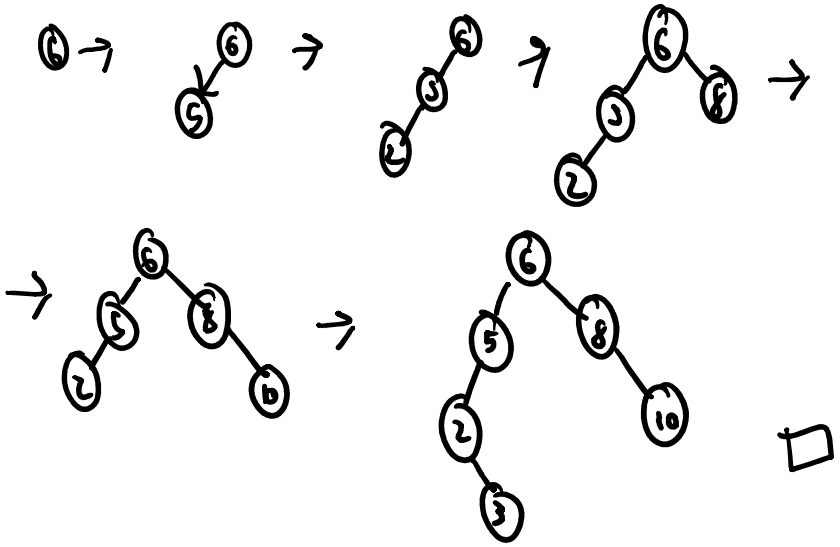
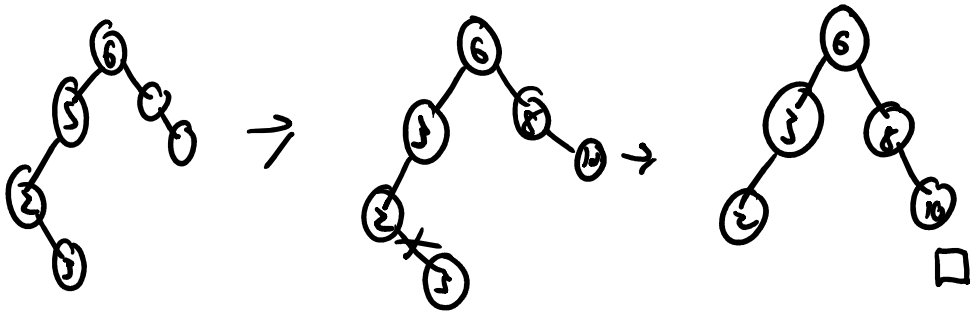
Problem 2 (Trees):

1. Binary Search Trees
    a. Stores things in a sorted way, easy to find
    b. O(log(n))
    c. O(log(n)) ←?
    d. Worst case find is a O(n) as everything must be traversed
    e. If the tree is not balanced.
2. BST examples
    a. Draw a tree when {6, 5, 2, 8, 10, 3} are inserted

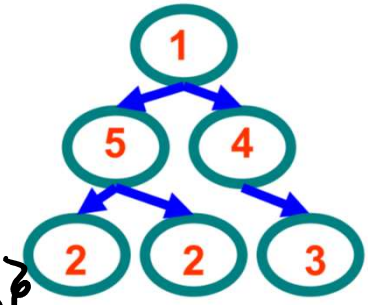


    b. Remove node 5 (to remove go left 1, then all the way right)

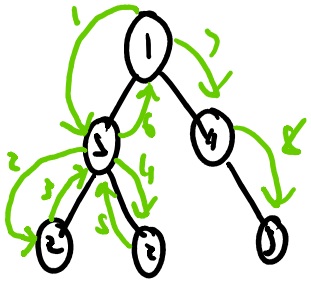


3. Traversals
    a. A tree traversal is the act of going through every node in the tree.
    b. The difference between depth and breadth first is depth will go all the way to the farthest down node on a branch before checking the next branch. Breadth will check the whole level before going further down.
    c. T?
    d. F

e. For the following tree provide an example of a:



