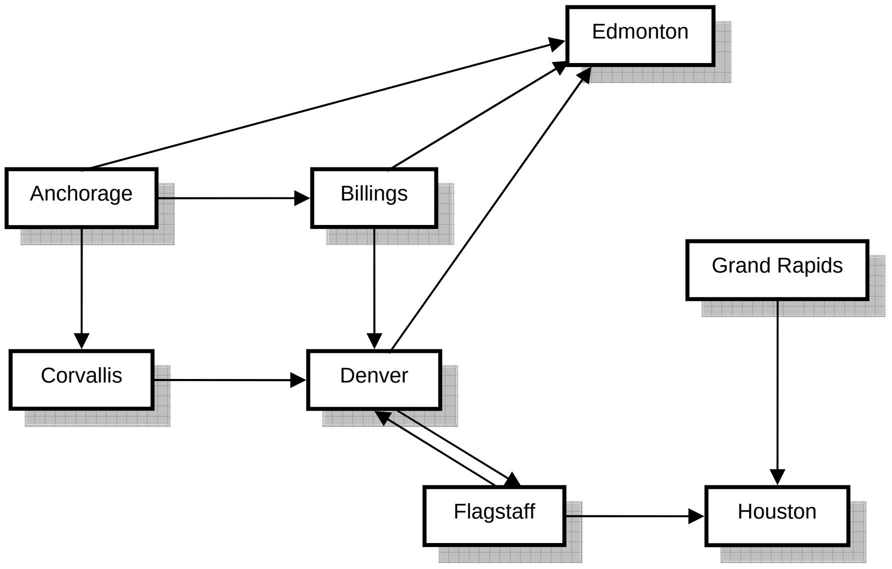
An undirected graph is **connected** if, for any pair of vertices, there is a path between them. The graph above is connected, while the following one is not, as there are no paths to Corvallis.



A **tree** data structure can be described as a **connected, acyclic graph** with one element designated as the root element. It is *acyclic* because there are no paths in a tree which start and finish at the same element.

Directed Graphs

In a **directed graph**, or **digraph**, each edge is an ordered pair of vertices – it has a direction defined. The direction is indicated by an arrow:



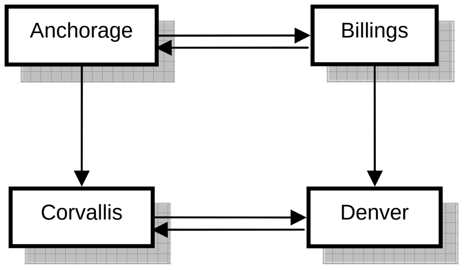
A **path** in a directed graph **must follow the direction of the arrows**. Note that there are two edges in this example between Denver and Flagstaff, so it is possible to travel in either direction. The following is a path in this graph

Billings-Denver-Flagstaff

while the following is not, because there is no edge from Denver to Billings:

Flagstaff-Billings-Denver

A directed graph is **connected** if, for any pair of vertices, there is a path between them. The following example graph is not connected – can you see why? What single edge could you change to make it connected?



Traversing a graph

Traversal is the facility to move through a structure visiting each of the vertices once. We looked previously at the ways in which a binary tree can be traversed. Two possible traversal methods for a graph are **breadth-first** and **depth-first**.

Breadth-First Traversal

This method visits all the vertices, beginning with a specified **start vertex**. It can be described roughly as "neighbours-first". No vertex is visited more than once, and vertices are only visited if they can be reached – that is, if there is a path from the start vertex.

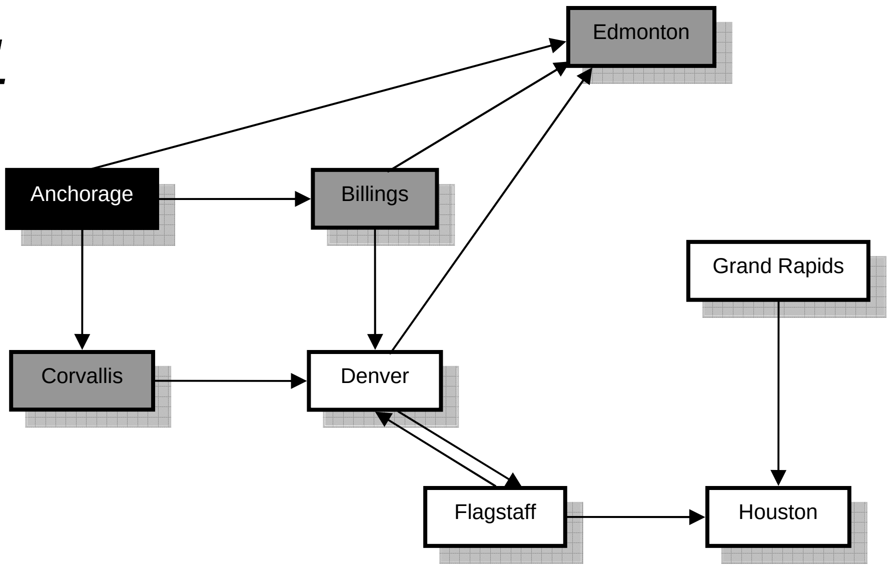
Breadth-first traversal makes use of a **queue data structure**. The queue holds a list of vertices which have not been visited yet but which should be visited soon. Since a queue is a first-in first-out structure, vertices are visited in the order in which they are added to the queue.

Visiting a vertex involves, for example, outputting the data stored in that vertex, and also **adding its neighbours to the queue**. Neighbours are not added to the queue if they are already in the queue, or have already been visited.

