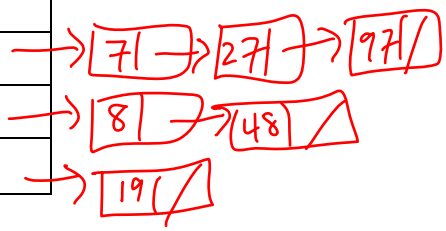
1) [11 points total] Hash Tables

For a) and b) below, insert the following elements in this order: 19, 48, 8, 27, 97, 7. For each table, TableSize = 10, and you should use the primary hash function h(k) = k%10. If an item cannot be inserted into the table, please indicate this and continue inserting the remaining values.

17au

a) Separate chaining hash table – use a sorted linked list for each bucket where the values are ordered by **increasing value**

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	



b) Quadratic probing hash table

0	
1	97
2	8
3	
4	
5	
6	7
7	27
8	48
9	19

c) What is the load factor in Table a)? **0.6**

d) Would you implement lazy deletion on Table a)? In 1-2 sentences describe why or why not. Be specific.

No. Deletion is equivalent in cost to a find operation (once found, just remove the node from the linked list). So you do not really gain much by implementing lazy deletion here – it will merely leave nodes marked deleted in the table, causing other finds/inserts/deletes to have to traverse over them. There could be a small advantage to leaving the node around if it was likely to be inserted again soon, but the advantages of this is probably outweighed by the disadvantages.

e) In 1-2 sentences (or pseudo code) describe how you would implement re-hashing on Table b). Be specific.

Two ways you could trigger rehashing are when the load factor on the table gets to be > 0.5 or when an insertion fails. To re-hash, first create a new hash table roughly 2x as large as this one (and a prime number). Traverse the original table from top to bottom, for every value encountered, calculate the hash function on the key and insert it into the new table (mod-ing by the new tablesize).

f) What is the big-O worst case runtime of an Insert in a hash table like Table a) containing N elements? **_O(N)_**

g) What is the big-O worst case runtime of determining what the maximum value is in a hash table like Table b) containing N elements? **_O(N)_**

2) [12 points total] Graphs!

a) **[2 points]** What is the big-O running time of Dijkstra's algorithm (assuming an adjacency list representation) if:

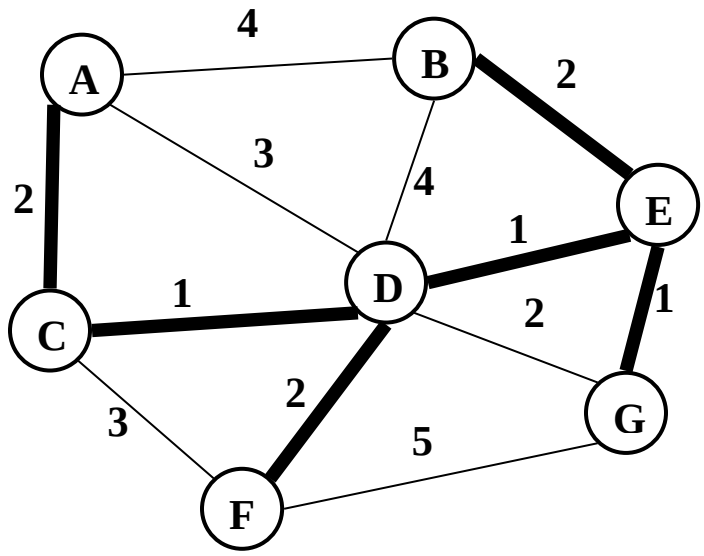
(i) A priority queue is used? **O(E log V + Vlog V) or just O(E log V)**

(ii) An unsorted list is used? **O(EV + V^2)**

b) **[2 points]** Which implementation of Dijkstra's (priority queue vs. unsorted list) is likely to be faster if the graph is known to be dense? **Explain your answer in ~one sentence for any credit**.

Priority Queue is better than unsorted list regardless of whether the graph is dense or sparse. O(log V) is always better than O(V) whether E is close to V or V^2 .

c) **[2 points]** Give a Minimum Spanning Tree (MST) of the graph below by highlighting the edges that would be part of the MST.



d) **[6 points]** For (ii) and (iv) below, ***you are given a perfect binary tree of height h containing n nodes***. Your answer should be an **exact formula**, such as 3/2 log h, or 5 × 3^n, not big-O notation.

(i) Depth First Search: What is the name of the data structure used in DFS? **stack**

(ii) What is the maximum size of that data structure during a DFS ?

