Final Exam

Autumn 2024

Name Answer Key

Net ID (@uw.edu)

Academic Integrity: You may not use any resources on this exam except for your one-page (front and back) reference sheet, writing instruments, your own brain, and the exam packet itself. This exam is otherwise closed notes, closed neighbor, closed electronic devices, etc.. The last three pages of this exam provide a list of potentially helpful identities, the code referenced in Section 6, and room for scratch work. Please detach those last three pages from the exam packet. No markings on these last three pages will be graded. Your answer for each question must fit in the answer box provided.

Instructions: Before you begin, **Put your name and UW Net ID at the top of this page.** Make sure that your name and ID are LEGIBLE. Please ensure that all of your answers appear within the boxed area provided.

Section	Max Points
Asymptotic Analysis	9
Pre-Midterm Data Structures	11
Sorting	10
Graphs	18
Parallelism	16
Concurrency	13
Extra Credit	(+2)
Total	77

Section 1: Asymptotic Analysis

(2 pts)**Question 1: Asymptotic Analysis of Code**

For this question we will analyze the asympotic running time of the following code, which takes a list of integers as input. First, it adds all sums of pairs of values into an AVL tree. Then it iterates over all keys (using an in-order traversal) and adds those to a binary min heap. Finally, it returns the smallest value from the min heap.

For each box below, give a simplified Θ bound on the best and worst case running times for the given portion of the code. (By simplified, we mean that it should contain no constant coefficients or non-dominant terms.)

```
1  public int doStuff(ArrayList<Integer> nums){
2      AVLTree<Integer,Integer> d = new AVLTree<>();
3      MinHeap<Integer> pq = new MinHeap<>();
4      for(int i = 0; i < nums.size(); i++){
5          for(int j = i; j < nums.size(); j++){
6              int x = nums.get(i)+nums.get(j);
7              d.insert(x, x);
8          }
9      }
10     for(Integer key : d.getKeys()){
11         pq.add(key);
12     }
13     return pq.extractmin();
14 }
```

1. Best case lines 4-9: $\Theta($ n^2 $)$
2. Worst case lines 4-9: $\Theta($ $n^2 \log n$ $)$
3. Best case lines 10-13: $\Theta($ 1 $)$
4. Worst case lines 10-13: $\Theta($ n^2 $)$

(3 pts)**Question 2: Which is larger?**

For each pair of functions $f(n)$ and $g(n)$ below, select the choice that characterizes the asymptotic relationship between f and g. Write the letter corresponding with your answer in the box provided.

1. $f(n) = n^{\log_2 n}$, $g(n) = n^2$

   A. $f(n) \in \Theta(g(n))$
   B. $f(n) \in O(g(n))$ and $f(n) \notin \Theta(g(n))$
   C. $f(n) \in \Omega(g(n))$ and $f(n) \notin \Theta(g(n))$

   C

2. $f(n) = \log_2(n^n)$, $g(n) = n^2$

   A. $f(n) \in \Theta(g(n))$
   B. $f(n) \in O(g(n))$ and $f(n) \notin \Theta(g(n))$
   C. $f(n) \in \Omega(g(n))$ and $f(n) \notin \Theta(g(n))$

   B

3. $f(n) = n^{\log_n 2}$, $g(n) = n$

   A. $f(n) \in \Theta(g(n))$
   B. $f(n) \in O(g(n))$ and $f(n) \notin \Theta(g(n))$
   C. $f(n) \in \Omega(g(n))$ and $f(n) \notin \Theta(g(n))$

   B

(4 pts)**Question 3: Recurrence Relation**

The recursive method provided below was not written to compute any meaningful function. It's behavior, though, is that it recurses upon the subset of numbers in the given list which are not divisible by the first number. The base cases occur when the list is either empty, or the first value is not positive. In each recursive case the first value will be divisible by itself, and so this code does not have an infinite loop.