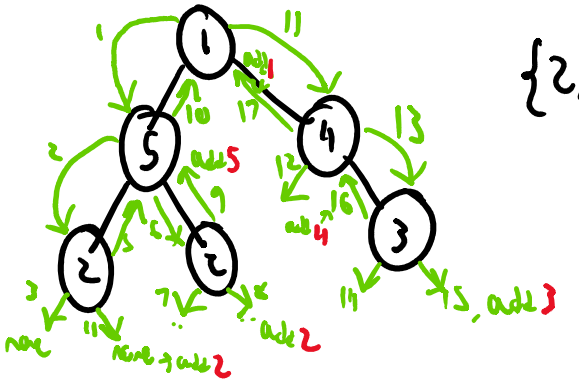
i. Preorder Traversal:



{1, 5, 2, 2, 4, 3}

#= visit order

ii. Post-order Traversal



{2, 2, 5, 3, 4, 1}

iii. In-order Traversal: {2, 5, 2, 1, 4, 3}

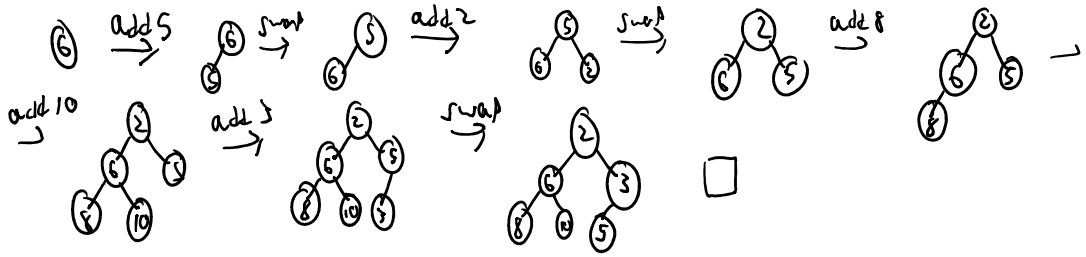
4. Write code given this definition for a binary tree:

```
public class BinarySearchTree <K extends Comparable<K>, V> {
    private class Node {
       private K key;
       private V data;
       private Node left, right;

       public Node(K key, V data) {
          this.key = key;
          this.data = data;
       }
    }
    private Node root;

}
```

a. (see code)

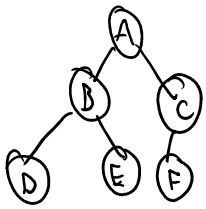
Problem 3 (Heaps)

1. Properties & characteristics
    a. Two key properties of a heap are the values at each node (and their ordering) and its shape (being complete). Heaps must have a specific shape to be valid, and subtrees must only contain either values less than the root (minheap), or values greater than the node (maxheap).
    b. BST has all values in the left subtree lower than the root, and all values in the right subtree are bigger than the root.
    c. Insert complexity is related to the height of the heap, since that height lets us know how many swaps will be needed to rebalance the heap. O(log(n))
    d. Finding values in the heap is O(n), since we do not know much about the interior ordering apart from increasing/decreasing as the levels move down.
    e. You can use an equation to know where each specific node would be stored in an array.
    f. Heaps are used to implement priority queues because they allow for values to be moved up in the queue based on their key value. This means some nodes are given 'priority' based on their key value, hence a priority queue.
2. Examples
    a. Heap created by inserting {6, 5, 2, 8, 10, 3}, in that order.

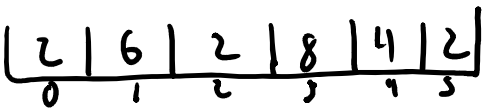


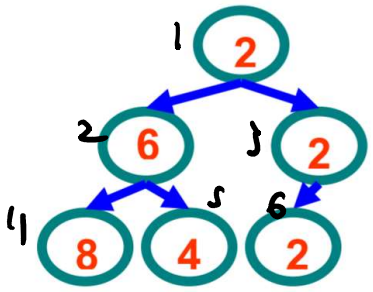
    b. Previous heap with the smallest value removed.

    c. Tree representation of {A, B, C, D, E, F}



d. Array representation of this specific tree





Problem 4 (Algorithmic Complexity)

1. Algorithmic complexity
   a. Algorithmic complexity is the behavior of the run time equation for large input amounts typically in terms of an input n.
   b. Benchmarking is how long the algorithm will actually take, it is less approximated than algorithmic complexity and will take into account variations in the hardware being benchmarked on.
   c. Best case looks at the fastest the algorithm could be given ideal inputs, usually not very helpful. Worst case looks at the slowest the algorithm could potentially be, useful for guaranteeing an upper limit on run time. Average case looks at the average run time over all inputs, most useful in practice.
   d. Big-O notation represents the simplified high order behavior of the complexity. Gives a general comparison that is relatively easy to compare with.
2. Circle critical sections:

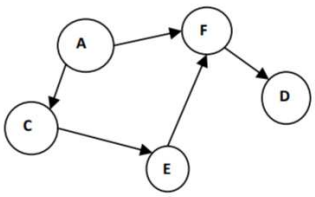
Calculate the asymptotic complexity of the code snippets below (using Big-O notation) with respect to the problem size n:

a.
```
for (i = 1; i < n; i = i * 2) {
    for (j = 1; j < n; j++) {
        ...
    }
}
```
f(n) = O( )

b.
```
for (i = 0; i < n - 2; i++) {
    for (j = 0; j < n; j = j * 2) {
        for (k = 1; k < 5000; k = k * 5) {
            ...
        }
    }
    for (j = 0; j < 1000 * n; j = j +2) {
        ...
    }
    for (j = 0; j < n / 5; j = j + 5) {
        ...
    }
}
```
f(n) = O( )

Problem 5 (Graphs)

1. Properties
    a. A directed graph has paths that only go one way, while an undirected graph has paths that can be taken either direction. Directed is a one-way street, undirected is a two way street.
    b. A path is an order of traversal between any two points. A cycle must start and end at the same node (potentially after traversing through another node)
    c. Can use a 2D matrix, but this is uses a lot of space. The other method is to use an adjacency list/set/map as in the project.
2. Traversals
    a. Graph traversal is more difficult because there is no explicit start and end, and there could be multiple ways to traverse between two points.
    b. Breadth-first goes through all nodes neighboring the start node, then all nodes neighboring those nodes, etc. Depth-first will go as far as it can down one path, then do the same for a different path of unvisited nodes.
3. Graph Traversals
    a. Two different BFS orders starting from A
        i. A>F>C>E>D
        ii. A>C>F>D>E
    b. Two different DFS orders starting from A
        i. A>F>D>C>E
        ii. A>C>E>F>D

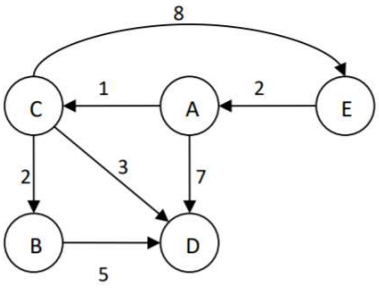


4. Single source shortest paths

After processing the first node. Write the name of the node you are processing here: ____A____

1 = none

Node	A	B	C	D	E
Cost	0	∞	1	7	∞
Predecessor	\	\	A	D	\



After processing the second node. Write the name of the node you are processing here: ____C____

Node	A	B	C	D	E
Cost	0	3	1	4	9
Predecessor	\	C	A	C	C

Problem 6 (Java Language Features)

1. Java Inner Classes
    a. Inner classes are **non-static** classes that are defined inside of another class.
    b. Nested classes are **static** classes that are defined inside of another class.
        i. Inner classes are just non-static nested classes, that's the difference.
    c. Inner classes should be used when the inner class only needs to be accessible to the class it is inside.
    d. Anonymous inner classes should be used when the full properties of a class are not needed, but some class functionality is needed to implement something (ie lambda expression or similar)
    e. Lambda expression only works for functional interfaces (those with 1 method). Anonymous inner classes can have more than 1 method defined, and can work for classes or interfaces, but methods not in the original class will not be callable.
    f. Node a = new Node(){blah blah blah};
2. Java support for OO programming
    a. All non-static initialization blocks are executed when objects are created (T)
    b. Code in initialization blocks is executed at the end of every constructor (F)
        i. Executes at the beginning of the constructor.
    c. If no visibility modifier is specified, methods are private by default (F)
        i. Methods are package visible by default.
    d. Protected access is less visible than package access (F)
        i. Protected is more visible than package. Protected is available to any subclasses, while package(default) is only available in the package.
3. Generics
    a. Advantages of using generics are:
        i. Works for many different objects
        ii. Worry less about the details
        iii. More portable
    b. public class example<E> implements Deque<E>{//code}
4. Lambda Expressions
    a. A lambda expression is a way to implement a functional interface (one with only 1 abstract method) as an anonymous class.
    b. Runnable interface implemented with lambda expression:

```
Runnable r = ()->{
    for(int i = 0; i <=100; i+=2){
            System.out.println(i);}
};
```

        i. Can be run by calling r.run();
    c. An interface must have only 1 abstract method to be implemented using lambda expression.