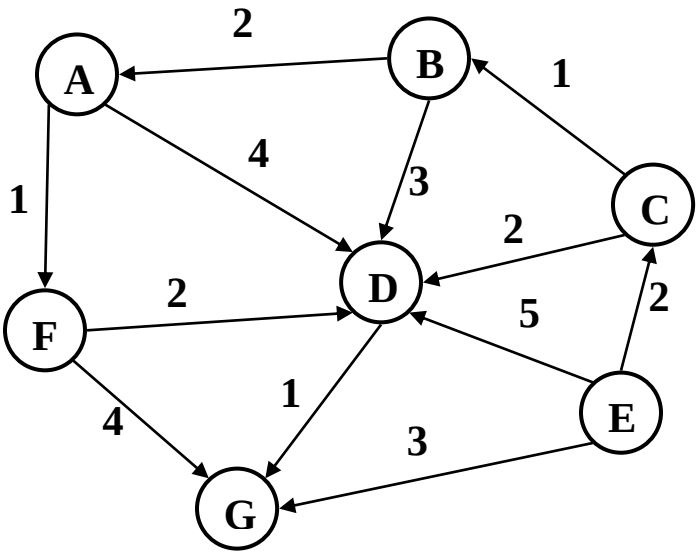
h + 1

(iii)Breadth First Search: What is the name of the data structure used in BFS? **queue**

(iv)What is the maximum size of that data structure during a BFS ?

2^h

3) [9 points total] More Graphs! Use the following graph for this problem:



a) [2 points] List a valid **topological ordering** of the nodes in the graph above (if there are no valid orderings, state why not).

E C B A F D G

b) [3 points] In lecture we described an optimization to the topological sorting algorithm that used a queue. Your partner proposes that you use a priority queue instead of a FIFO queue. What would be the worst case running time of this new version of topological sort (assuming an adjacency list representation of the graph is used)? **For any credit, briefly describe your answer with pseudo code or in a couple of sentences.**

```
calculateIndegreeOfAllVertices();   // O(E + V)
buildheapOfVertices();              // O(V)
for (ctr=0; ctr < numVertices; ctr++){ // V times
      v = deletemin();              // O (log V)
      put v next in output          // O(1)
      for each w adjacent to v {    // E times total
            decreasekey(w, 1);      // O(log V)
      }
}  // So we get: O( E + V + V + V log V + E log V) → O(E log V)
```

c) [2 points] ASSUMING the edges above are undirected and unweighted, give a valid breadth first search of this graph, starting at vertex A, **using the algorithm described in lecture**.