First, express the worst case running time of the code as a recurrence relation. Next solve that recurrence relation using any method of your choice and express the running time as a simplified Θ bound.

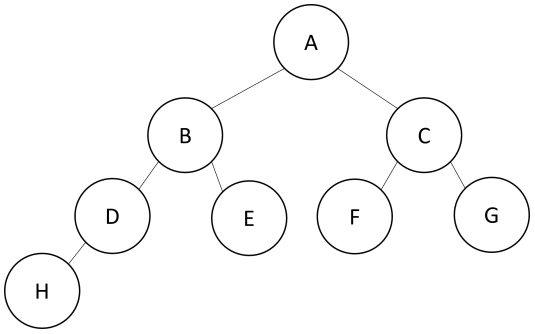
```
boolean myFunc(ArrayList<Integer> nums){
    if(nums.size() == 0){ return false; }
    if(nums.get(0) <= 0){ return true; }
    int denom = nums.get(0);
    ArrayList<Integer> next = new ArrayList<>();
    for(int x : nums){
        if(x % denom != 0){
            next.add(x);
        }
    }
    return myFuncRec(next);
}
```

1. Recurrence Relation: $T(n) = T(n-1) + n$

2. Solved and simplified Θ bound: n^2

Section 2: Pre-Midterm Data Structures

(3 pts)**Question 4: Max Heap - Always, Sometimes, Never**
Suppose that the tree below is a valid maxheap, and its contents are integers represented by the variables A through H. For each statement below, indicate whether it is always true, sometimes true, or never true by writing "A", "S" or "N" in the box provided. (You may assume all of the heap's contents are unique).



1. $B < C$

   S

2. $H < C$

   S

3. H is the smallest value in the heap

   S

4. H was the most-recently added value in the heap.

   S

5. After inserting one more item (with no other intermediate operations), H will still be a leaf.

   A

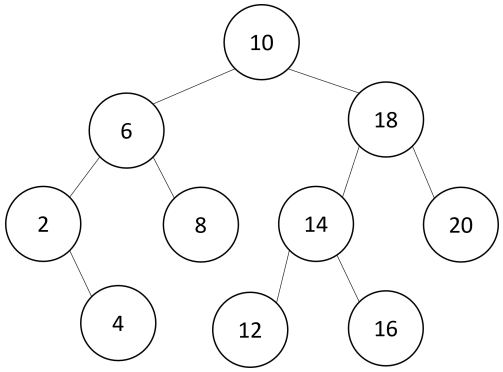
6. After removing the largest item (with no other intermediate operations), H will still be a leaf.

   S

(4 pts)**Question 5: AVL Tree Rotations**

Use the following AVL Tree to complete the table. The first column should give an item not already in the tree. The second column should identify the type of rotation(s) performed (in order). The third column identifies the value of the parent of the new node after the insertion is complete (so ***AFTER any rotations have been performed***). If the new node is the root then write "root". The first row is done for you.

Each row's value will be inserted independent of the other rows. I.e., "reset" to this tree between rows.



New Value Inserted	Rotation type/types	Parent of Inserted Node
21	None	20
3	right rotation then left rotation	6
13	right	12
17	left then right	18
5	left	4

(4 pts)**Question 6: Double Hashing**

Insert 3, 13, 20, 73, 30, 41 (in that order) into the open addressing hash table below. You should use the primary hash function

$$h(k) = k\%10$$

In the case of collisions, use double hashing for collision resolution where the secondary hash function is

$$g(k) = 1 + (k\%7)$$

If an item cannot be inserted into the table, indicate this and continue inserting the remaining values.