The following example shows a **breadth-first traversal starting at Anchorage**. Neighbours are added to the queue in alphabetical order. Visited and queued vertices are shown as follows:

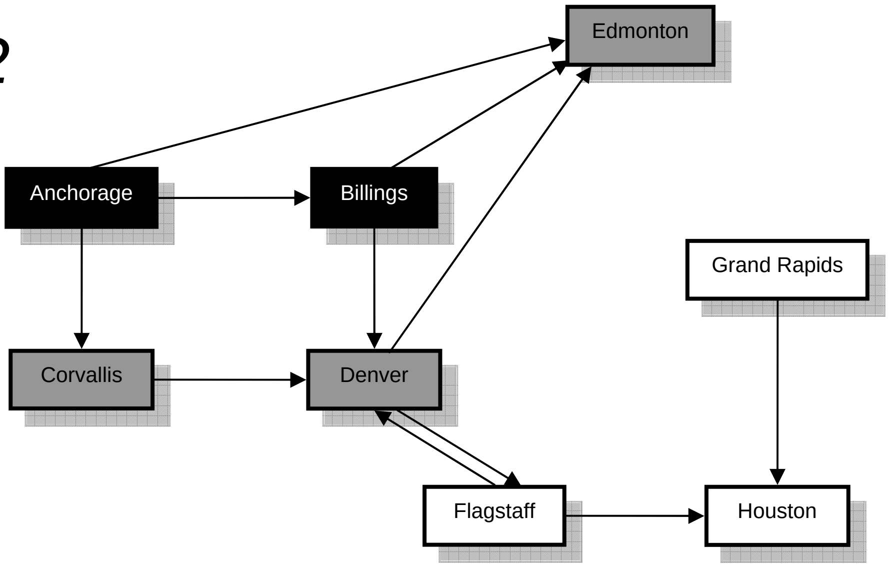
Billings
visited
Billings
queued

1



Visited: *Anchorage*
Queue: *Billings, Corvallis, Edmonton* visit *Billings* next

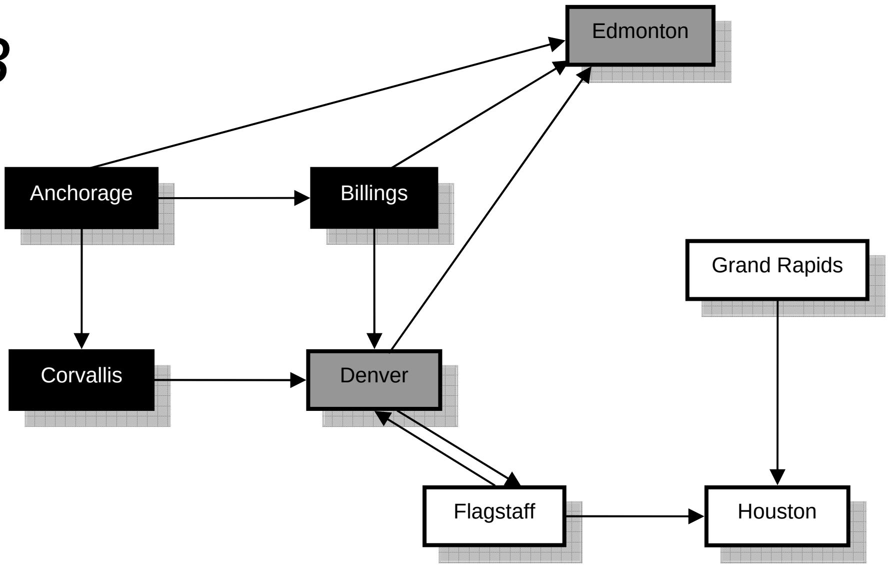
2



Visited: *Anchorage, Billings*
Queue: *Corvallis, Edmonton, Denver* visit *Corvallis* next

Note that we only add Denver to the queue as the other neighbours of Billings are already in the queue.

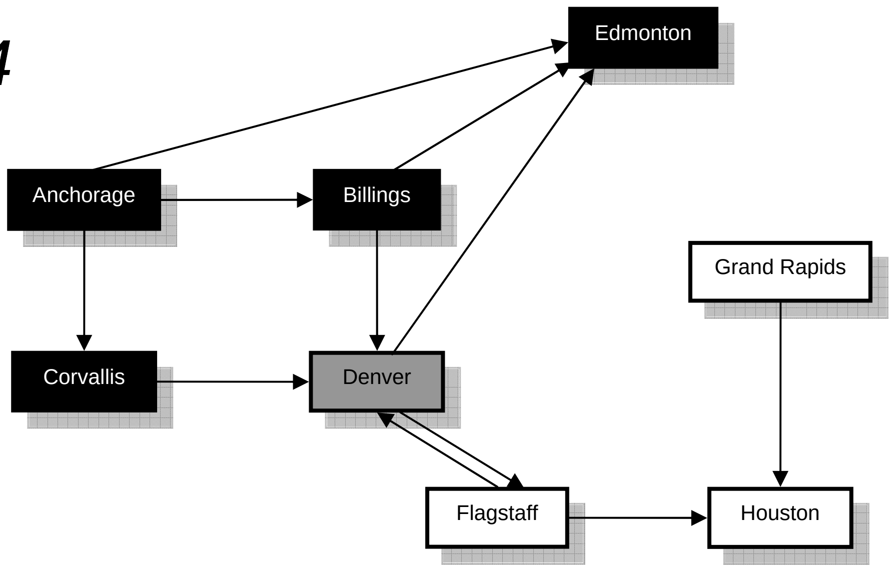
3



Visited: *Anchorage, Billings, Corvallis*
Queue: *Edmonton, Denver* visit *Edmonton* next

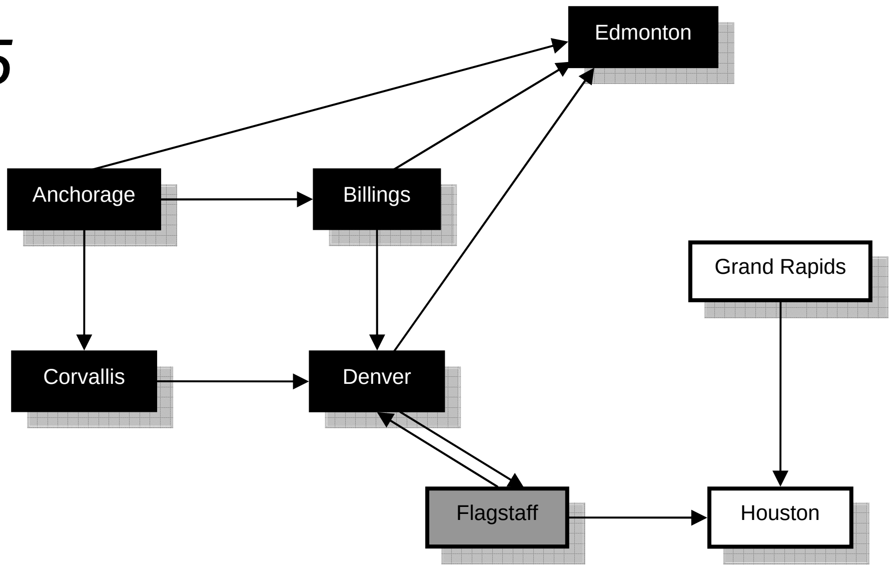
Note that nothing is added to the queue as Denver, the only neighbour of Corvallis, is already in the queue.

