# Final Review

In developing pseudocode algorithms for the exam, your algorithm must have one line of pseudocode per line of code (i.e. must be very specific and as close to code as possible). For algorithms, while writing actual code is acceptable, you will NOT lose points for compiler errors. However, when an implementation is requested, you are expected to write working code and will lose points for errors.

## Templates

* Use the vector class to build a collection
  + Use the [] and at() to access elements, why use at()?
    - **Vector<T> numbers(10);**
    - **numbers[2] = third element in vector;**
    - **numbers.at(2) = third element in vector;**
  + Use push\_back to add elements to the vector
    - **numbers.push\_back(3); = adds the number 3 to the vector**
  + Use other vector methods:
    - Size, resize
      * **Numbers.size() = number of elemts in vector**
      * **Numbers.resize(5) = resizes vector to hold 5 elements**
    - Capacity, reserve
      * **Numbers.capacity() = size of storage space allocated for vector in terms of elements**
      * **Numbers.reserve(20) = request vector capacity be changed to 20**
    - Erase, clear
      * **Numbers.erase(numbers.begin()) = erases first element in vector**
      * **Numbers.clear() = removes all elements from the vector**
    - Empty
      * **Numbers.empty() = bool statement that checks if vector is empty**
* Create a template function (like swap, max, min, etc.) to operate on any type
  + **Template <class T>**
  + **Vector<T> foo ();**
  + **Vector<T> bar ();**
  + **Foo.swap(bar);**
    - **Now foo holds the templated elements of bar, and bar holds the templated elements of foo.**
* Pass a template object as a parameter
  + DoublyLinkedList( const DoublyLinkedList<T>& list);
* Return a template object
  + **T DoublyLinkedList<T> :: RemoveFront()**
    - **T data; …..**
    - **Return data;**
* Understand that compiler doesn’t perform full type-check until instantiation

# Template Classes

* What are common uses of templates in classes?
  + **Linked list of trophies**
  + **Queue of customers**
  + **Stack of cards**
  + **Vector of grades**
  + **Binary tree of inventory items**
  + **Hash table of players**
* How do we have to modify a traditional class to incorporate templates?
  + **Add template statement in front of class declaration**
  + **Add template statement in front of each method implementation**
  + **Convert internal types, as appropriate, to template parameter**
  + **#include the .cpp and guard the .cpp file**
* How can you use 2 types in a template?
  + **By using a pair**
  + **Template <class Key, class Data>**
  + **Pair< string, Employee> worker (“mark”, mark);**
* How can you use template values?
  + **To hide the size of an array, return size of vector/array**
  + **SmartArray< T, size>;**
  + **SmartArray< int, 10>;**
* How can you use operators with templates as friends of the template class?
  + **By adding empty <>**
  + **Ostream& operator << <> (ostream& sout, const SmartArray<T,size>& the Array);**
* When must we forward declare the class and operators?
  + **Before we compile our code**

# Recursion

* What is recursion?
  + **A function that uses itself to solve a subproblem**
* What are the 3 essential elements of a properly structured recursive function? What role do each of these elements play in a recursive function?
  + **Base case**
  + **Recursive call**
  + **Progression toward base case**
* Can any iterative solution be converted to a recursive solution or vice-versa?
  + **Yes, but chose the correct one based on which is more time/space efficient**
* Predict the output of a recursive function.
* Determine if iterative and recursive solutions are equivalent.
* Build a recursive function.
  + **Print 10 to 1**
  + **printReverse(10);**
  + **public void printReverse(int i)**
    - **cout << i << endl;**
    - **if(I > 1)**
      * **printReverse( i – 1);**
* What are the strengths and weaknesses of recursion? When would you choose to use it over an iterative solution and vice versa?
  + **Is good when it reduces intellectual overhead, solution can be expressed as a combination of subproblems, and tree data structures**
  + **Is bad when iterative solution is more time/space efficient, problem is “deep” and would have a lot of recursive steps, large data sets**

# Trees

* Define a binary tree. Discuss the basic structure of a tree – how are the nodes connected?
  + **A tree is defined recursively w/ a hierarchical collection of nodes.**
  + **Root is at the top**
  + **Nodes are it’s children which contain data**
  + **Nodes are connected by an edge**
* Discuss the algorithmic complexity of a tree.
  + **O(logN)**
* Discuss the differences, benefits, and drawbacks to using trees versus lists, stacks, or queues.
  + **Better for search, add, remove, and storing sorted things hierarchically**
* Discuss the kinds of problems that trees are likely to solve.
* Develop the insert and search methods for a binary tree.





