Fundies 2 Cheat Sheet

- **Remember to include purpose statements**
- **Remember to include sufficient tests**
- **Remember to write EFFECT statements whenever doing mutation**

Visitor Pattern

```
interface I[Class]Visitor<T> {
      T apply ([Class] var);
      T visit[concrete1] ([concrete1] var);
      T visit[concrete2] ([concrete2] var);
}

interface Class {
      <T> T accepts(I[Class]Visitor vis);
}

class concrete1 implements Class {
      <T> T accepts(I[Class]Visitor vis) {
            vis.visit[concrete1](this);
      }
}
```

Iterable and Iterator

- A class that is `Iterable<T>` must implement the `iterator()` method, which must return an iterator on elements of type T.
  - **For-each loops** can only be used on classes that implement Iterable<T>.
- A class that is an `Iterator<T>` must implement `boolean hasNext()` and `T next()` .
  - `boolean hasNext()` should never change or mutate the data in anyway. It should always return the same answer no matter how many times it is called.
  - `T next()` should first call `boolean hasNext()` in order to prevent some lower level exception and instead return a custom one for the particular iterator.

Hashing and Equality

- Whenever overwriting `boolean equals(Object obj)`, you must also overwrite `int hashcode()` to maintain the following invariants:
  - If two objects are equal according to the `equals` method, then calling the `hashcode` method on each of the two objects must produce the same integer result.
  - If two objects are unequal according to the `equals` method, then calling the `hashcode` method on each of the two objects ***may or may not*** produce the same integer result.
    - This is due to Hash collisions, which is guaranteed by the **Pigeonhole Principle**.
  - An easy way to get a "good" hashcode for something is using `Objects.hashcode(Object o).`
- When overwriting `boolean equals(Object obj)` for a particular class, you must first check if the given `obj` is an `instanceof` that class, and then cast it to that type if it is, or return `false` otherwise.

Priority Queues and Heaps

- **Heap Invariant**:
  - Every value in the both the left and right subtrees of a node must be less than or equal to the value at the node, and
  - Both the left and right subtrees must themselves be valid heaps.
- A **Priority-Queue** satisfies the logical Heap Invariant, and the structural invariant of fullness. Thus, a priority queue is typically implemented with a heap, with some additional logic to maintain the structural invariant.
  - When adding items to the heap, we add them to the very bottom/end of the heap, and then we upheap them.
    - Upheaping is the process of recursively swapping an item with its parent if the item has a greater value than its parent, until it is no longer greater than its parent.
  - When removing an items from the heap, we remove from the very top (getting the item with the highest priority) and swap to the top of the heap the last item and downheap it.

        - Downheaping is the process of recursively swapping an item with the child with the greatest value, until the item no longer has any larger children.
    - Some implementations use an Arraylist to represent the heap, such that for any index/node:
        - The left child can be found at $leftChild(idx) = (2 \times idx) + 1$
        - The right child can be found at $rightChild(idx) = (2 \times idx) + 2$
        - The parent can be found at $parent(idx) = floor(\frac{idx-1}{2})$
- Heapsort depends on a priority-queue and takes advantage of the `removeMax` procedure (removing the first element). Each time after removing the first element, the priority-queue adjusts itself so that the first element has the highest priority, so if we keep inserting the new max element to the front of the list, eventually we produce a list of sorted elements.

Trees

- Depth-First Search
    - Iterates through the tree using a Stack (which can be implemented using Deques)
- Breadth-First Search
    - Iterates through the tree using a Queue (which can be implemented using Deques)

Graphs

- Possible Implementation

```
class Vertex {
  ... any data about vertices, such as people's names, or place's GPS coordinates ...
  IList<Edge> outEdges; // edges from this node
}
class Edge {
  Vertex from;
  Vertex to;
  int weight;
}
class Graph {
  IList<Vertex> allVertices;
}
```

- Prim's Algorithm
    - A greedy algorithm (making a locally optimal choice for finding a global optimum) for building a minimum spanning tree. Starts at an arbitrary node, and builds the tree by connecting new nodes by the minimum edge that can possibly connect to the existing tree.
- Kruskal's Algorithm
    - A greedy algorithm for building a minimum spanning tree. At each time step, choose the edge with the lowest cost unless it connects nodes already part of itself (resulting in a cycle)