4

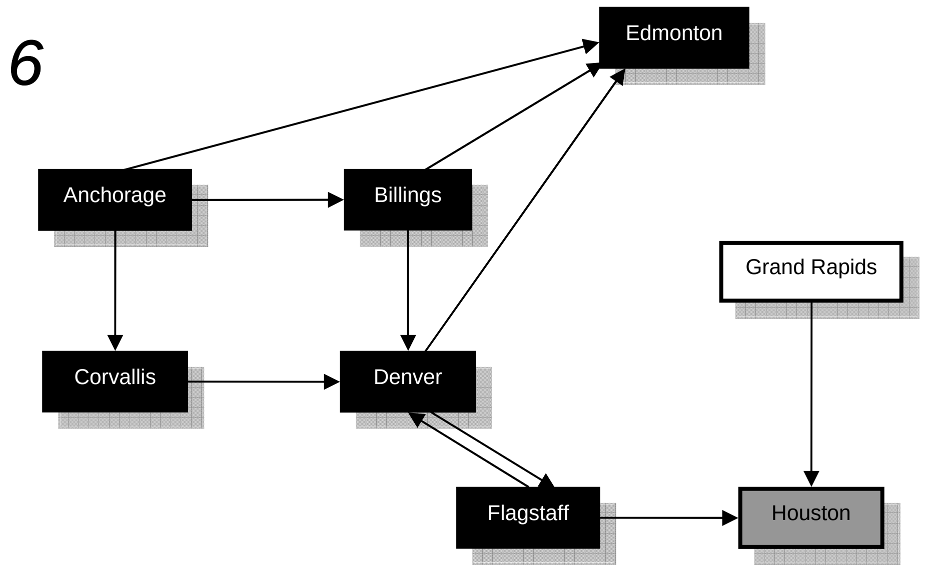


Visited: *Anchorage, Billings, Corvallis, Edmonton*
Queue: *Denver* visit *Denver* next

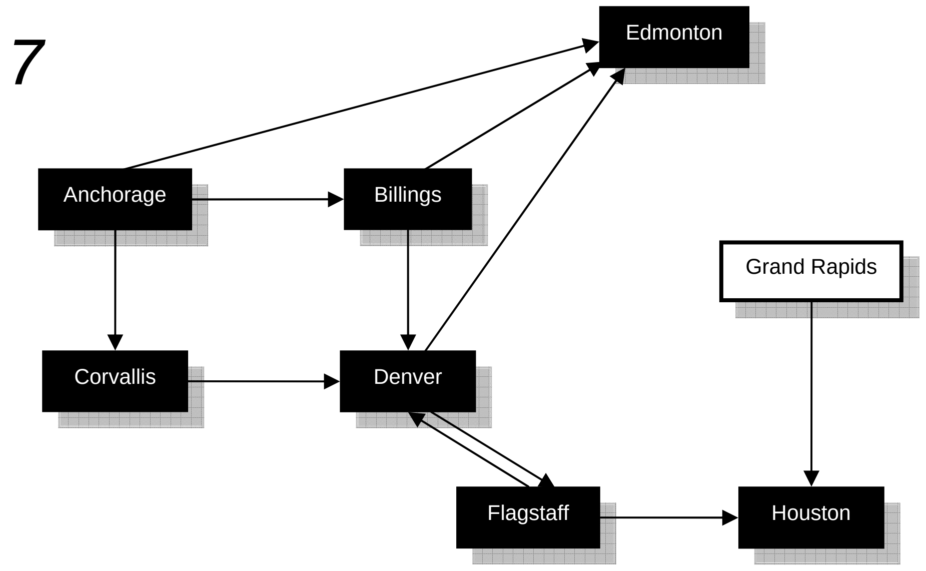
5



Visited: *Anchorage, Billings, Corvallis, Edmonton, Denver*
Queue: *Flagstaff* visit *Flagstaff* next



Visited: *Anchorage, Billings, Corvallis, Edmonton, Denver, Flagstaff*
Queue: *Houston* visit *Houston* next



Visited: *Anchorage, Billings, Corvallis, Edmonton, Denver, Flagstaff, Houston*
Queue: *empty*

Note that Grand Rapids was not added to the queue as there is no path from Houston because of the edge direction. Since the queue is empty, we must stop, so the traversal is complete. The order of traversal was:

Anchorage, Billings, Corvallis, Edmonton, Denver, Flagstaff, Houston

EXERCISE: Traversal

1. Find the order of breadth-first traversals of the graph in the example starting at (a) Billings and (b) Flagstaff

2. A **depth-first traversal** works in a similar way, except that the neighbours of each visited vertex are added to a **stack data structure**. Vertices are visited in the order in which they are popped from the stack, i.e. last-in, first-out.

   Find the order of a depth-first traversal of the graph in the example starting at Anchorage[1].

[1] *Answer should be: Anchorage, Edmonton, Corvallis, Denver, Flagstaff, Houston, Billings*

Implementing a Graph

The diagrams we have seen of graphs show the data and connections in a visual way. To make a Java Graph class, we have to work out a way in which that information can actually be stored and accessed. This is known as the **internal representation**.

There are several possible internal representations for a graph data structure (this is also true for binary trees). We will look at one which stores information as follows:

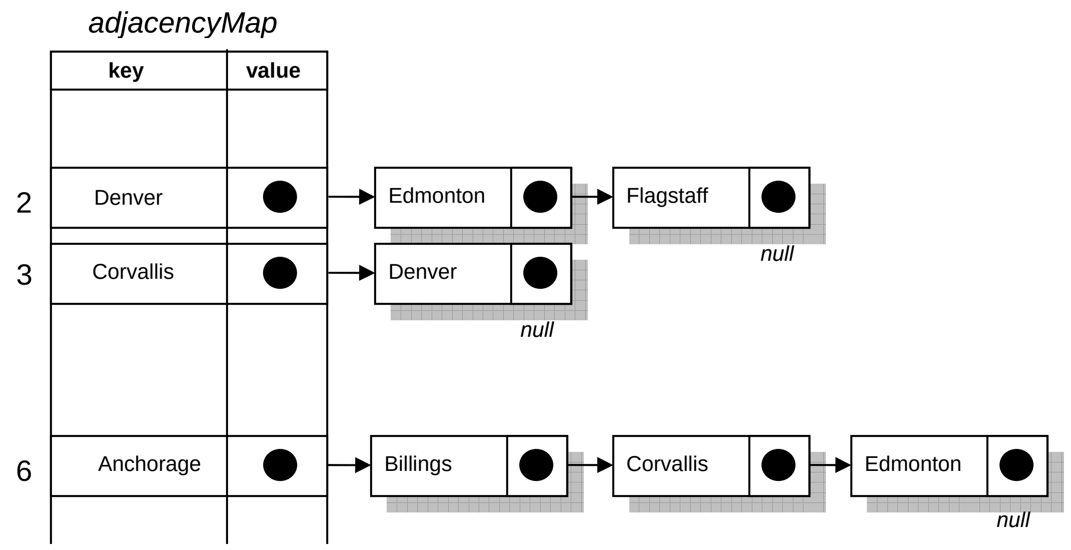
Vertices are stored as **keys in a Map structure** – this means a vertex can be quickly looked up. This Map is known as the **adjacency map**.

Edges starting from each vertex are stored as **a List of the adjacent vertices**. This List is stored as the **value** associated with the appropriate key in the Map.

For example, the representation of the graph used in the examples above would consist of a Map with the following entry representing *Anchorage*:

Key: "Anchorage"
Value: ["Billings", "Corvallis", "Edmonton"]

If the adjacency map is a *HashMap*, and the edges for each vertex are stored in a *LinkedList*, then the following diagram shows part of the internal representation of the example graph:



The following code shows a basic graph class. The *HashMap* and *LinkedList* classes are the ones you have used in previous chapters. Alternatively, you could use the equivalent Java Collections Framework classes.

```
/**
 * class Graph
 *
 * @author Jim
 * @version 1.0
 */
public class Graph
{
    protected HashMap adjacencyMap;

   /**
    *  Initialize this Graph object to be empty.
    */
    public Graph()
    {
        adjacencyMap = new HashMap();
    }

    /**
     *  Determines if this Graph contains no vertices.
     *
     *  @return true - if this Graph contains no vertices.
     */
    public boolean isEmpty()
    {
            return adjacencyMap.isEmpty();
    }

   /**
     *  Determines the number of vertices in this Graph.
     *
     *  @return the number of vertices.
     */
    public int size()
    {
            return adjacencyMap.size();
    }
   /**
     *  Returns the number of edges in this Graph object.
     *
     *  @return the number of edges.
     */
    public int getEdgeCount()
    {
        int count = 0;
        for (int i=0;i<adjacencyMap.CAPACITY;i++){
            if (adjacencyMap.keys[i] != null){
               LinkedList edges = (LinkedList)
                    adjacencyMap.get(adjacencyMap.keys[i]);
               count += edges.size();
            }
        }
        return count;
    }

   /**
     *  Adds a specified object as a vertex
     *
     *  @param vertex - the specified object
     *  @return true - if object was added by this call
     */
    public boolean addVertex (Object vertex)
    {
            if (adjacencyMap.containsKey(vertex))
                    return false;
            adjacencyMap.put (vertex, new LinkedList());
            return true;
    }

   /**
      *  Adds an edge, and vertices if not already present
      *
      *  @param v1 - the beginning vertex object of the edge
      *  @param v2 - the ending vertex object of the edge
      *  @return true - if the edge  was added by this call
      */
    public boolean addEdge (Object v1, Object v2)
    {
            addVertex (v1); addVertex (v2);
            LinkedList l = (LinkedList)adjacencyMap.get(v1);
            l.add(v2);
            return true;
    }
}
```

EXERCISE: Looking at a Graph

Create a new BlueJ project called *simplegraph*. Add a new class *Graph* using the above code. Add the *HashMap* class from your *simplehashmap* project. Add the *List*, *Node* and *LinkedList* classes from your *simplelist* project.

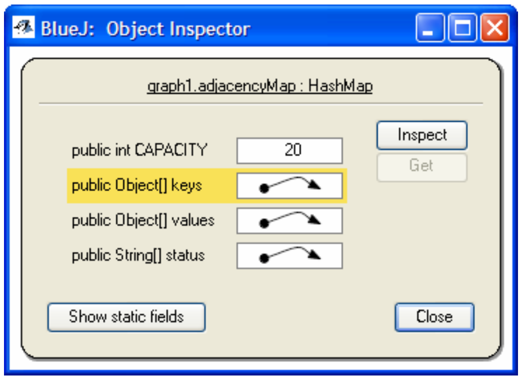
Create a new instance of *Graph* called *graph1*. Call the *addEdge* method repeatedly to add the following edges (this should construct a directed graph equivalent to the example used earlier in this chapter):

```
addEdge("Anchorage", "Billings");
addEdge("Anchorage", "Corvallis");
addEdge("Anchorage", "Edmonton");
addEdge("Billings", "Denver");
addEdge("Billings", "Edmonton");
addEdge("Corvallis", "Denver");
addEdge("Denver", "Edmonton");
addEdge("Denver", "Flagstaff");
addEdge("Flagstaff", "Denver");
addEdge("Flagstaff", "Houston");
addEdge("Grand Rapids", "Houston");
```

Call the *size* and *getEdgeCount* methods of *graph1*.

What results did you get? Are these correct?

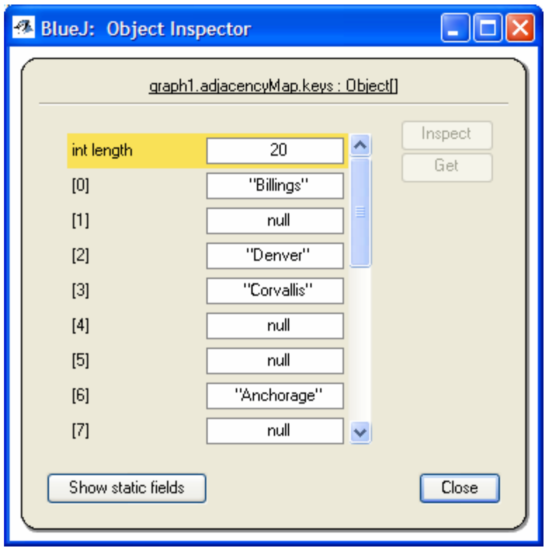
Inpsect *graph1*. The only field is *adjacencyMap*. Click the **Inspect** button in the **Object Inspector** to inspect *adjacencyMap*. This allows you to access the keys and the values stored in the map.



graph1.adjacencyMap : HashMap

public int CAPACITY 20
public Object[] keys
public Object[] values
public String[] status

Inspect the *keys* array. Check that *"Anchorage"* is included. Note its position (6 in this screenshot).



