# Abstract Graphs

## Data Structures for Sets

- abstract sets are collections of elements
- sets implemented with arrays provide random access to elements
- sets implemented with linked lists provide dynamic growth but linear access
- sets implemented with binary trees provide dynamic growth with log n access

## Summary of Set Implementations

| | add | find | memory |
|---|---|---|---|
| unsorted array | O(1) | O(n) | static |
| sorted array | O(n) | O(log n) | static |
| linked list | O(1) | O(n) | dynamic |
| binary tree | O(log n) | O(log n) | dynamic |

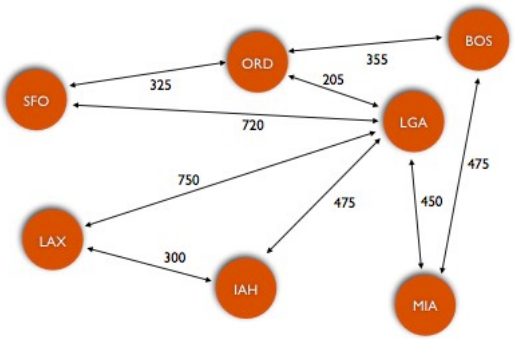
## Graphs

- graphs are data structures for relationships rather than collections
- graphs represent values and relationships among the values
- airline travel routes — connections between cities and associated costs
- social networks — connections between people
- the world wide web — connections between pages

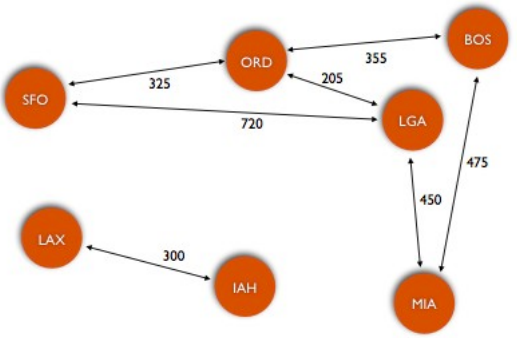
## Definition

- G = (N, E)
- N = { n1, n2, ... }
- E = { ($n_{i1}$, $n_{j1}$), ($n_{i2}$, $n_{j2}$), ... }
- a directed graph is an ordered pair (N, E) where N is a set of nodes, and E is a set of ordered pairs of nodes (i.e. edges)
- in an undirected graph, the edges are pairs rather than ordered pairs

## Flight Graph with One Connected Component



## Flight Graph with Two Connected Component



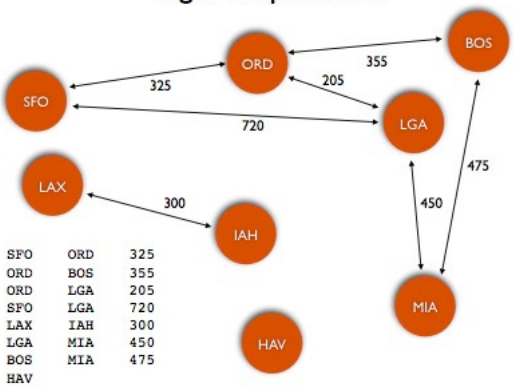

## Abstract Datatype

- a graph has a set of nodes and a set of edges
- its size is the number of nodes; each node identified by an integer 0..size-1
- nodes and edges can be added to the graph
- each node can be marked
- each node can have an associated value (any object)
- edges off of a node can be iterated over
- each edge can have a cost
- a node's degree is the number of edges off of it
- a graph whose node values are strings can be read from or written to a stream

## Graphs in Streams

- node values are string names
- each line in the stream is an edge represented as a tab separated list with source node name, destination node name, cost
- nodes with no edges are listed on their own lines as just node name

## Flight Graph Stream




```
SFO     ORD     325
ORD     BOS     355
ORD     LGA     205
SFO     LGA     720
LAX     IAH     300
LGA     MIA     450
BOS     MIA     475
HAV
```

```
public interface Graph {
    public static final int INFINITY = Integer.MAX_VALUE;
    public static final int FAIL = -1;

    public boolean marked (int node);
    public void mark (int node);
    public void clear (int node);

    public int size ();

    public Object value (int node);
    public int find (Object value);

    public Iterator iterator (int node);

    public int cost (int src, int dest);
    public void setCost (int src, int dest, int cost);

    public boolean connected (int src, int dest);

    public int degree (int node);

    public int addNode (Object o);
    public void addEdge (int src, int dest, int cost);

    public void read (BufferedReader in) throws IOException;
    public void read () throws IOException;
    public void write (PrintStream out);
    public void write ();

}
```

```
public abstract class AbstractGraph implements Graph {
    protected int maxsize;
    protected int size;
    protected Object[] values;
    protected boolean[] mark;

    abstract public Iterator iterator (int node);
    abstract public int cost (int src, int dest) ;
    abstract public void setCost (int s, int d, int v) ;
.
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
.
    public boolean marked (int node) {
        return this.mark[node];
    }

    public void mark (int node) {
        this.mark[node] = true;
    }

    public void clear (int node) {
        this.mark[node] = false;
    }
.
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
.
    public int size () { return this.size; }

    public Object value (int node) {
        return this.values[node];
    }

    public int find (Object o) {
        for (int i = 0; i < this.size; i++)
            if (this.values[i].equals(o)) return i;
        return Graph.FAIL;
    }
.
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
.
    public boolean connected (int src, int dest) {
        return this.cost(src, dest) != Graph.INFINITY;
    }

    public int degree (int node) {
        int count = 0;

        for (Iterator i = this.iterator(node);
             i.hasNext();
             i.next())
            count++;

        return count;
    }
.
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
.
    public int addNode (Object o) {
        if (this.size == this.maxsize) return Graph.FAIL;
        this.values[size] = o;
        return size++;
    }

    public void addEdge (int src, int dest, int cost) {
        setCost(src, dest, cost);
    }
.
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
.
    private int findOrAdd (Object o) {
        int i = this.find(o);
        if (i != Graph.FAIL) return i;
        return addNode(o);
    }
.
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
    public void read (BufferedReader in) throws IOException {
        final int SOURCE = 0;
        final int DESTINATION = 1;
        final int COST = 2;
        String line;
        while ((line = in.readLine()) != null &&
               line.length() != 0) {
            String sdc[] = line.split("\t");
            int src = this.findOrAdd(sdc[SOURCE]);
            if (src == Graph.FAIL)
                throw new IOException("Graph too large.");
            if (sdc.length > 1) {
                int dest = this.findOrAdd(sdc[DESTINATION]);
                if (dest == Graph.FAIL)
                    throw new IOException("Graph too large.");
                addEdge(src, dest, Integer.parseInt(sdc[COST]));
            }
        }
    }
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
    private void writeEdges (PrintStream p, int src) {
        Iterator i = this.iterator(src);
        int dest;
        while (i.hasNext()) {
            dest = (Integer) i.next();
            mark(src);
            mark(dest);
            p.println(this.value(src) + "\t" +
                      this.value(dest) + "\t" +
                      this.cost(src,dest));
        }
    }

    public void write (PrintStream p) {
        for (int i = 0; i < this.size; i++) this.clear(i);
        for (int i = 0; i < this.size; i++) this.writeEdges(p, i);
        for (int i = 0; i < this.size; i++)
            if (!this.marked(i)) p.println(this.value(i));
        p.println();
    }
.
.
}
```

```
public abstract class AbstractGraph implements Graph {
.
.
.
    public void read () throws IOException { read(IO.stdin); }
    public void write () { write(IO.stdout); }
.
.
.
}
```