In case it helps, we have calculated $k\%7$ for each key k for you (note that these are not $g(k)$).

$$3\%7 = 3$$
$$13\%7 = 6$$
$$20\%7 = 6$$
$$73\%7 = 3$$
$$30\%7 = 2$$
$$41\%7 = 6$$

Items that could not be inserted:

0	13
1	73
2	
3	3
4	
5	
6	30
7	20
8	41
9	

Section 3: Sorting

(4 pts)**Question 7: World Cup Sorting**
For each scenario below, select the sorting algorithm property the situation most needs, then select the *fastest* sorting algorithm with that property. Select only from the options provided.

1. World cup teams earn points for each game they win or tie. When ranking teams, points earned is most important. If two teams are tied by points then the team with the better score differential is ranked higher. If there is a tie by points and differential then total goals scored is used. If we wanted to rank all world cup teams, which algorithm should we use?

   Property Options: In Place, Stable, Adaptive, Online, Non-Comparison-Based.

   Property Needed: Stable

   Algorithm Options: Quick Sort, Insertion Sort, Heap Sort, Merge Sort.

   Algorithm Suggestion: Merge

2. Throughout the history of the world cup there have been over 10,000 players. Miroslav Kosle of Germany has the record number of world cup goals scored at 16. If we wanted to sort all players by number of goals scored, what algorithm should we use?

   Property Options: In Place, Stable, Adaptive, Online, Non-Comparison-Based.

   Property Needed: Non-Comparison-Based

   Algorithm Options: Quick Sort, Insertion Sort, Heap Sort, Bucket Sort.

   Algorithm Suggestion: Bucket Sort

(2 pts)**Question 8: Non-Comparison Sorting**
We showed that when using a comparison-based sorting algorithm, we could not have a running time asymptotically faster than $n \log n$. However, if we do not rely on comparisons we could have linear time sorting algorithms. Why would we ever use a comparison-based sorting algorithm? Answer using 1-2 sentences.

A non-comparison-based algorithm is less general because it relies on additional assumptions about the contents of the list to order them.

(2 pts)**Question 9: Quick Sort Runtime**
Using 1-2 sentences, explain why using a random value of the pivot often results in a faster algorithm compared to using the item at the lowest index.

In practices, lists are approximately sorted, and in that case the subproblems' sizes will be unbalanced.

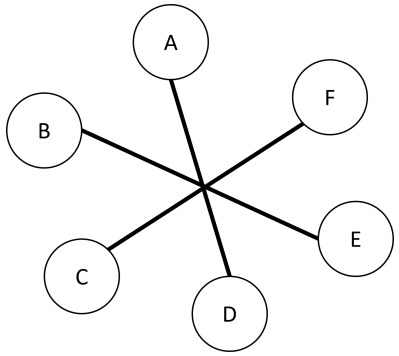
(2 pts)**Question 10: Decision Trees**
Recalling our decision tree argument to show a running time lower bound for comparison-based sorting, explain what the height of the tree represented. Give your answer using 1-2 sentences.

The height of the tree was the worst case number of comparisons done by the algorithm.

Section 4: Graphs

(3 pts)**Question 11: Undirected Graph Properties**

Use the following undirected graph for all subquestions below. Each subquestion names a graph property. Each of the answer options will be a collection of edges. Select the answer choice such that if those edges are added to the graph, then the graph will have the named property. If multiple choices result in the graph having the property, select the one with the fewest edges.



1. Connected
   A. The graph is already connected
   B. (A, B)
   C. $(A, B), (A, E)$
   D. $(A, B), (A, E), (B, C)$
   E. None of the above result in a connected graph.

   D

2. Acyclic
   A. The graph is already acyclic
   B. $(F, B), (C, A)$
   C. $(E, A), (D, E)$
   D. $(A, B), (B, C), (C, D)$
   E. None of the above result in an acyclic graph.

