Quiz 3 Study Guide

Below are review questions for you to study for the upcoming final. Only the material covered since the last quiz is covered in this study guide. There has been three topics since the last quiz: Graph Definitions and Implementations; Graph Algorithms; and C++ References and Curiously Recurring Templates.

The final will consist 6-8 questions of the following variety: (1) Comprehension and Recall, where you will be asked to recall and answer knowledge questions; (2) C++ Programming, where you will be asked to program, in syntax correct C++ code, solutions to simple programming problems; and (3) Advancements of Knowledge, where you will be asked to advance your knowledge of a topic by applying that knowledge to a similar but related problem, for example, consider implementing a data structure not covered directly in class.

You will have 3 hours to complete the exam, but the exam as written should take you no more than 1.5 hours. Feel free to use the entire time to complete the exam.

1 Graph Definitions and Implementations

- What kinds of systems, like a social network, do graphs represent well?
- Write down the mathematical constructions of graphs for vertices and edges. Show all the extensions for directed, undirected, labelled, unlabelled, weighted and unweighted.
- Using your lab as a reference, explain how each of the properties of graphs are leverages: directed, undirected, labelled, unlabelled, weighted and unweighted.
- What is the degree of a vertex? Whats i the in-degree? What is the out-degree?
- What is the difference between connected, strongly connected, and completely connected graphs?
- What are the two main properties of simple graphs?
- In a simple graph, show that the maximal number of edges is $O(|V|^2)$, where $|V|$ is the number of vertices.
- Assume that each vertex has a probability of $1/|V|$ of being connected to any other vertex. What is the expected average degree of all vertices in the graph? What if the probability of two vertices being connected by an edge is $k/|V|$ where $k < |V|$?
- If a graph is simple and complete, what is the sum across the degree's of all the vertices?
- Consider implementing a graph using an *adjacency matrix*: What are the advantages, and what are the disadvantages? For what kinds of graphs are an adjacency matrix an appropriate implementation type.

- Consider implementing a graph using an *adjacency list*: What are the advantages and what are the disadvantages? For what kinds of graphs are an adjacency list an appropriate implementation type.
- If you had a directed graph implementation, could you use that implementation to encode an undirected graph? Explain how.
- If you had an undirected graph implementation, could you use that implementation to encode a directed graph? Explain how.

2 Graph Algorithms

- Explain how breadth first search of a maze relates to finding the shortest path of an unweighted graph.
- Draw a random unweighted graph with 7 vertices, execute Breadth First Search on it.
- Explain the bookkeeping table in BFS? How does a particular path extracted from the table information.
- For the Breadth First Search routine from class, what is the complexity?
- Explain why Breadth First search does not work on a graph with weighted edges.
- What aspect of Dijkstra's Algorithm is *greedy*? What makes an algorithm greedy?
- Apply random edge weights to the random graph above, perform Dijkstra's on the graph.
- In psuedo-code, write out the Dijkstra's algorithm.
- What is the complexity of Dijkstra's algorithm? What data structures are needed to implement Dijkstra's? Which data structure/procedure dominates the run time of Dijkstra's?
- Can you use Dijkstra's algorithm on a unweighted graph? Can you argue that it reduces to Breadth First Search?
- Why does Dijkstra's fail on graph with negative edges? Can you provide a sample graph demonstrating this failure?
- Bellman-Ford can find shortest paths in a graph with negative edges: What is the runtime? How does this runtime compare to BFS and Dijkstra's in terms of graph complexity.
- Define a Minimum Spanning Tree (MST). What makes it *minimum*, *spanning* and a *tree*?
- Consider a MST: Argue why adding an edge to the tree must generate a cycle.
- For the weighted graph you wrote out above, extract the minimum spanning tree using Kruskal's Algorithm and Prim's Algorithm.
- Explain, in general terms, the difference between Kruskal's and Prim's algorithm. Write pseudo code for both. You can assume all requisite data structures, such as a priority queue and a disjoint set.
- Explain what a Disjoint-Set is, and how it relates to solving Kruskal's aglorithm. Why can't you just use the same bookkeeping table as before?

- Argue that the runtime analysis for Prim's is the same as Dijsktra's.
- What is the runtime of Kruskal's? Why does Kruskal's genearlly run much faster than its worst case analysis?

3 Advanced C++

- Draw the stack diagram at the marked points in the program:

```
int foo(int x, int * p){
  x = x+1;
  *p = x + (*p)++;
  p = &x;
  //POINT B
  return x;
}

int main(){
  int a,b, *c;
  a = 1;
  c = &a;
  // POINT A

  b = foo(a,c);

  // POINT C
  a = foo(a,&b);

  // POINT D
  return 0;
}
```

- In the expression:

```
a = foo(a, &b)
```

  How many copies occurred? Enumerate them.

- draw the stack diagram for the following code:

```
int foo(int x, int  &y, int *z){
  x += y;
  z = &y;
  y = *z*2
}

int main(){
  int a,b, *c;
  a = 1;
  b = 2;
  c = new int(3);
  // POINT A
```

```
  b = foo(*c,a,&b);
  // POINT C

  *c = foo(a,b,c);
    // POINT D
  return 0;
}
```

- What is the difference between a reference and a pointer? What is a property of pointers that references cannot do?
- Consider implementing a *shallow* copy constructor for a basic ArrayList. Fill in the code below:

```
template <typename T>
ArrayList(const ArrayList<T>& rhs){









}
```

- Consider implementing a *deep* copy constructor for a basic ArrayList. Fill in the code below:

```
template <typename T>
ArrayList(const ArrayList<T>& rhs){









}
```

- Consider implementing a copy assignment operator for a basic ArrayList that uses *shallow* copies. Fill in the code below:

```
template <typename T>
ArrayList<T>& operator=(const ArrayList<T>& rhs){









}
```

- Consider implementing a copy assignment operator for a basic ArrayList that uses *deep* copies. Fill in the code below:

```
template <typename T>
ArrayList<T>& operator=(const ArrayList<T>& rhs){









}
```

- In the assignment operator, why might you want a shallow copy over a deep copy?
- What is the rule of 3 for C++? Justify it.
- Draw the inheritance graph the curiously recurring template pattern below (`Circle` is a subclass of `Shape`:

```
template <class Tbase, class Tder>
class Cloneable: public Tbase {
public:

  virtual Tbase* clone() const {
    return new Tder(static_cast<const Tder&>(*this));
  }

};

class Circle: public Cloneable<Shape, Circle> {
public:

  int radius;

  Circle(int r): radius(r) {}

  virtual ~Circle() {}

  virtual void draw() const {
    std::cout << "() <- this is a circle of radius " << radius << "\n";
  }

};
```

- Why do I think the `Cloneable` class is “dirty”? And why does Matt Zucker think otherwise?