    d. Lol idk
5. Exceptions in Java
    a. Exceptions are errors raised due to unexpected things happening in the code
    b. Exceptions are used to represent errors that could potentially be handled by the code in some way. This contrasts with code that would halt on any error, or continue to run with garbage until a segmentation fault occurs (lol c)
    c. Exception should be errors that require fixing or issues that must not be ignored
    d. Try will run code that could produce an error (ie methods that 'throw' something), if something is thrown, it can be caught by the catch block so that the code can continue running. The finally block is **always run** last, as long as no errors cause the code to stop executing. The difference between try/catch and finally is that finally will always run, while the catch block will only run if an error occurs.
    e. Checked exception must be either propagated or caught, must be in a try/catch block (could work but the compiler won't compile it). Checked exceptions are used for more common errors. Code that throws unchecked exceptions does not need to be in a try/catch block. Unchecked exceptions normally indicate logic errors, null pointer errors, or other serious errors.
    f. What will be printed?
        i. ABD
        ii. ACD

Problem 7 (Multithreading & Synchronization)

1. Multithreading
    a. The motivation is to do things faster by having more processes running.
    b. New, runnable, running, blocked, dead.
        i. New is a yet to run thread, runnable, is one that can begin execution, running is one currently executing, blocked is a thread waiting for something, and dead is a finished thread.
    c. Calling the start method of a thread class will start the new thread of execution.
    d. Calling the join method of a thread will cause the current thread to wait for the spawned thread to finish.
    e. Scheduling the process of giving each thread access to the CPU in an efficient manner.
    f. With non-preemptive scheduling a process cannot be paused or stopped to allow another process to run. Preemptive scheduling allows for a process to be given higher priority. A high priority process will be cause the CPU to pause it's current process in order to do the high priority process immediately.
    g. Data races are when a section of memory is being read from and written two by no synchronous processes. The actual value stored in the memory cannot be confidently predicted by the program, meaning it would lead to unexpected results.
    h. (see code)

    i. Programs should avoid data races because they create unexpected results that cannot be dealt with efficiently (or sometimes at all).
2. Synchronization & deadlocks
    a. Synchronization is controlling which processes are accessing what with respect to time.
    b. Java programs should use synchronization to allow for effective multithreading which can avoid data races.
    c. Java locks are objects that only one process can have access to at a time. Threads can use the lock by using the synchronized method.
    d. Java locks can be used to allow for multiple processes to write/read to the same object without causes a data race.
    e. Deadlocks are when a thread won't release the lock, but other threads are waiting for it. This halts the code essentially.
    f. (see code)
3. Multithreading code
    a. If add() and remove() are called by multiple threads at the same time, a data race would occur. The actual outcome would be hard to predict.
    b. (see code) basically just put everything inside those methods into synchronized blocks.
4. (My code is meh and this is doable)
5. Threads could speed up the search operation in a BST by looking down each branch. Though, this isn't really faster if the tree is balanced, or anything, in fact its not really faster at all, it's just easier. The search operation of a BST is already very fast.

Problem 8 (Graphic User Interfaces)

1. In a GUI, the model is the data structures that provide the functionality. The view is the actual graphic being looked at, and the controller is the user.
2. These should be kept separate to make upgrading each easier, and to make porting them simpler.
3. Events are things that occur at anytime as the result of external input. They are actions like clicking on a button or moving a scroll bar.
4. Events are used in GUIs because it allows for the user to interact with the interface smoothly, and prevents unnecessary overhead caused by constant polling.