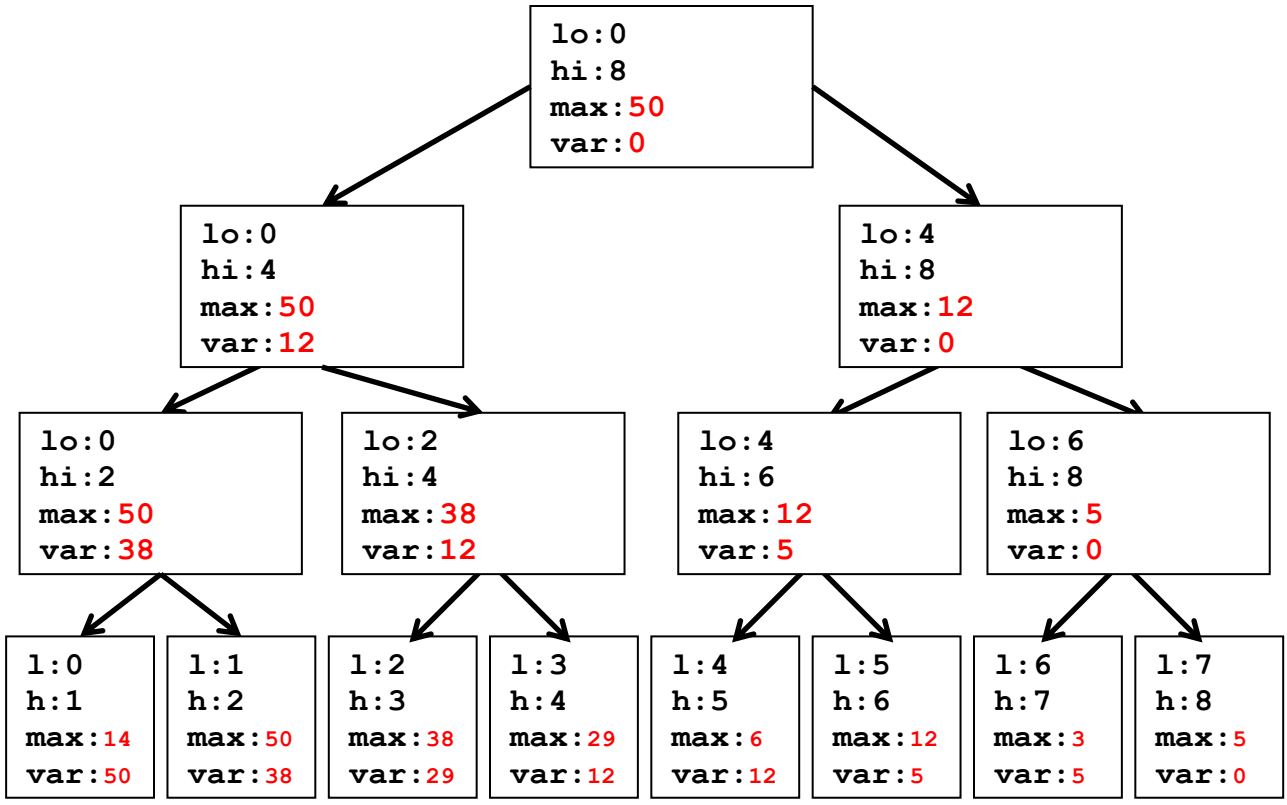
One possibility: **A B D F C E G**

d) [2 points] ASSUMING the edges above are undirected and unweighted, give a valid depth first search of this graph, starting at vertex A, **using the non-recursive algorithm described in lecture**.

One possibility: **A B C E G F D**

4) [10 points] Parallel "Suffix Max" (Like Prefix, but <u>from the Right</u> instead):

a) Given the following array as input, calculate the "suffix max" using an algorithm similar to the parallel prefix algorithm discussed in lecture. Fill the **output** array with the max of the values **contained in all of the cells to the right** (including the value contained in that cell) in the input array. The first pass of the algorithm is similar to the first pass of the parallel prefix code you have seen before. Fill in the values for **`max`** and **`var`** in the tree below. The output array has been filled in for you. Do not use a sequential cutoff. **You can assume that the array contains only positive integers.**



Index	0	1	2	3	4	5	6	7
Input	14	50	38	29	6	12	3	5
Output	50	50	38	29	12	12	5	5

b) How is the **`var`** value computed for the left and right children of a node in the tree. Give **<u>exact code</u>** (not just an English description) where **`p`** is a reference to the current tree node.

```
p.left.var =    Math.max(p.var, p.right.max)
```

```
p.right.var  =  p.var
```

c) How is **`output[i]`** computed? Give **<u>exact code</u>** assuming **`leaves[i]`** refers to the leaf node in the tree visible just above the corresponding location in the **`input`** and **`output`** arrays in the picture above.

```
output[i] = Math.max(leaves[i].max, leaves[i].var)
```

5) [14 points] In Java using the ForkJoin Framework, write code to solve the following problem:

• **Input**: An array of ints
• **Output**: the array index of the rightmost number greater than zero in the Input array.

For example, if input array is {-3, 2, 5, -2, 7, 0, -4}, the output would be 4, since that is the index of the value 7. If there are no values greater than 0 in the array then -1 should be returned.

- Do **not** employ a sequential cut-off: **the base case should process one element**. (You can assume the input array will contain at least one element.)
- Give a class definition, `RightmostPosTask`, **along with any other code or classes needed**.
- Fill in the function `findRightmostPos` **below.**

You may not use any global data structures or synchronization primitives (locks).

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.RecursiveAction;


class Main{
     public static final ForkJoinPool fjPool = new ForkJoinPool();

     // Returns the index of the rightmost positive value
     // in the array input. Returns -1 if no values > 0 in input.
     public static int findRightmostPos (int[] input) {

         return fjPool.invoke(
                      new RightmostPosTask(input, 0, input.length));
     }
}
```

Please fill in the function above and write your class on the next page.

5) (Continued) Write your class on this page.

```
public class RightmostPosTask extends RecursiveTask<Integer> {
   int[] array;
   int lo;
   int hi;


   public RightmostPosTask(int[] array, int lo, int hi) {
      this.lo = lo;
      this.hi = hi;
      this.array = array;
   }

   protected Integer compute() {
      if (hi - lo < 2) {
         if (array[lo] > 0) {
           return lo;
         } else {
           return -1;
         }
      } else {
         int mid = lo + (hi - lo) / 2;
         RightmostPosTask left = new RightmostPosTask (array,
lo, mid);
         RightmostPosTask right = new RightmostPosTask (array,
mid, hi);

         left.fork();
         int rightResult = right.compute();
         int leftResult = left.join();

         if (rightResult > leftResult) {
            return rightResult;
         } else {
            return leftResult;
         }
      }
   }
}
```

6) [13 points] Concurrency: The following class implements a Bank account class that keeps track of multiple Bank Accounts a user might have. Multiple threads could be accessing the same `BankAccounts` object.