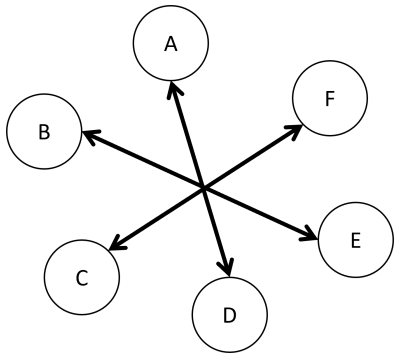
   A

3. Tree
   A. The graph is already a tree
   B. $(F, B), (C, A)$
   C. $(E, A), (D, E)$
   D. $(A, B), (B, C), (C, D)$
   E. None of the above result in a tree.

   B

(3 pts)**Question 12: Directed Graph Properties**

Use the following directed graph for all subquestions below. Each subquestion names a graph property. Each of the answer options will be a collection of edges. Select the answer choice such that if those edges are added to the graph, then the graph will have the named property. If multiple choices result in the graph having the property, select the one with the fewest edges.



1. Strongly Connected
   A. The graph is already strongly connected
   B. $(A, B), (B, F)$
   C. $(A, B), (B, F), (C, D)$
   D. $(A, B), (B, A), (B, F), (F, B)$
   E. None of the above result in a strongly connected graph.

   C

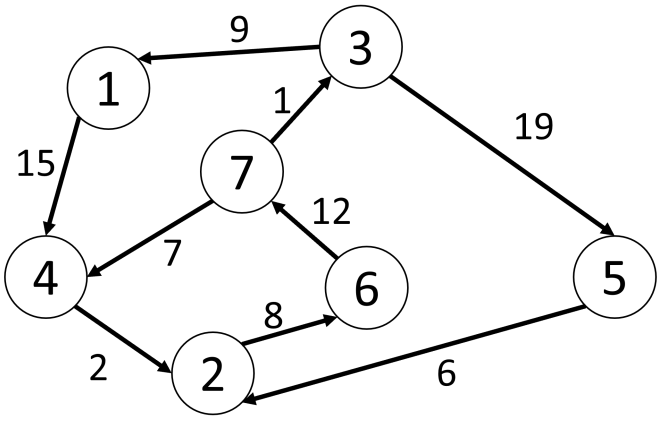
2. Weakly Connected
   A. The graph is already weakly connected
   B. $(A, B), (B, F)$
   C. $(A, B), (B, F), (C, D)$
   D. $(A, B), (B, A), (B, F), (F, B)$
   E. None of the above result in a weakly connected graph.

   B

3. Has a Topological Ordering
   A. The graph is already has a topological ordering
   B. $(A, B), (B, F)$
   C. $(A, B), (B, F), (C, D)$
   D. $(A, B), (B, A), (B, F), (F, B)$
   E. None of the above result in a graph with a topological ordering.

   E

All problems on this page will reference this graph.



(2 pts)**Question 13: BFS**
List the nodes in an order that they might be removed from the queue in a BFS starting from node 3.

BFS Order: 3,[1 and 5], [2 and 4], 6, 7
(e.g. 3,1,5,2,4,6,7)

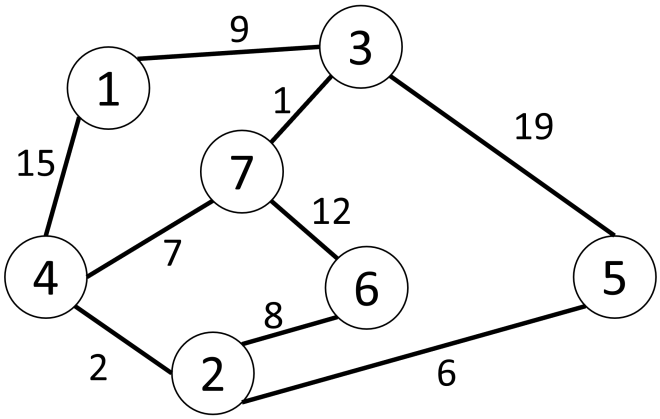
(2 pts)**Question 14: DFS**
List the nodes in an order that they might be removed from the stack in a DFS starting from node 3.