Problem 9 (Sorting & Algorithm Strategies)

1. Sorting algorithms
    a. Comparison sort is any type of sort that relies on pairwise key comparison
    b. An algorithm isn't a comparison sort when it uses some other property of the key to sort the data (ie radix sort)
    c. A stable sort is one that keeps equal keys in the same order as they started in.
    d. In-place sorts only require a constant amount of additional space to do the sort.
    e. An external sort is one that needs more space than is available in main memory.
    f. What is the average case complexity of sorting using:

i. Bubble sort: O(n^2)
ii. Heap sort: O(n*log(n))
iii. Quick sort: O(n*log(n))
iv. Counting sort: No on slides so who cares

g. Worst case complexity:
i. Selection sort: O(n^2)
ii. Tree sort: O(n^2)
iii. Heap sort: O(n*log(n)) (it's like tree sort but without the worst case)
iv. Radix sort: O(n)

h. Can the following be performed in a stable manner?
i. Selection sort: Yes
ii. Quick sort: No
iii. Counting sort: lol who knows probably

i. Can the following be performed using an in-place algorithm?
i. Selection sort: Yes
ii. Tree sort: No
iii. Merge sort: No

2. Algorithm strategies (not very sure about some of these)
a. Divide-and-conquer is the strategy of breaking the task up into smaller tasks that can all be done the same way. Smaller tasks must be of the same type, and do not necessarily need to overlap. Solve subproblems recursively.
b. Dynamic programming is divide-and-conquer, but look up solutions to subproblems if previously solved. Combine solutions to solve the original problem, store this solution.
c. Dynamic programming stores the result so that the second time the problem needs to be solved it can be done much faster using a look-up. On the first solve, the two are roughly the same (assuming no subproblems have been previously solved).
d. Backtracking will first check if a solution is known (or if the correct decision at a certain point is known). If it isn't, the algorithm will try and find a solution recursively by making a decision. If the decision doesn't lead to a solution, it will make the other decision and try again.
e. Heuristic algorithms will try and find a 'mostly acceptable' solution that is simple, but might not work for every situation. A greedy algorithm will try and find the best solution at each step, hoping to get the global optimum with enough steps. If you asked a heuristic algorithm for directions, it might just tell you 'go north'. Your destination might be north, but maybe you have to cross a bridge to the west first (so 'go north' doesn't work). If you asked a greedy algorithm for directions, if would take out a map and tell you to take all the shortest streets, as each would the shortest path you could take towards your destination at that time.
f. Brute force will try every option at every point until the optimal solution is found, while branch and bound will stop trying on options that don't seem like they will yield an optimal solution.

g. A reason to use dynamic programming is that it has all the speed benefits of divide and conquer, while also utilizing past results to make answering the same question multiple times much faster.
h. A reason to use backtracking is that it will utilize previous solutions to find new ones.
i. A reason to use a brute force algorithm is that it will guarantee that an answer is found, though it may take forever. Another reason is if there is no way to approach the problem (ie breaking OTP encryption).
j. Kruskal's algorithm is a greedy algorithm (picks best option at each step).
k. Greedy? It picks the best at each point (the least number of neighbors presumably) until it gets a solution.
l. Dynamic programming, since finding Fibonacci numbers can be sped up by using previous results (4! = 4*3!, if we've done 3! before we don't need to do it again).

Problem 10 (Hashing)

1. The advantages of hashing are quick look up times.
2. The java hash code contract requires that if object A equals object B, hash(A)=hash(B). Note that it is perfectly valid to have hash(A)=hash(B) when A!=B, as long as hash(A)=hash(B) when A=B the java hash code contract is satisfied.
3. Memory location of the integer, as it would be passed by value and therefore stored in a different location each time. This would satisfy the JHCC because the object would presumably store the integer in one location. Another possibility is to just always return 3, since we just want to satisfy the JHCC and we have no standards.
4. If you used HashMap or HashSet with a class that doesn't follow the JHCC, it would be impossible to find items after they were put into the map/set since you might not be able to recompute the hash used.

Problem 11 (Design Patterns)

1. Design patterns
    a. A design pattern is a way of organizing classes/objects.
    b. Design patterns are used whenever organization is needed?
    c. Iterator, Marker, State, Observer, Decorator, and Singleton.
        i. Example of a state would be a vending machine which enters different states depending on how much money is inserted.
        ii. Example of a decorator would be classes representing pizza, with different decorator classes representing toppings.
    d. (see code)
    e. (see code)
    f. Cloneable is a marker design pattern, implementing it just lets the code now it can be cloned. Serializable is marker design pattern.
    g. (see code)

Problem 12 (Linear Data Structures)

I was asked not to post the code for this 😊

You can do this, I believe in you