BlueJ: Object Inspector

graph1.adjacencyMap.keys : Object[]

int length	20
[0]	"Billings"
[1]	null
[2]	"Denver"
[3]	"Corvallis"
[4]	null
[5]	null
[6]	"Anchorage"
[7]	null

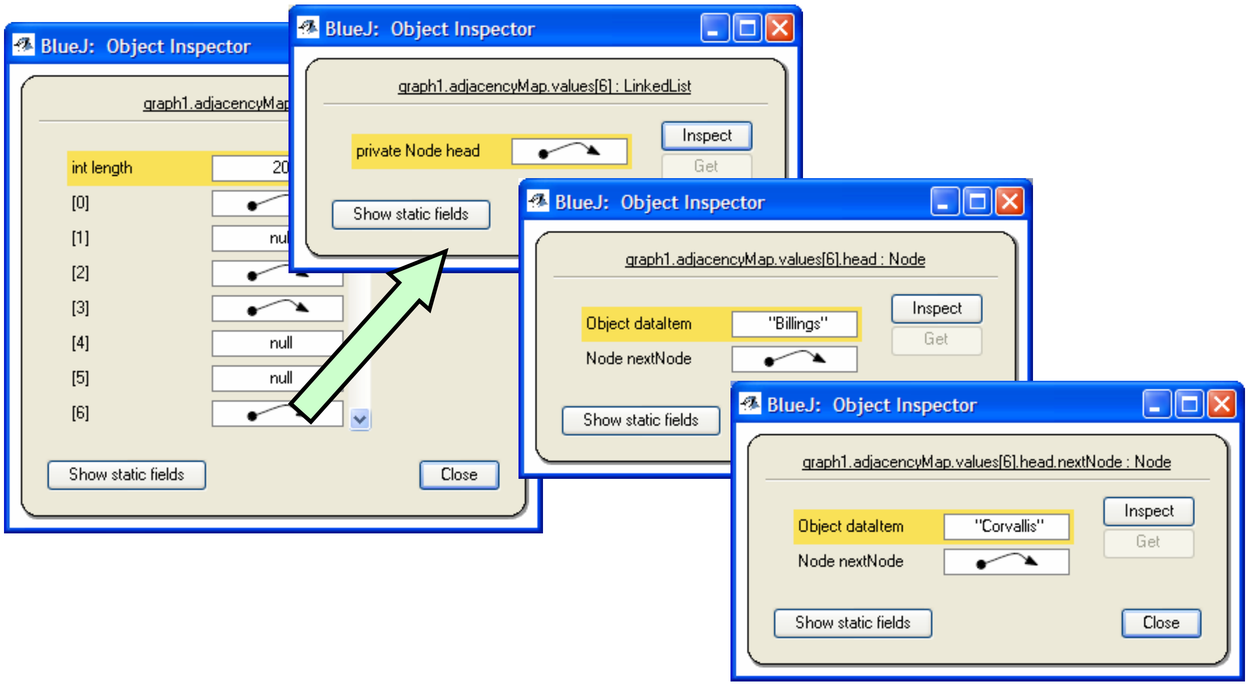
Inspect the *values* array. You should see an array which includes some object references. Inspect the object at the same position as *"Anchorage"* occupied in the keys array (6 in this case).

What kind of object is this?

Inspect this object further.

Compare with the figure on page 9.

What do you expect to find if you inspect the value object with the same position as "Denver"?



BlueJ: Object Inspector

graph1.adjacencyMap

int length	20
[0]	●→
[1]	null
[2]	●→
[3]	●→
[4]	null
[5]	null
[6]	●→

BlueJ: Object Inspector

graph1.adjacencyMap.values[6] : LinkedList

private Node head	●→

BlueJ: Object Inspector

graph1.adjacencyMap.values[6].head : Node

Object dataItem	"Billings"
Node nextNode	●→

BlueJ: Object Inspector

graph1.adjacencyMap.values[6].head.nextNode : Node

Object dataItem	"Corvallis"
Node nextNode	●→

EXERICISE: Implementing Traversal

Add the *Queue* class from your *queues* project. Add the following method to your *Graph* class[2]:

```
/**
  * Lists the vertices reached using a Breadth First
  * traversal with a specified starting point
  *
  * @param start - the starting vertex for the traversal
  */
  public void breadthFirstTraversal(Object start)
  {
      Queue queue = new Queue();
      HashMap reached = new HashMap();
      Object current;
      for (int i=0;i<adjacencyMap.CAPACITY;i++){
          if (adjacencyMap.keys[i] != null){
             reached.put(adjacencyMap.keys[i], false);
          }
      }
      queue.add(start);
      reached.set (start, true);

      while (!(queue.isEmpty()))
      {
          Object to;
          current = queue.remove();
          LinkedList edgeList = (LinkedList)
             adjacencyMap.get (current);

          for (int i=0; i<edgeList.size(); i++){
                  to = edgeList.get(i);
                  Boolean check =(Boolean)reached.get(to);
                  if (!check)
                  {
                      reached.set(to, true);
                      queue.add(to);
                  }
              }
          System.out.println("Vertex: " + current);
      }
   }
```

[2] requires J2SE 5.0 or later

Create a new instance of *Graph* called *graph1* and add the same edges as in the previous exercise.

Call the *breathFirstTraversal* method of graph1 and specify *"Anchorage"* as your start vertex.