DFS Order: (3, 1, 4, 2, 6, 7, 5) or
(3, 5, 2, 6, 7, 4, 1)

(2 pts)**Question 15: Dijkstras**
List the nodes in an order that they might be removed from the priority queue when running Dijkstra's algorithm starting from node 3.

Dijkstra's Order: 3, 1, 5, 4, 2, 6, 7

All questions on this page will reference this graph (which is the same as the previous but undirected).



(3 pts)**Question 16: Kruskals's**
What are the weights of the first three edges added to the minimum spanning tree when running Kruskal's algorithm?

1. First edge's weight: 1
2. Second edge's weight: 2
3. Third edge's weight: 6

(3 pts)**Question 17: Prim's**
What are the weights of the first three edges added to the minimum spanning tree when running Prims's algorithm starting with node 3?

1. First edge's weight: 1
2. Second edge's weight: 7
3. Third edge's weight: 2

Section 5: Parallelism

(8 pts)**Question 18: ForkJoin**

For this question you will complete a parallel implementation of the following sequential method `fillSubtreeSums` using the Java ForkJoin Framework. Consider this to be a method inside of a binary search tree class.

```
public class BST{
    public BSTNode root;
    ... //suppose other fields and constructors go here
    public class BSTNode{
        public int data;
        public int sum;
        public BSTNode left;
        public BSTNode right;
    }
    public void fillSubtreeSums(){
        fill(root);
    }
    public int fill(BSTNode curr){
        if(curr == Null){
            return 0;
        }
        int leftsum = fill(curr.left);
        int rightsum = fill(curr.right);
        curr.sum = leftsum + rightsum + curr.data;
        return curr.sum
    }
}
```

This method populates the `sum` field for each node in a binary search tree, then returns the `sum` value of the current node. When finished, each node's `sum` field will contain the sum of the `data` fields for all nodes in the current subtree (so `curr` and all of its descendants).

On the next page we have an incomplete parallel implementation of `fillSubtreeSums` using ForkJoin. In particular, we have provided:

– A BST class that has the `fillSubtreeSums` method.

– Fields for a `FillTask` class

– The constructor for the `FillTask` class

– The signature of the `compute` method of `FillTask` including the base case.

And the code is missing:

– The body of the `fillSubtreeSums` method, which should create an instance of `FillTask` and then call the `invoke` method of `ForkJoinPool`. (Starts on line 16)

– The class that `FillTask` should extend. (Line 20)

– The parallelized portion of the `compute` method. (Starts on line 32)

Complete our implementation by providing the missing code in the boxes following the code.

```
1  import java.util.concurrent.*;
2
3  public class BST {
4      private BSTNode root;
5      public static final ForkJoinPool POOL = new ForkJoinPool();
6      ... //suppose other fields and constructors go here
7
8      public class BSTNode{
9          public int data;
10         public int sum;
11         public BSTNode left;
12         public BSTNode right;
13     }
14
15     public void fillSubtreeSums(){
16         // Part 1 answer will go here
17     }
18 }
19
20 public class FillTask extends ??? { // Part 2 will replace the ???
21
22     public BSTNode curr;
23
24     public FillTask(BSTNode curr){
25         this.curr = curr;
26     }
27
28     public Integer compute(){
29         if(curr == null){
30             return 0;
31         }
32         // Your implementation of compute from Part 3 will go here.
33     }
34 }
```

Finish the code in the boxes below:

1. Implement the body of `fillSubtreeSums` in the box provided. The code you provide will be placed starting at line 16 of the code above

```
POOL.invoke(new FillTask(root));
```

2. Finish line 20 from the code above by filling the in the ??? with the class that `fillSubtreeSumsTask` should extend.

```
RecursiveTask<Integer>
```

