ECE 590.01 PRACTICE Final Exam

This is a full length practice midterm exam. If you want to take it at exam pace, give yourself 180 minutes to take the entire test, but you should be able to finish in 120 minutes. Just like the real exam, each question has a point value. There are 120 points in the exam, so that you can pace yourself to average 1 point per minute (some parts will be faster, some slower), and have 1 hour to spare.

Questions:

1. Multiple Choice: 10 points
2. Concurrency: 10 points
3. Data Structure Concepts: 15 points
4. OO Implementation: 15 points
5. Coding 1: 20 points
6. Coding 2: 20 points
7. Coding 3: 30 points

This is the solution set to the practice exam. The solutions appear in blue.
Question 1: Multiple Choice [10 pts]

1. Which data structure is most efficient to use for implementing a priority queue?
   (a) Linked List
   (b) AVL Tree
   (c) Red/Black Tree
   (d) Heap
   (e) Hash Table

2. What principle is key to building large pieces of software?
   (a) Locality
   (b) Reciprocation
   (c) Abstraction
   (d) Redundancy
   (e) None of the Above

3. Which of the following accurately describes abstraction?
   (a) Pointers to sub-classes can be treated as pointers to their super-class.
   (b) Separation of interface from implementation.
   (c) Method calls are always dispatched to the implementation in the dynamic type of the object.
   (d) Data is often re-referenced again soon after it has been referenced.
   (e) None of the Above
4. What is the main disadvantage of heap sort?
   
   (a) Its worst case running time is $O(N^2)$
   
   (b) Its average case running time is $O(N^2)$
   
   (c) It requires allocating extra space for temporary arrays
   
   (d) It requires very complex operations at each step which take a long time.
   
   (e) None of the Above $\leftarrow$ Poor spatial locality

5. Which algorithmic category best describes Dijkstra’s shortest path algorithm?
   
   (a) Dynamic Programming
   
   (b) Brute Force
   
   (c) Genetic
   
   (d) Greedy $\leftarrow$
   
   (e) None of the Above
Question 2: Concurrency [10 pts]

1. Briefly explain the concept of a “critical section.”

   **Answer:**
   When multiple threads need to access the same data, they generally cannot do so in parallel safely. A critical section is a piece of code where execution must be serialized to races on shared data.

2. Briefly explain why a critical section cannot be protected by simply doing something like this:

   ```
   while(locked) { ; }
   locked = 1;
   //critical section
   locked = 0;
   ```

   **Answer:**
   Testing and setting the lock is not atomic—there is still a race between checking the value of locked and setting it to 1. Two threads can both observe a value of 0 for locked at the same time, and enter the critical section concurrently.

3. Give an example of one of the primitives that can be used to build a lock which correctly protects a critical section.

   **Answer:**
   Atomic compare-and-swap.
Question 3: Data Structure Concepts [15 pts]
Show the results of performing each of the following operations on the shown data structures:

1. Add 12 to the following min-heap

```
2
12 9
16 32 14 16
26 40
```

2. Add 43 to the following (regular, un-balanced) BST

```
14
10 99
8 34
2 43
```
3. Remove 14 from the BST in the previous part (before you added 43).

4. Add 38 to the following AVL Tree
5. Add 45 to the following Red/Black Tree (the coloring of each node is shown with a B or an R)
Question 4: OO Implementation [15 pts]
For parts 1–5 use the following class declarations:

class A {
    int a;
    virtual void foo();
};
class B : public A {
    int b;
    virtual void bar();
}
class C: public A {
    int c;
    virtual void foo();
};
class D: public B, public C{
    int d;
    virtual void xyz();
}

1. Draw the layout of objects of type A

2. Draw the layout of objects of type B

3. Draw the layout of objects of type D
4. Suppose you wanted objects of type D to have only one A instead of two. Show how you would change the above declarations to do this.

**Answer:**
Make both B and C inherit A virtually.

```cpp
class B : public virtual A
class C : public virtual A
```

5. Draw the layout of objects of type D with the change you made in part 4.
6. Name and *briefly* explain a **pragmatic consideration** of Mixins.