> ***Compare the output with the worked example which starts on page 4 of this chapter.***

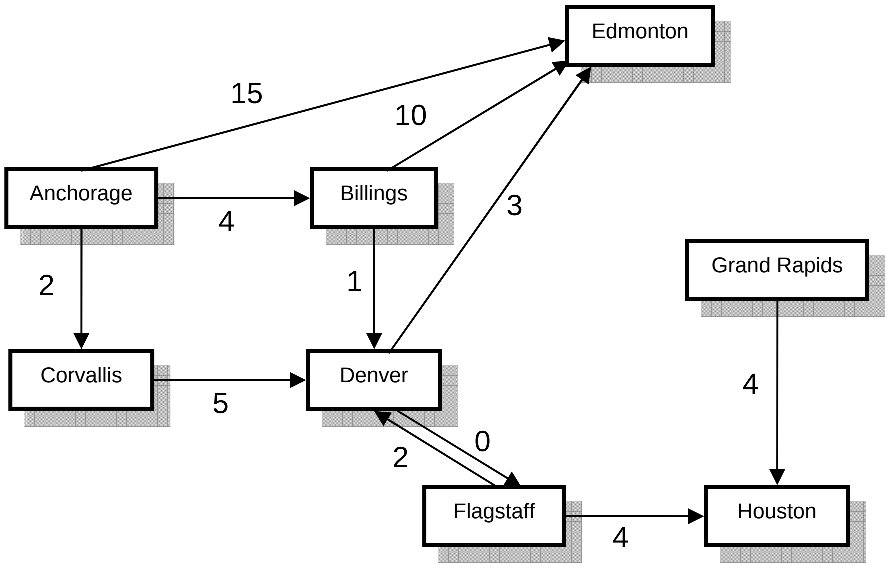
Try starting the traversal from *"Billings"* and from *"Flagstaff*".

> ***Compare the output with your previous worked examples.***

Add a new method *depthFirstTraversal* to your *Graph* class and test it. You will need to make use of the *Stack* class from your *stacks* project.

Networks

Sometimes the edges in a graph have numbers, or **weights**, associated with them. Weights in the example below could be based on, for example, costs of flights, or on WAN bandwidth. A graph like this is called a **weighted graph**, or a **network**.

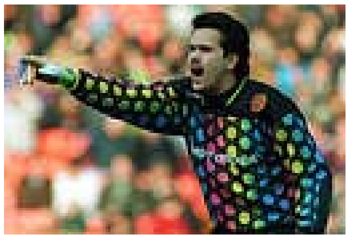


In a network, each path has a total weight. For example, the path *Anchorage-Billings-Edmonton* has a total weight of 4 + 10 = 14. This is in fact a shorter path than the direct path *Anchorage-Edmonton* which has a weight of 15.

Finding the shortest path between two vertices is often important in answering questions like *"What is the cheapest way to fly from Anchorage to Flagstaff?"* or *"What is the best way to route WAN traffic between Billings and Edmonton?"*

Algorithm for finding the shortest path

The shortest path can be found using **Dijkstra's algorithm**. This is similar to the breadth first traversal we looked at earlier in this chapter, except that it uses a special kind of queue data structure called a **priority queue**.



Dijkstra's algorithm is nothing to do with him!

In the priority queue, items are **removed in order of value** rather than in order of being added to the queue. When the target vertex is removed from the priority queue, the shortest path has been found.

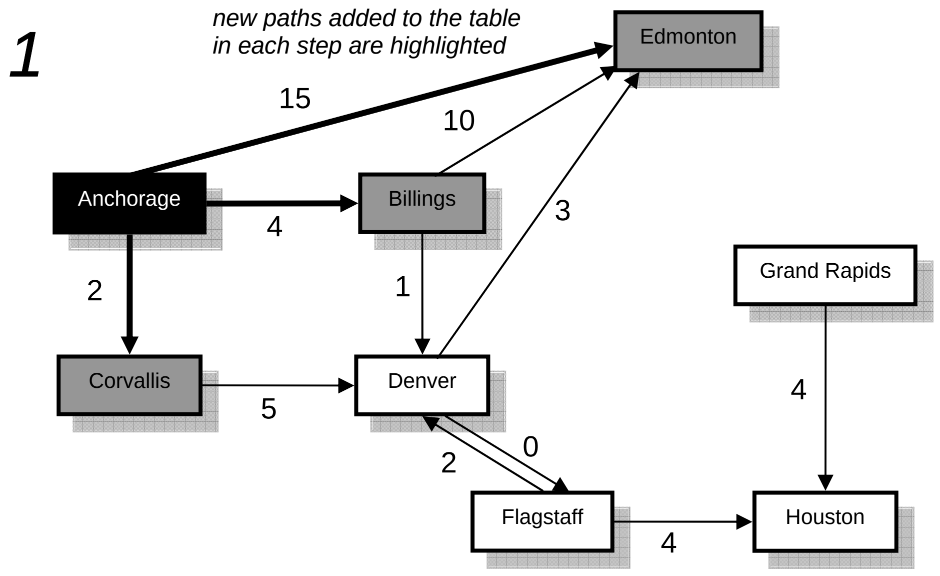
Example – find the shortest path from Anchorage to Edmonton.

We need to keep track of three things:

- The lowest total path weight, or **weightsum**, from the start point to each vertex
- The immediate **predecessor** in that path for each vertex
- The contents of the priority queue

We do not need to include Grand Rapids as it cannot be reached from Anchorage.

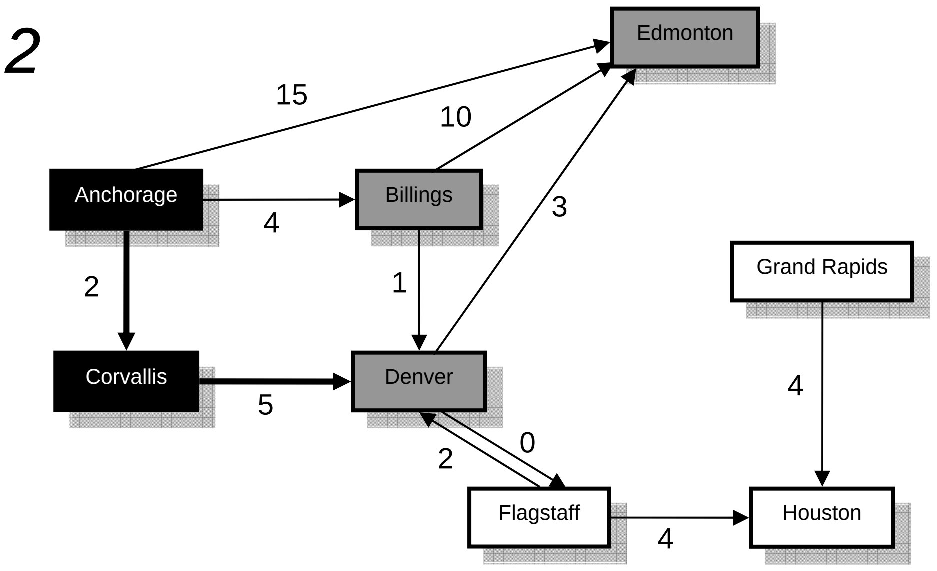
The priority queue contains vertices in order of weightsum value – the lowest is removed first. All weightsum are set to a large value to start with.



