Graph Implementations

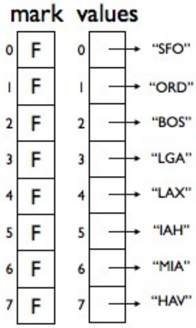
Representing Connectivity

- the abstract class AbstractGraph implements all methods of the abstract datatype (interface) Graph except those that implement connectivity
- public Iterator iterator (int node) must return an iterator object that allows for iteration over the edges emerging from the node
- public int cost (int src, int dest) must return the cost of the edge connecting nodes src and dest; it should return Graph.INFINITY if the nodes are not connected
- public void setCost (int s, int d, int v) must modify the cost of the existing edge between s and d to be v; or if no such edge exists, it should add the edge between s and d with cost v

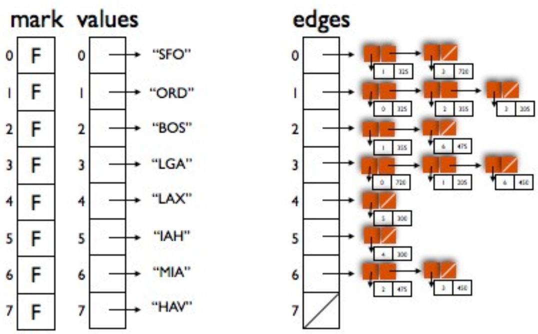
Data Structures for Connectivity

- two approaches for implementing graph connectivity are adjacency list and an adjacency matrix representations
- adjacency list representation maintains an array indexed by node number of linked lists of edges
- adjacency matrix representation maintains a two-dimensional array in which each element at i,j stores the cost of the edge between nodes i and j

Abstract Graph Representation



Adjacency List Representation




```
public class LGraph extends AbstractGraph implements Graph {
    private List[] edges;
    private static final int DESTINATION = 0;
    private static final int COST = 1;

    public LGraph (int max) {
        this.maxsize = max;
        this.size = 0;
        this.values = new Object[max];
        this.mark = new boolean[max];
        this.edges = new List[max];
        for (int i = 0; i < max; i++) {
            this.values[i] = null;
            this.mark[i] = false;
            this.edges[i] = List.NIL;
        }
    }
.
.
.
}
```

```
public class LGraph extends AbstractGraph implements Graph {
.
.
.
    public Iterator iterator (int node) {
        return new LGraphIterator(this.edges[node]);
    }
.
.
.
}
```

```
public class LGraph extends AbstractGraph implements Graph {
.
.
.
    public int cost (int src, int dest) {
        for (List p = this.edges[src];
             !p.isEmpty();
             p = p.tail()) {
            int[] current = (int[]) p.head();
            if (current[DESTINATION] == dest)
                return current[COST];
        }
        return Graph.INFINITY;
    }
.
.
.
}
```

```
public class LGraph extends AbstractGraph implements Graph {
.
.
.
    public void setCost (int src, int dest, int cost) {
        Object e = findEdge(src, dest);
        if (e == null) {
            int[] newEdge = new int[2];
            newEdge[DESTINATION] = dest;
            newEdge[COST] = cost;
            this.edges[src] = this.edges[src].push(newEdge);
        }
        else ((int[]) e)[COST] = cost;
    }
.
.
.
}
```

```java
public class LGraph extends AbstractGraph implements Graph {
.
.
.
    private Object findEdge (int src, int dest) {
       for (List p = this.edges[src];
            !p.isEmpty();
            p = p.tail()) {
           int[] current = (int[]) p.head();
           if (current[DESTINATION] == dest) return current;
       }
       return null;
    }
.
.
.
}
```

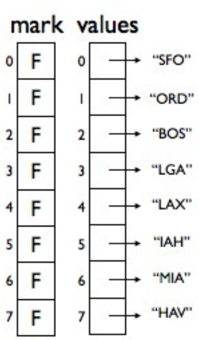
```java
public class LGraphIterator implements Iterator {
    private List edges;

    public LGraphIterator (List edges) {
        this.edges = edges;
    }

    public boolean hasNext() {
        return !this.edges.isEmpty();
    }

    public Object next () {
        if (this.edges.isEmpty())
            throw new NoSuchElementException();
        int[] result = (int[]) this.edges.head();
        this.edges = this.edges.tail();
        return result[0];
    }
}
```

Abstract Graph Representation



Adjacency Matrix Representation

	mark	values
0	F	"SFO"
1	F	"ORD"
2	F	"BOS"
3	F	"LGA"
4	F	"LAX"
5	F	"IAH"
6	F	"MIA"
7	F	"HAV"

cost

	0	1	2	3	4	5	6	7
0	∞	325	∞	720	∞	∞	∞	∞
1	325	∞	355	205	∞	∞	∞	∞
2	∞	355	∞	∞	∞	∞	475	∞
3	720	205	∞	∞	∞	∞	450	∞
4	∞	∞	∞	∞	∞	300	∞	∞
5	∞	∞	∞	∞	300	∞	∞	∞
6	∞	∞	475	450	∞	∞	∞	∞
7	∞	∞	∞	∞	∞	∞	∞	∞

```java
public class AGraph extends AbstractGraph implements Graph {
    private int[][] cost;

    public AGraph (int max) {
        this.maxsize = max;
        this.size = 0;
        this.values = new Object[max];
        this.mark = new boolean[max];
        this.cost = new int[max][max];

        for (int i = 0; i < max; i++) {
            this.values[i] = null;
            this.mark[i] = false;
            for (int j = 0; j < max; j++)
                this.cost[i][j] = Graph.INFINITY;
        }
    }
.
.
.
}
```

```java
public class AGraph extends AbstractGraph implements Graph {
.
.
.
    public AGraph (LGraph g) {
        this(g.size);
        this.size = g.size;

        for (int i = 0; i < this.size; i++) {
            this.values[i] = g.value(i);
            this.mark[i] = g.marked(i);
        }

        for (int src = 0; src < this.size; i++) {
            Iterator iter = g.iterator(src);
            while (iter.hasNext()) {
                int dest = (Integer) iter.next();
                addEdge(src, dest, g.cost(src, dest));
            }
        }
    }
.
.
.
}
```

```
public class AGraph extends AbstractGraph implements Graph {
.
.
.
    public Iterator iterator (int node) {
        return new AGraphIterator(this.cost, node);
    }

    public int cost (int i, int j) {
        return this.cost[i][j];
    }

    public void setCost (int src, int dest, int value) {
        this.cost[src][dest] = value;
    }
.
.
.
}
```

```
public class AGraphIterator implements Iterator {
    private int[][] cost;
    private int src;
    private int next;

    public AGraphIterator (int[][] cost, int src) {
        this.cost = cost; this.src = src; this.next = 0;
    }

    public boolean hasNext() {
        for (int dest = this.next;
             dest < this.cost[this.src].length;
             dest++)
            if (this.cost[this.src][dest] != Graph.INFINITY)
                return true;
        return false;
    }

    public Object next () {
       while (this.next < this.cost[this.src].length)
         if (this.cost[this.src][this.next] != Graph.INFINITY)
             return next++;
       throw new NoSuchElementException();
    }
}
```

Pros and Cons

- disadvantage of matrix implementation is requirement for $O(n^2)$ space vs $O(n)$
- advantage of matrix implementation is fast cost lookup $O(1)$
- sparse graphs (few edges) are better implemented with lists