

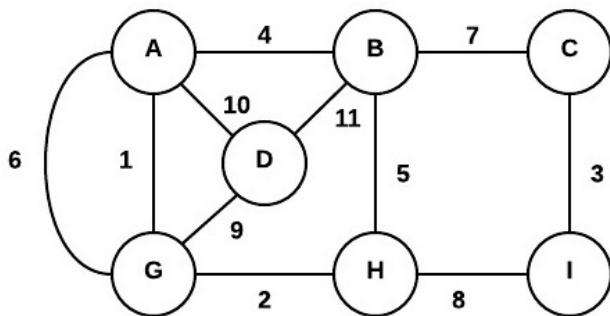
## Section Handout #8: More graphs, inheritance and polymorphism

Based on handouts by various current and past CS106B/X instructors and TAs.

Extra practice problems: CodeStepByStep – polymorphismMystery1-11, PancakeStack

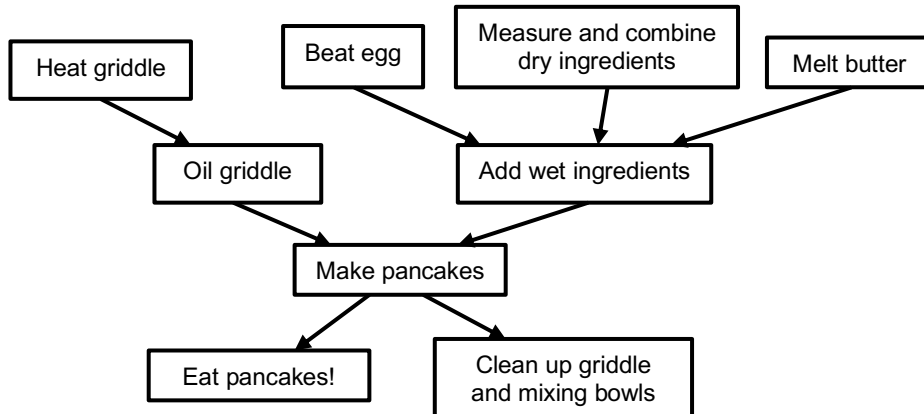
### 1. Kruskal's algorithm (*graphs*)

List the edges that *Kruskal's algorithm* would select to be part of a minimum spanning tree (MST) for the graph below. List them **in the same order** that Kruskal's would add them to the MST. What is the MST cost?



### 2. Topological sort (*graphs*)

Trace through a topological sort of the graph below. Is the ordering you found the only valid topological sort ordering of this graph?



Recall the pseudocode for Kahn's algorithm for topological sort:

```
function topologicalSort():
```

- map {each vertex  $\rightarrow$  its in-degree}.
- create a queue of all vertices with in-degree = 0.
- initially we have an empty topological sort ordering.

Until the queue is empty:

- dequeue the first vertex  $v$  from the queue.
- append  $v$  to the topological sort ordering.
- decrease the in-degree of all  $v$ 's neighbors by 1 in the map.
- enqueue any neighbors whose in-degree is now 0.

If all vertices are processed, success! Otherwise, there is a cycle.

### 3. Inheritance and polymorphism trace (*inheritance/polymorphism*)

Consider the following classes; assume that each is defined in its own file.

```
class Lettuce {
public:
    virtual void m1() {
        cout << "L 1" << endl;
        m2();
    }

    virtual void m2() {
        cout << "L 2" << endl;
    }
};

class Bacon : public Lettuce {
public:
    virtual void m1() {
        Lettuce::m1();
        cout << "B 1" << endl;
    }

    virtual void m3() {
        cout << "B 3" << endl;
    }
};

class Hamburger : public Bacon {
public:
    virtual void m2() {
        cout << "H 2" << endl;
        Bacon::m2();
    }

    virtual void m4() {
        cout << "H 4" << endl;
    }
};

class Mayo : public Hamburger {
public:
    virtual void m3() {
        cout << "M 3" << endl;
        m1();
    }

    virtual void m4() {
        cout << "M 4" << endl;
    }
};
```

Now let us assume that the following variables are defined:

```
Lettuce* var1 = new Bacon();
Bacon* var2 = new Mayo();
Lettuce* var3 = new Hamburger();
Bacon* var4 = new Hamburger();
Lettuce* var5 = new Lettuce();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the **line breaks with slashes** as in "x / y / z" to indicate three lines of output with "x" followed by "y" followed by "z".

If the statement does not compile, write "**compiler error**". If a statement would crash at runtime or cause unpredictable behavior, write "**crash**".

<u>Statement</u>	<u>Output</u>
var1->m1();	_____
var1->m2();	_____
var1->m3();	_____
var2->m1();	_____
var2->m2();	_____
var2->m3();	_____
var2->m4();	_____
var3->m1();	_____
var3->m2();	_____
var4->m2();	_____
var4->m3();	_____
var4->m4();	_____
((Bacon*) var1)->m1();	_____
((Bacon*) var1)->m3();	_____
((Mayo*) var5)->m3();	_____
((Lettuce*) var4)->m3();	_____
((Hamburger*) var2)->m4();	_____

## 4. Rigged Dice (*inheritance*)

In this problem, you will extend an existing class named **Dice** that represents a set of 6-sided dice that can be rolled by a player. See the table at the bottom of this page for the public functionality of the **Dice** class.

Write the .h and .cpp files for a new class called **RiggedDice** that extends **Dice** through inheritance. Your class represents dice that let a player "cheat" by ensuring that every die always rolls a value that is greater than or equal to a given minimum value. You should provide the same member functions as the **Dice** superclass, as well as the following new public behavior:

RiggedDice Member	Description
RiggedDice(int count, int min)	constructs a rigged dice roller to roll the given number of dice; all dice initially have the value 6 ( <b>hint:</b> similar to <b>Dice</b> constructor!) the given minimum value will be used for all future rolls, but throw an integer exception if the min value passed in is not between 1-6
virtual int getMin() const	returns the minimum roll value as passed to the constructor

**RiggedDice** should behave exactly like a **Dice** object except for the following differences. Note that you may need to override existing behavior in order to implement these changes.

- Every time a rigged die is rolled, ensure that the value rolled is greater than or equal to the minimum value passed to your constructor. Do this by re-rolling the die as long as the rolled value is too small.
- A **RiggedDice** object should return a total sum that lies and claims to be 1 higher than the actual total sum. For example, if the actual sum of the values on the dice is 13, your **RiggedDice** object's **total** method should return 14.
- When a **RiggedDice** object's **toString** is called or when the object is printed, it should display that the dice are rigged, then the dice values, then the minimum die value, in exactly the following format: "rigged {4, 3, 6, 5} min 2"

*Public functionality of the existing Dice class:*<sup>1</sup>

Dice Member	Description
Dice(int count)	constructs a dice roller to roll the given number of dice; all dice initially have the value of 6
~Dice()	frees all memory associated with a <b>Dice</b> object
virtual int getCount() const	returns the number of dice managed by this dice roller, as passed to the constructor
virtual int getValue(int index) const	returns the die value (1-6) at the given 0-based index
virtual void roll(int index)	rolls the given die to give it a new random value from 1-6
virtual int total() const	returns the sum of all current dice values in this dice roller
virtual string toString() const	returns string of dice values, e.g. "{4, 1, 6, 5}"
ostream& operator <<(ostream& out, Dice& dice)	prints the given <b>Dice</b> in its <b>toString</b> format (by calling its <b>toString()</b> method)

<sup>1</sup> The private instance variables and methods of the **Dice** class are not listed because private fields and methods cannot be accessed by subclasses, so they are not relevant to this problem.