vertex	weightsum	predecessor
Anchorage	0	Anchorage
Billings	4	Anchorage
Corvallis	2	Anchorage
Denver	1000	null
Edmonton	15	Anchorage
Flagstaff	1000	null
Houston	1000	null

Priority Queue: *(Corvallis;2), (Billings;4), (Edmonton;15)*
Visit *Corvallis* next

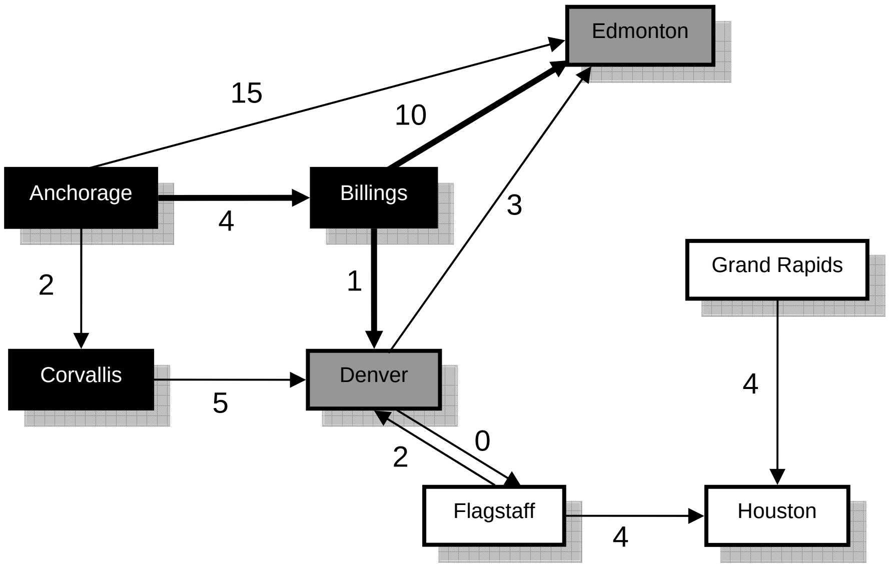
2



vertex	*weightsum*	*predecessor*
Anchorage	0	Anchorage
Billings	4	Anchorage
Corvallis	2	Anchorage
Denver	7	Corvallis
Edmonton	15	Anchorage
Flagstaff	1000	null
Houston	1000	null

Priority Queue: *(Billings;4), (Denver;7), (Edmonton;15)*
Visit *Billings* next

3



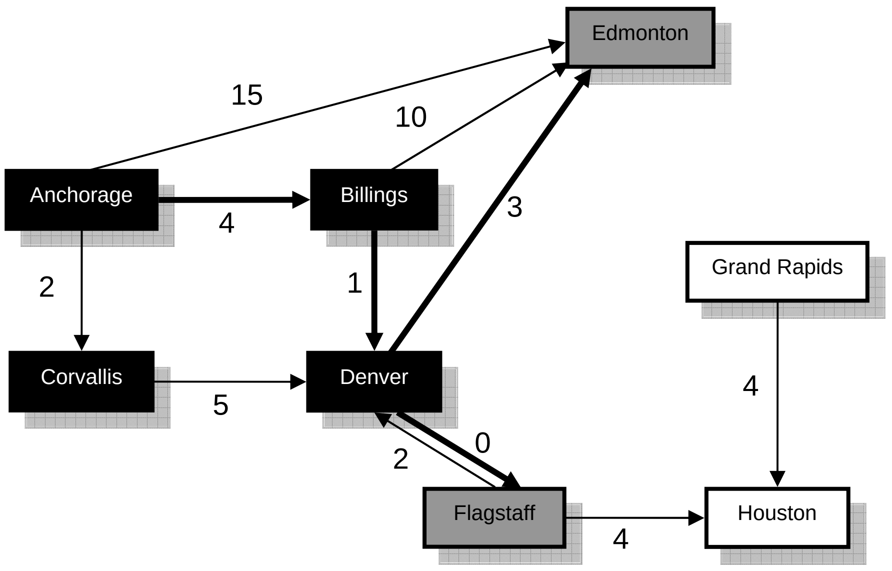
vertex	*weightsum*	*predecessor*
Anchorage	0	Anchorage
Billings	4	Anchorage
Corvallis	2	Anchorage
Denver	5	Billings
Edmonton	14	Billings
Flagstaff	1000	null
Houston	1000	null

Priority Queue: *(Denver;5), (Denver;7), (Edmonton;14), (Edmonton;15)*
Visit *Denver* next

Note that the new, shorter paths to Denver and Edmonton through Billings are added to the priority queue. They do not replace the previous paths, through Corvallis, but take priority over them because the weightsums are lower.

The weightsums and predecessors in the table are updated to take account of the new paths.

4

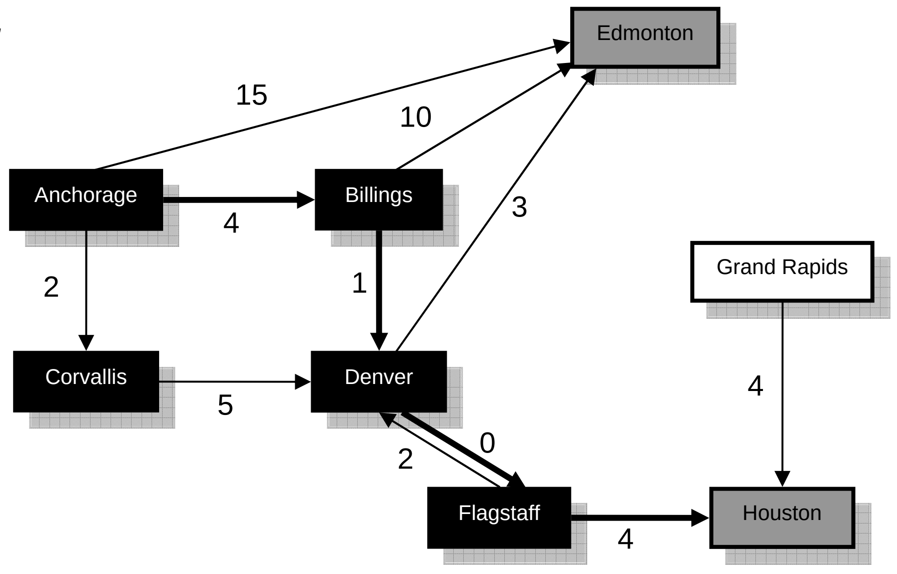


vertex	*weightsum*	*predecessor*
Anchorage	0	Anchorage
Billings	4	Anchorage
Corvallis	2	Anchorage
Denver	5	Billings
Edmonton	8	Denver
Flagstaff	5	Denver
Houston	1000	null

