public class StackOfStrings		
	StackOfStrings()	*create an empty stack*
void	push(String item)	*insert a new string onto stack*
String	pop()	*remove and return the string most recently added*
boolean	isEmpty()	*is the stack empty?*
int	size()	*number of strings on the stack*

public class QueueOfStrings		
	QueueOfStrings()	*create an empty queue*
void	enqueue(String item)	*insert a new string onto queue*
String	dequeue()	*remove and return the string least recently added*
boolean	isEmpty()	*is the queue empty?*
int	size()	*number of strings on the queue*

```
class    DLLofString
```

```
    DoublyLinkedList()

void     insertFirst(String s)  inserts s at the head of the list
String   getFirst()          returns string at the head of the list
boolean  deleteFirst()       removes string at the head of the list
void     insertLast(String s)   inserts s at the end of the list
String   getLast(String s)       returns string at the end of the list
boolean  deleteLast()        removes string at the end of the list

void     insertBefore(int pos, String s)  inserts s before position pos
String   get(int pos)             returns string at position pos
boolean  deleteAt(int pos)             deletes string at position pos
```

```
public class MaxPQ<Key extends Comparable<Key>>
```

	`MaxPQ()`	*create an empty priority queue*
	`MaxPQ(Key[] a)`	*create a priority queue with given keys*
`void`	`insert(Key v)`	*insert a key into the priority queue*
`Key`	`delMax()`	*return and remove the largest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`Key`	`max()`	*return the largest key*
`int`	`size()`	*number of entries in the priority queue*

public class ST<Key, Value>		
	ST()	*create an empty symbol table*
void	put(Key key, Value val)	*put key-value pair into the table* ← a[key] = val;
Value	get(Key key)	*value paired with key* ← a[key]
boolean	contains(Key key)	*is there a value paired with key?*
void	delete(Key key)	*remove key (and its value) from table*
boolean	isEmpty()	*is the table empty?*
int	size()	*number of key-value pairs in the table*
Iterable<Key>	keys()	*all the keys in the table*

```
public class ST<Key extends Comparable<Key>, Value>
```

	...	
Key	min()	*smallest key*
Key	max()	*largest key*
Key	floor(Key key)	*largest key less than or equal to key*
Key	ceiling(Key key)	*smallest key greater than or equal to key*
int	rank(Key key)	*number of keys less than key*
Key	select(int k)	*key of rank k*
void	deleteMin()	*delete smallest key*
void	deleteMax()	*delete largest key*
int	size(Key lo, Key hi)	*number of keys between lo and hi*
Iterable<Key>	keys()	*all keys, in sorted order*
Iterable<Key>	keys(Key lo, Key hi)	*keys between lo and hi, in sorted order*

Graph API

public class Graph		
	Graph(int V)	*create an empty graph with V vertices*
	Graph(In in)	*create a graph from input stream*
void	addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer>	adj(int v)	*vertices adjacent to v*
int	V()	*number of vertices*
int	E()	*number of edges*

public class UF	
UF(int N)	*initialize union-find data structure with N singleton objects (0 to N – 1)*
void union(int p, int q)	*add connection between p and q*
int find(int p)	*component identifier for p (0 to N – 1)*
boolean connected(int p, int q)	*are p and q in the same component?*

public class Digraph		
	Digraph(int V)	*create an empty digraph with V vertices*
	Digraph(In in)	*create a digraph from input stream*
void	addEdge(int v, int w)	*add a directed edge v→w*
Iterable<Integer>	adj(int v)	*vertices pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

public class DirectedEdge		
	DirectedEdge(int v, int w, double weight)	*weighted edge v→w*
int	from()	*vertex v*
int	to()	*vertex w*
double	weight()	*weight of this edge*
String	toString()	*string representation*

public class EdgeWeightedDigraph		
	EdgeWeightedDigraph(int V)	*edge-weighted digraph with V vertices*
	EdgeWeightedDigraph(In in)	*edge-weighted digraph from input stream*
void	addEdge(DirectedEdge e)	*add weighted directed edge e*
Iterable<DirectedEdge>	adj(int v)	*edges pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Iterable<DirectedEdge>	edges()	*all edges*
String	toString()	*string representation*

public class SP		
	SP(EdgeWeightedDigraph G, int s)	*shortest paths from s in graph G*
double	distTo(int v)	*length of shortest path from s to v*
Iterable <DirectedEdge>	pathTo(int v)	*shortest path from s to v*
boolean	hasPathTo(int v)	*is there a path from s to v?*

public class FlowEdge		
	FlowEdge(int v, int w, double capacity)	*create a flow edge v→w*
int	from()	*vertex this edge points from*
int	to()	*vertex this edge points to*
int	other(int v)	*other endpoint*
double	capacity()	*capacity of this edge*
double	flow()	*flow in this edge*
double	residualCapacityTo(int v)	*residual capacity toward v*
void	addResidualFlowTo(int v, double delta)	*add delta flow toward v*

public class FlowNetwork		
	FlowNetwork(int V)	*create an empty flow network with V vertices*
	FlowNetwork(In in)	*construct flow network input stream*
void	addEdge(FlowEdge e)	*add flow edge e to this flow network*
Iterable<FlowEdge>	adj(int v)	*forward and backward edges incident to v*
Iterable<FlowEdge>	edges()	*all edges in this flow network*
int	V()	*number of vertices*
int	E()	*number of edges*
String	toString()	*string representation*