Final Exam Questions Study Guide (Draft!)

These questions are for review only and are not calibrated for time!

[**2-3 Tree**]: Given the following 2-3 tree declaration, implement the **`insert_right()`** function.

```
template<class T> class BTree {
public:
  class iterator;
  T* insert(T* data);
  iterator begin();
  iterator end();
  //...
private:
  struct Node {
    Node(T* data, Node* left=0, Node* middle=0);
    bool is_leaf() {return left_ == 0;}
    Node* insert_leaf(T* data, T**found);
    Node* insert_left(T* data, T**found);
    Node* insert_middle(T* data, T**found);
    Node* insert_right(T* data, T**found);
    Node* insert(T* data, T** found);
    //...
    T* data1_;
    T* data2_;
    Node* parent_;
    Node* left_;
    Node* middle_;
    Node* right_;
  };
};

template<typename T>
typename BTree<T>::Node*
BTree<T>::Node::insert(T* data, T** found) {
  if (is_leaf()) {
    return insert_leaf(data, found);
  } else if (*data < *data1_) {
    return insert_left(data, found);
  } else if (*data1_ < *data && (!data2_ || *data < *data2_)) {
    return insert_middle(data, found);
  } else if (data2_ && *data2_ < *data) {
    return insert_right(data, found);
  } else if (*data1_ < *data) {
    *found = data2_;
    return 0;
  } else {
    *found = data1_;
    return 0;
  }
}
```

[**2-3 Tree**]: Given the previous declaration, implement the **`begin()`** and **`end()`** functions and the **`iterator`** class with **`operator*()`**, **`operator++()`**, **`operator--()`**, **`operator+()`**, **`operator-()`** and **`operator==()`**. Do you also need to implement **`operator=()`**? Why or why not?

[Adjacency Matrix]: The adjacency matrix for an undirected graph is symmetrical about its diagonal. That is, (i, j) is an edge if and only if (j, i) is an edge. You decide to take advantage of this to save memory space by only holding the lower triangle, i <= j. The first row (row 0) will have one entry, the second two entries, etc. So a graph with N vertices will have N * (N+1) / 2 entries (sum from 1 to N).

Assuming the following declaration, implement the **`AdjacencyMatrix::edge()`** function (returns true if there is an edge between i and j).

```
class AdjacencyMatrix {
public:
  AdjacenceMatrix(int n)
    : n_vertices_(n), edges_(n*(n+1)/2, false) {}
  void add_edge(int i, int j);
  bool edge(int i, int j);
private:
  int n_vertices_;
  vector<bool> edges_;
};
```

[Adjacency Matrix]: Not realizing that **`std::vector<bool>`** is optimized for space, you decide to save even more space by implementing your own bitmap (http://en.wikipedia.org/wiki/Bitmap). Do it.

[**Graphs**]: Suppose a graph is given according to the following declaration:

```
class Graph {
  public:
    //...
    int color();
  private:
    struct Vertex {
      //...
      vector<Vertex*> successors_;
      int color_;
    };
  vector<Vertex*> vertices_;
};
```

We wish to assign a positive integer value, called *color*, to each vertex such that no two vertices sharing an edge have the same color value. Clearly, we could just enumerate vertices and assign N different colors but we'd really like to do this with the smallest number of distinct color values.

Since that is known to be an NP-complete problem, we will just use this heuristic: sort the vertices by degree (number of outgoing edges) and assign each vertex the lowest-number color not already taken by one of its neighbors. Implement the **`color()`** function using this heuristic. It should return the highest color number allocated. Add additional fields or methods as required. [since this is a study question, I *dare* you to make this an actual, running program].

[**Graph**]: Given the following graph declaration:

```
  public:
     //...
   private:
     struct Vertex {
       //...
       vector<Vertex*> successors_;
       int color_;
     };
   vector<Vertex*> vertices_;
};
```

Assume the graph is directed. Implement a function to determine whether the graph is a DAG (directed acyclic graph). Add fields and methods as required.

[**Graph**]: Given an undirected graph:

1. write a function that verifies every node has even degree
2. write a function that finds an Eulerian circuit (path from node back to itself that traverses every edge exactly once). See: http://en.wikipedia.org/wiki/Eulerian_path#Fleury.27s_algorithm

[implement it!]

[**Destructors**]: What is the output of the following program:

```
#include <iostream>
#include <string>

using namespace std;

class Base {
public:
  Base(): n_(count_++) {cout << "Creating a Base of operations " << n_ << endl;}
  virtual ~Base() {cout << "Base " << n_ << " destroyed\n";};
private:
  static int count_;
  int n_;
};

class FirstBase: public Base {
 public:
  FirstBase() {cout << "Creating a BaseBall object\n";}
  virtual ~FirstBase() {cout << "Can't get to first base\n";}
 private:
};

class StarBase: public Base {
 public:
  StarBase(const string& name): name_(name) {cout << "Creating StarBase " << name_
<< endl;}
  virtual ~StarBase() {cout << "StarBase " << name_ << " overrun by aliens.\n";}
 private:
  StarBase();
  string name_;
};

int Base::count_ = 0;

int main() {
  Base* alpha = new StarBase("alpha");
  StarBase* beta = new StarBase("Beta");
  StarBase* gamma = new StarBase("Gamma");
  StarBase delta("Delta");
  StarBase epsilon("Epsilon");
  Base* ballgame = new FirstBase();
  {
      StarBase omega = StarBase("Omega");
      delete ballgame;
  }
  Base* b;
  b = beta;
  delete b;
  delete gamma;
  return 0;
}
```