**Answer:**
Any of the items described in Section 3 of the Mixin Paper.
Question 5: Coding 1 [20 pts]
Fill in the remove method in the IntBST class shown below (which is a binary search tree holding ints—it is just a plain BST, without balancing). You may write any helper functions you need on the next page:

class IntBST {
    class Node {
        public:
            int key;
            Node * left;
            Node * right;
    };
    Node * root;
public:
    void remove(int k) {

        root = remove(root,k);
    }
};
(helper methods for Question 5)

Node * twoChildHelper(Node * curr, Node * replace) {
    if(curr->right != NULL) {
        curr->right = twoChildHelper(curr->right, replace);
    }
    else {
        replace->key = curr->key;
        Node * temp = curr->left;
        delete curr;
        return temp;
    }
}

Node * remove(Node * curr, int k) {
    if (curr == NULL) return NULL;
    if (curr->key == k) {
        Node * temp = curr;
        if ( curr->left == NULL) {
            temp = curr->right;
            delete curr;
        }
        else if (curr->right == NULL) {
            temp = curr->left;
            delete curr;
        }
        else {
            curr->left = twoChildHelper(curr->left, curr);
        }
        return temp;
    }
    else if (k < curr->key) {
        curr->left = remove ( curr->left, k );
    }
    else {
        curr->right = remove ( curr->right, k );
    }
    return curr;
}
Question 6: Coding 2 [20 pts]
Suppose you have already written a templated Set class, with the following interface:

```cpp
template<class T>
class Set {
public:
    Set();
    void add(const T& item);
    bool contains(const T& item) const;
    void remove(const T& item);
    class iterator {
        iterator & operator++();
        T& operator*();
        bool operator==(const iterator & rhs);
        bool operator!=(const iterator & rhs);
    };
    iterator begin() const;
    iterator end() const;
};
```

and you also have the following abstract Function class:

```cpp
template<class R, class A>
class Function {
public:
    virtual R invoke(A arg) =0;
};
```

Write a function which filters a Set, creating a new Set which is a subset of the first set, containing only those items for which the Function (f) it is passed returns true:

(answer on the next page)
template<class T>
Set<T> * filterSet(Set<T> * inSet, Function<bool, const T&E28;> * f) {

    Set<T> * ans = new Set<T>();
    Set<T>::iterator it = inSet->begin();
    while (it != inSet->end()) {
        T & x = *it;
        if (f->invoke(x)) {
            ans->add(x);
        }
        ++it;
    }
    return ans;
}
Question 7: Coding 3 [30 pts]
A $k$-clique in a graph is a set of $k$ nodes such that each of the $k$ nodes in the clique is adjacent to every other node in the clique (it may or may not be adjacent to other nodes outside the clique too). Assuming that you have the Set described in the previous question (as well as the filterSet function you wrote), and the following Graph implementation:

```cpp
class Node {
public:
    Set<Node *> * getAdjacencies();
    bool isAdjacentTo(const Node & n);
};

class Graph {
public:
    Set<Node *> * getAllNodes();
};
```

Write the function hasKClqie which determines if the graph has a $k$-clique. Note that the clique problem is NP-complete, so your algorithm does not need to be efficient, it just needs to work. You may write any helper methods or classes that you want. You may assume that no node is adjacent to itself in the graph for simplicity.

(answer on the next page)
bool hasKClue(Graph * g, int k) {
    return hasKClue (g->allNodes(), k);
}

class NodeFilter : public Function<bool, const Node *&> {
    Set<Node *> * set;
public:
    NodeFilter (Set<Node*>* s) : set(s) {}
    bool invoke (const Node *& n) {
        return set->contains(n);
    }
}

bool hasKClue(Set<Node *> * nodes, int k) {
    if (k <= 0) {
        return true;
    }
    Set<Node*>::iterator it = nodes->begin();
    while (it != nodes->end()) {
        Node * n = (*it);
        Set<Node *> * adj = n->getAdjacencies();
        Function<bool, const Node *&> * filter = new NodeFilter(nodes);
        Set<Node *> * x = filterSet (adj, filter);
        if (hasKClue (x, k-1)) {
            delete x;
            return true;
        }
        delete x;
        ++it;
    }
    return false;
}