Priority Queue: *(Flagstaff;5), (Denver;7), (Edmonton;8), (Edmonton;14), (Edmonton;15)*
Visit *Flagstaff* next

Note that the new, shorter path to Edmonton through Denver is added to the priority queue, as is a path to Flagstaff.

5



vertex	weightsum	predecessor
Anchorage	0	Anchorage
Billings	4	Anchorage
Corvallis	2	Anchorage
Denver	5	Billings
Edmonton	8	Denver
Flagstaff	5	Denver
Houston	9	Flagstaff

Priority Queue: *(Denver;7), (Edmonton;8), (Houston;9), (Edmonton;14), (Edmonton;15)*
Visit *Denver* next

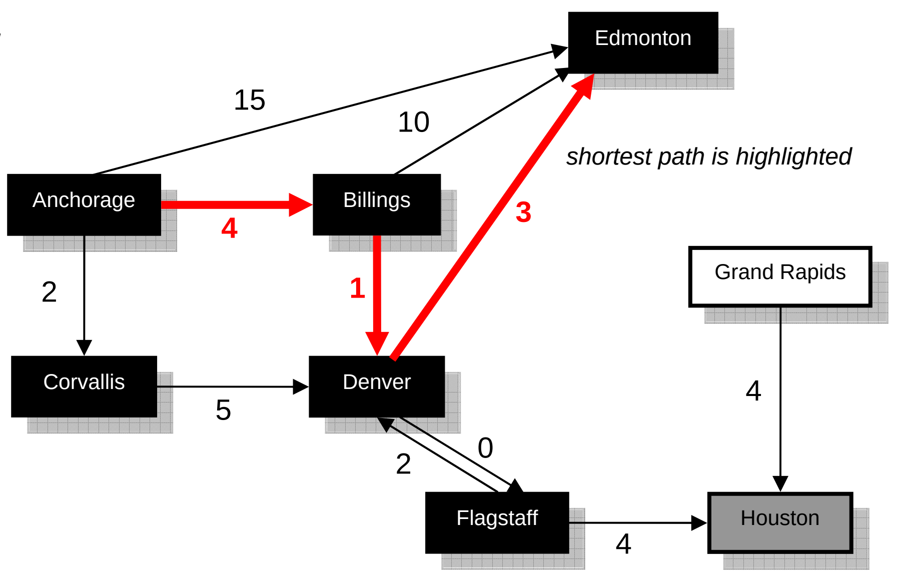
6 no change to diagram as Denver has already been visited

no change to table

Priority Queue: *(Edmonton;8), (Houston;9), (Edmonton;14), (Edmonton;15)*
Visit *Denver* next

Note that in this step we dequeued path to Denver with weightsum 7. This value is higher than the value of 5 in the table, so the table should not be updated.

7



vertex	*weightsum*	*predecessor*
Anchorage	0	Anchorage
Billings	4	Anchorage
Corvallis	2	Anchorage
Denver	5	Billings
Edmonton	**8**	**Denver**
Flagstaff	5	Denver
Houston	9	Flagstaff

Priority Queue: *(Houston;9), (Edmonton;14), (Edmonton;15)*

We have now removed a path, with weightsum 8, to the target, Edmonton, from the queue. **This must be the shortest path to Edmonton** as the shortest paths to any vertex are always removed first.

All we need to do now is to trace the path by looking at the predecessors in the table. The predecessor of Edmonton is Denver; the predecessor of Denver is Billings; the predecessor of Billings is Anchorage.

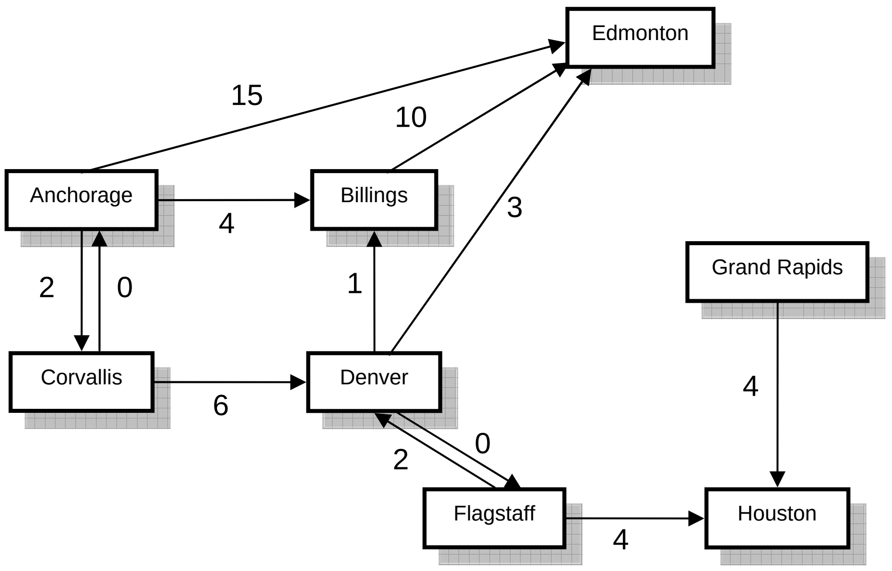
The **shortest path** is therefore:

Anchorage-Billings-Denver-Edmonton

with a **total weight of 8.**

EXERCISE: Using Dijkstra’s Algorithm

Use Dijkstra’s algorithm to find the **shortest** path from Corvallis to Edmonton in the following graph (note that the graph is a bit different from the one in the example).[3]



Further Reading

Data Structures and the Java Collections Framework by W.J. Collins includes a full Java implementation of a Network class which uses Dijkstra’s algorithm.

[3] Answer should be: Corvallis-Denver-Edmonton, total weight 9