Tips from Varun about Job interviews:

1. Start off with the most naïve and simple solution for a problem.

2: Data Structures: Store data and do operations on the data.

* Arrays (1D and 2D arrays) – See if these arrays can be fit anywhere in the solution, especially if the domain is limited to a certain size

Go to the next data structures and implement them in your solution when you are trying to optimize your solution. Reiterate the data structure that I’m trying to use for this problem and convince myself that I can reach to an answer that can be solved using this data structure.

* Linked Lists: If the size of the elements is unknown, store them in a dynamic collection of elements in a linked list
  + Circular Linked List
  + Doubly Linked List
* Stacks: LIFO kind of optimization
* Queue: FIFO kind of optimization
* Trees (Binary Search Tree): Requiring the most efficient insertions, deletions, and searches in O(log(n)) time.
* Hash Tables: Tables held with values pointed to by keys. These keys map to a certain index of the table using the Hash Function, H(x), which is crucial in this implementation. Two keys can point to the same index of a table and can be resolve by maintaining a linked list of values. This is advanced data structure.

3: Search Algorithms: Start off with a basic Search algorithm and then optimize it based on the requirement.

* Linear Search
* Binary Search if the array is Sorted

4: Sorting Algorithms: Start off with a basic Sorting algorithm and then optimize it based on the requirement.

* Selection Sort
* Bubble Sort
* Insertion Sort
* Merge Sort, Quick Sort [Divide and Conquer]

5: Write some code in C++ to practice and compile it to make sure that the code works. This will give me confidence that I can write bug free code that executes correctly to come to a final answer.

10/14 Plan:

* ~~Complete the MyString program~~
* Go over Hash Tables, Heaps, Templates
* Do cracking the code object oriented programming questions
* Do data structures interview questions from the Career Cup. Think about the problems and come up with the most naïve solution first. Then think about optimizing it. As naïve the solution can be, come up with different ways to solve it in the easiest way possible.
* Write another program for Linked List and probably for trees as well.

There are no static constructors in C++. There are private constructors that prohibit the instantiation of a class by a user. In a singleton pattern, one can make the default constructor private and then make a static variable that’s the instance of the class and can instantiated using a Static Member function. Static Member functions only operate on Static Variables as they do not have a “this” pointer.

UML Relationships:

1: Association:

* When a class “uses” the instance of another class
* No parent-child relationship
* Lifetime of the objects is independent of each other
* Example: A Manager instance uses a SwipeCard instance using a public member function that receives a reference to instance of the SwipeCard class.

bool Manager::EnterTheBuilding (SwipeCard& obj)

2: Aggregation:

* When a class “contains” the instances of another class
* Parent-child relationship
* Lifetime of the objects is independent of each other
* Example: A Classroom instance can have an array of Student instances. However, Student instances can exist outside the Classroom instance.

class Classroom

{

int num\_Students;

int max\_Students;

Student \*m\_Students; // a pointer to an array of Students

public:

Classroom (int maxStudents)

{

max\_Students = maxStudents;

if (max\_Students!=0)

{

m\_Students = new Student [max\_Students];

}

else

{

m\_Students = 0;

}

num\_Students = 0;

}

void AddStudent (Student& obj)

{

if (num\_Students<=max\_Students)

{

m\_Students[num\_Students] = obj;

}

else

{

cerr << "You cannot add more students than " << max\_Students;

}

}

}

3: Composition

* When a class instance is responsible for instantiating an instance of another class
* Parent-Child relationship
* Death Relationship. The lifetime of the child instance is dependent on the lifetime of the parent’s instance
* Example: A class car that cannot exist without other child instances like Gear, Seats, Steering, etc. When the car instance goes away, the Gear, Seats, Steering, etc instances should be destroyed with that.

class Car

{

Steering steer;

Wheel wheel;

Seat seat;

}

When the instance of car is created, the instances of Steering, Wheel, and Seat are also created. Additionally, when the instance of car goes out of scope, the instance of Steering, Wheel, and Seat also go out of scope. Obviously, if there are any pointers involved, we need our own destructors, copy constructors, and assignment operators.