[**RAII/Destructors**]: Implement the **`std::shared_ptr`** template class
http://www.cplusplus.com/reference/memory/shared_ptr/

[**Virtual Functions**]: What is the output of the following program:

```
#include <iostream>

using namespace std;

class Base {
public:
  virtual void foo(int flag) {
      cout << "Base::foo()";
      if (flag) {
        cout << " ";
        bar(0);
      }
  }
  void bar(int flag) {
      cout << "Base::bar()";
      if (flag) {
        cout << " ";
        foo(0);
      }
  }
};

class Derived: public Base {
public:
  virtual void foo(int flag) {
      cout << "Derived::foo()";
      if (flag) {
        cout << " ";
        bar(0);
      }
  }
  void bar(int flag) {
      cout << "Derived::bar()";
      if (flag) {
        cout << " ";
        foo(0);
      }
  }
};

main() {
  Base* bp;
  Derived* dp;

  Base base_obj;
  Derived derived_obj;

  bp = &base_obj;
  cout << "Base pointer to Base object" << endl;
  bp->foo(1);
  cout << endl;
  bp->bar(1);
  cout << endl;
  cout << endl;

  bp = &derived_obj;
  cout << "Base pointer to Derived object" << endl;
  bp->foo(1);
  cout << endl;
  bp->bar(1);
  cout << endl;
  cout << endl;

  dp = &derived_obj;
  cout << "Derived pointer to Derived object" << endl;
  dp->foo(1);
  cout << endl;
  dp->bar(1);
  cout << endl;
}
```

[**Inheritance & Virtual Functions**]: The following program relies on knowledge of the implementation details of a particular C++ compiler and cannot be expected to work as intended in all cases (we've ventured into the home territory of the evil nasal demons), ***but*** it does work as intended using **g++** on a Linux system (and probably works under MS Visual Studio). Given what you know about C++ language implementation details, what is the output of the following program? *Why*?

Hint: *think* about how inheritance and virtual functions are implemented. Copy the code to a file and run it to verify your hypothesis.

```
#include <iostream>
#include <string>
using namespace std;

class Player {
public:
  Player(const string& name) : name_(name) {}
  virtual void print() const {cout << "Player " << name_ << endl;}
  const string& name() const {return name_;}
private:
  string name_;
};

class Elf : public Player {
public:
  Elf(const string& name, const string& weapon)
    : Player(name), weapon_(weapon) {
  }
  virtual void print() const {
    cout << "Elf " << name() << " wielding " << weapon_ << endl;
  }
private:
   string weapon_;
};

class Hobbit : public Player {
public:
  Hobbit(const string& name, const string& food)
    : Player(name), food_(food) {
  }
  virtual void print() const {
    cout << "Hobbit " << name() << " eating " << food_ << endl;
  }
private:
  string food_;
};

void do_print(const Player* player) {player->print();}

int main() {
  Elf legolas("Legolas", "Bow and Arrow");
  Hobbit samwise("Samwise Gamgee", "lembas wafer");
  legolas.print(); samwise.print(); cout << endl;
  void** p1 = reinterpret_cast<void**>(&legolas);
  void** p2 = reinterpret_cast<void**>(&samwise);
  void* p3 = *p1; *p1 = *p2; *p2 = p3;
  do_print(&legolas); do_print(&samwise); cout << endl;
  return 0;
}
```

[**Finite-State Machine**]: Define the states and events of a vending machine. Write out a state transition table (row for each state, column for each event).

[**Finite-State Machine**]: Implement the vending machine FSM defined above using a lookup table (`std::map`) to determine the next state.

[**Regular Expression/Finite-State Automaton**]: Write a regular expression to match any word with

1. at least 4 'a's. (e.g. Guadalajara, abracadabra, paraphernalia)
2. exactly 4 'a's. (e.g. amalgamation, baccalaureate, catamaran).

Write an equivalent finite-state automaton. Test the expression using grep, sed, or Perl, or any other R.E.-breathing utility using data from the Linux system file **`/usr/share/dict/words`**. See also: https://xkcd.com/208/

[**Regular Expression**]: construct a regular expression over the symbols {0, 1} to match strings with odd parity (odd number of 1-bits).

[**Context-Free Grammars**]: Construct a CFG grammar over the terminal symbols {a, b} such that it contains at least as many a's as b's.

[**Context-Free Grammars**]: Construct a CFG grammar over the terminal symbols {a, b, c} that match $a^n b^m c^{m+n}$ for n > 0 and m>= 0. Try doing this without any epsilon-productions.

[**Context-Free Grammars**]: Given the following grammar (matched parentheses):

```
S -> (A)A
A -> ε
A -> (A)A
```

Show a derivation/parse tree for the string `((())))()(()))`.

[**Context-Free Grammars**]: construct a grammar to recognize matched parentheses that does not use ε-productions.