* Develop the findMin and findMax methods for a binary tree.
  + **FindMin(Node n)**
    - **If (n has left child)**
      * **Recurse left**
    - **If (n does not have a left child)**
      * **Return value of n**
  + **FindMax(Node n)**
    - **If (n has a right child)**
      * **Recurse right**
    - **If (n does not have a right child)**
      * **Return value of n**
* Discuss the three remove scenarios from a binary tree. Draw a picture to illustrate each scenario.
  + **If node has no children**
  + **If node has 1 child**
  + **If node has 2 children**
* Develop a method that removes a node from the tree.



* [from AVL lesson] Develop a method to perform a preorder, inorder, and postorder print of a binary tree.
* [from AVL lesson] Given a provided tree, perform a preorder, inorder, or postorder print of the nodes.

# AVL Trees

* Discuss the reasons behind needing to “balance” a binary tree. Be sure to discuss the algorithmic complexity in your answer. Draw a picture to illustrate your point.
  + **Enforces that no search will be worse than O(logN + 1)**
* Describe in detail the AVL method for balancing a tree.
* Discuss the 4 cases of insertion that require rebalancing. Draw a picture to illustrate each rebalancing solution.
  + **Left subtree of left child**
  + **Right subtree of left child**
  + **Left subtree of right child**
  + **Right subtree of right child**
* Given a tree, insert one or more provided values, redraw the tree after rebalancing.
* Develop a method that inserts into an AVL tree, rebalancing if necessary.

# Priority Queues

* Discuss the differences between a priority queue and a queue.
  + **Queue is first in first out, while priority queue is associated with data in item**
* Discuss the algorithmic complexity of a priority queue.
* Develop a class to represent a prioritized item. Implement the CompareTo method on this prioritized item.
* Develop the enqueue, dequeue, peek, and count methods for a priority queue.

# Heaps

* Discuss how heaps are used to implement a priority queue
* Discuss how a heap uses a single rule to maintain “partial order”
* Demonstrate how heaps use arrays to implement a priority queue
* Develop the Insert, DeleteMin, and Remove methods for a heap
  + Include the “percolate up” and “percolate down” concepts

# Graphs

* Discuss the components of a graph including vertices, edges, paths, loops, and cycles
* Discuss and identify attributes including undirected, directed, connected, complete, acyclic, strongly connected, weakly connected graphs, and weighted.
* Given an example graph, show the steps of a breadth-first search
* Develop a breadth-first **algorithm** for an unweighted graph
* Develop Dijkstra’s algorithm to determine the shortest path for a weighted graph
* Given an example graph, show the steps of Dijkstra’s algorithm.

# Sets

* What are the characteristics of a set?
  + **Collection of values**
  + **Sorted, unique values, internally stored as a balanced tree, uses operator < to sort items, must use iterator to step through set, cant modify, only insert/erase**
* What is the underlying data structure that is used to implement a set?
* What is the algorithmic complexity of a set’s insertion, search, and removal?
* Why can’t you modify an item in a set?
* Use the set methods to modify a set
  + **Insert**
  + **Erase**
* Use iterators to iterate through the set
  + **Begin**
  + **End**
  + **++, --**
* Use find() to find an item in the set, what does this method return?
  + **An iterator to the item**
* How is a multiset different from a set?
  + **They can allow duplicates**
  + **Duplicates are sorted next to each other**
  + **Same library as set**

# Hash Tables

* Describe the fundamental structure of a Hash Table
* Discuss the algorithmic complexity of each Hash Table method
* Discuss the importance, role, and use of as Hash Function
* Discuss what makes a “good” hash function, provide examples.
* Describe the characteristics of data that will work well as a hash table key.
* Discuss how integers (like a social security number or student id) and strings (like an email address or a username) can be hashed well.
* Describe the “load” of a hashtable, why is it good to keep this under ½ full.
* Why is it good to use a prime number for the size of the hashtable?
* Describe when we increase the size of a hashtable and how we accomplish this.
* Describe the 4 most common techniques for handling collisions in a hashtable
  + Separate chaining
  + Linear probing
  + Quadratic probing
  + Rehashing
* Predict the sequence of buckets tried with each of these techniques
* What are the strengths and weaknesses of each technique? What type of data would produce the worst case scenario in each?