```
public class BankAccounts {
      private Map<String, Double> acctsMap = new HashMap<>();
      ReentrantLock lock = new ReentrantLock();

      // Returns null if account does not exist
      public Double getBalance(String acctName) {
            lock.acquire();
            return acctsMap.get(acctName);
            Double balance = acctsMap.get(acctName);
            lock.release();
            return balance;
      }

      public Double withdraw(String acctName, Double amount) {
            lock.acquire();
            Double acctBalance = getBalance(acctName);

            if (acctBalance == null || acctBalance < amount) {
                  lock.release();
                  throw new InvalidTransactionException();

            }

            acctsMap.put(acctName, acctBalance - amount);
            lock.release();
            return amount;

      }

      // Deposit amount in acctName
      // Creates acctName if it does not already exist
      public void deposit(String acctName, Double amount) {
            lock.acquire();
            Double acctBalance = getBalance(acctName);

            if (acctBalance == null) {

                  acctBalance = 0.0;

            }

            acctsMap.put(acctName, acctBalance + amount);
            lock.release();

      }
}
```

6) (**Continued**)
a) Does the `BankAccounts` class above have (circle all that apply):

a race condition, potential for deadlock, **a data race**, none of these

If there are any problems, describe them in 1-2 sentences.

There is a data race on the `acctsMap`, e.g. one thread could be reading it in `getBalance` while another thread could be writing it in `withdraw` or `deposit`. Similarly multiple threads could be writing it at the same time by calling `withdraw` or `deposit` simultaneously. A data race is a race condition.

b) Suppose we made the `withdraw` method **`synchronized`**, and changed nothing else in the code. Does this modified `BankAccounts` class above have (circle all that apply):

a race condition, potential for deadlock, **a data race**, none of these

If there are any FIXED problems, describe why they are FIXED in 1-2 sentences. If there are any NEW problems, describe them in 1-2 sentences.

You can no longer have a race condition between two simultaneous calls to `withdraw`.

c) Modify the **code on the previous page** to use locks to avoid any of the potential problems listed above. Create locks as needed. Use any reasonable names for the locking methods you use. **DO NOT use `synchronized`**. You should create re-entrant lock objects as needed as follows (place this in your code as needed):

```
ReentrantLock lock = new ReentrantLock();
```

d) Clearly circle all of the **critical sections** in your **code on the previous page**.

7) [16 points] Sorting

a) **[2 points]** Give the ***recurrence*** for SEQUENTIAL Mergesort – worst case: (Note: We are NOT asking for the closed form.)

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$

b) **[3 points]** Give the ***recurrence*** for Quicksort (parallel sort & parallel partition) – worst case span: (Note: We are NOT asking for the closed form.)

$T(n) = T(n-1) + O(\log n)$

c) **[5 points]** Give the big-O runtimes requested below.

$O(\log^2 n)$ A) Quicksort (parallel sort & parallel partition) – best case span

$O(n \log n)$ B) Heapsort – worst case

$O(n^2)$ C) Insertion Sort – worst case

$O(n)$ D) Bucket Sort – best case

$O(n \log n)$ E) Mergesort (sequential) – worst case

d) **[1 point]** Is the version of Quicksort described in lecture a stable sort?

YES **NO**

e) **[2 points]** In 1-2 sentences, describe what it means for a sort to be stable?

In case of ties during the sort, the original ordering of values is preserved.

f) **[3 points] Radix Sort:** Give a formula for the worst case big-O running time of radix sort. For full credit, your formula should include all of these variables:

n	– the number of values to be sorted
max_value	– the values to be sorted range from 0 to max_value
radix	– the radix or base to be used in the sort

Answer: $O((\log_{radix} max\_value) \cdot (n + radix))$

8) **[9 points] P, NP, NP-Complete**

a) **[2 points]** "NP" stands for ____ **Nondeterministic Polynomial** _____

b) **[2 points]** What does it mean for a problem to be in NP?

Given a candidate solution, we can verify whether the solution is correct in polynomial time.

c) **[5 points]** For the following problems, circle **ALL** the sets they (most likely) belong to:

Finding the shortest path from one vertex to another vertex in a weighted directed graph	**NP**	**P**	NP-complete	None of these
Finding a cycle that visits each edge in a graph exactly once	**NP**	**P**	NP-complete	None of these
Determining if a program will run forever	NP	P	NP-complete	**None of these**
Finding the prefix sum of an array in parallel using 10 processors	**NP**	**P**	NP-complete	None of these
Finding a path that starts and ends at the same vertex that visits every vertex exactly once	**NP**	P	**NP-complete**	None of these