Data Structures Review

See problem: 9

Vectors

See problems 1.1, 12
Linked List

See problem: 5

- Variations: singly linked, doubly linked.
- Implementations for stack, queue, deque.

Trees

See problems: 3, 13

- Variations: binary tree, binary search tree, quad tree, red-black tree.
- Implementation for maps, sets.

<table>
<thead>
<tr>
<th>traversal</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order</td>
<td>left, current, right</td>
</tr>
<tr>
<td>pre-order</td>
<td>left, right, current</td>
</tr>
<tr>
<td>post-order</td>
<td>current, left, right</td>
</tr>
</tbody>
</table>

Heaps

See problems: 4, 14

- Variations: min-heap, max-heap (priority queues), leftist heaps
- Can implement binary heap as array:

```c
size_t get_parent(size_t i) { return (i-1)/2; }
size_t get_left(size_t i) { return 2*i+1; }
size_t get_right(size_t i) { return 2*i+2; }
```

Hash tables

See problems: 8.4

- Variations: separate chaining, open addressing.
- Index found by using hash function. $m\_hash(obj) \% len$
Miscellaneous Topics

Note! misc problems for order notation: 10
Note!! misc memory-related problems: 11.1, 12, 16

Operator Overloading

See problems: 2, 11.4

class Complex {
public:
    Complex(double x=0, double y=0)
        : real_(x), imag_(y) {}
    Complex(Complex const& old)
        : real_(old.real_), imag_(old.imag_) {}
    Complex& operator= (Complex const& rhs);
    double Real() const;
    void SetReal(double x);
    double Imaginary() const;
    void SetImaginary(double y);
    double Magnitude() const;
    Complex operator+ (Complex const& rhs) const;
    Complex operator- () const; // (3)

friend istream& operator>> (istream& istr, Complex& c); // (1)
private:
    double real_, imag_; }

Complex operator- (Complex const& left, Complex const& right);
ostream& operator<<(ostream& ostr, Complex const& c); // (2)

Inheritance

See problem: 15

Relationships
<table>
<thead>
<tr>
<th>Relationship</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is-A</td>
<td>C1 is a C2</td>
</tr>
<tr>
<td>Has-A</td>
<td>C1 has a C2</td>
</tr>
<tr>
<td>As-A</td>
<td>C1 is implemented as a C2</td>
</tr>
</tbody>
</table>

Polymorphism

Animal animal = Bird();
animal.doAnimalNoise(); // does not work
Animal *animal = new Bird();
animal->doAnimalNoise(); // tweet

For the curious: “Upcasting”, “Object Slicing”

Diamond Problem

- “A pokemon belongs to two egg groups”
- “Both egg groups subclass from the pokemon class”
- “Each egg group should virtually inherit from the egg superclass”

Garbage Collection

See problems: 7, 8.1, 8.2

Reference Counting

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Fast and Incremental</td>
</tr>
<tr>
<td>- Can’t handle cyclical data structures!</td>
</tr>
<tr>
<td>? Requires 33% extra memory (1 integer per node)</td>
</tr>
</tbody>
</table>

Stop And Copy

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>- requires a long pause in program execution</td>
</tr>
<tr>
<td>+ can handle cyclical data structures!</td>
</tr>
<tr>
<td>- requires 100% extra memory (you can only use half)</td>
</tr>
<tr>
<td>+ runs fast if most of the memory is garbage (it only touches the nodes reachable from the root)</td>
</tr>
</tbody>
</table>
Feature

+ data is clustered together and memory is “de-fragmented”

Mark And Sweep

Feature

− requires long pause in program execution
+ can handle cyclical data structures!
+ requires 1% extra memory (just one bit per node)
− runs the same speed regardless of how much memory is garbage. It must touch all nodes in the mark phase, and must link together all garbage nodes into a free list.

Concurrency and Asynchronous Computing

See problems: 6, 8.3

Acceptable behaviors of concurrent programs

• No two operations that change any shared state variables may occur at the same time.
• The concurrent system should produce the same result as if the threads/processes had run sequentially in some order.

Mutexes, Condition Variables

• serialize non-atomic operations (std::map::operator[], std::priority_queue::pop()) behind mutex.
• condition variables mark end of concurrent operation; can move on.

Dining philosopher’s problem

• Deadlock occurs if we do not have global orderings of resources.
• i.e. the philosophers are all left handed, and will always wait on the chop stick to the left.