3. Finally, finish the `compute` method. Your code will begin on line 32 above.

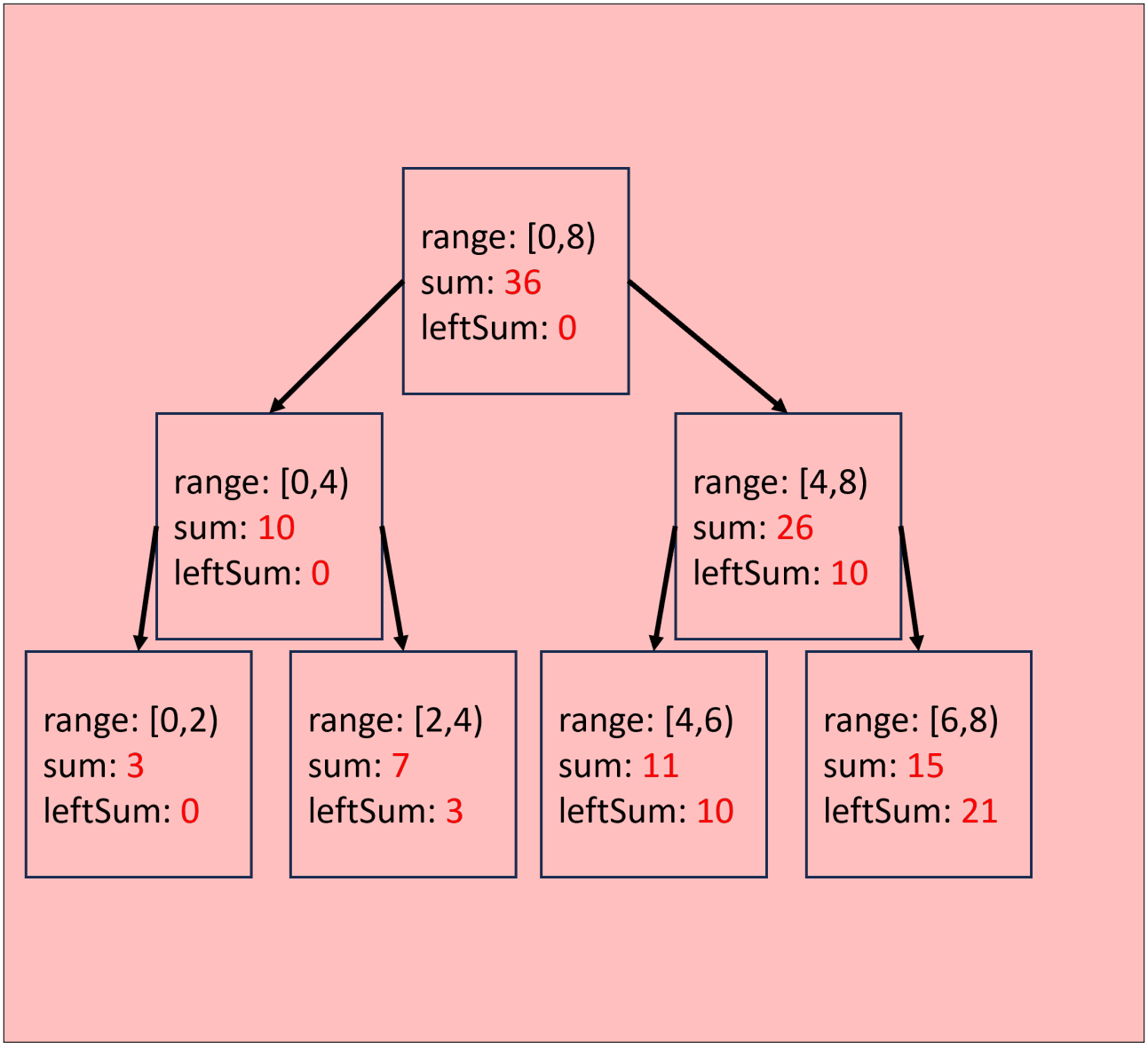
```
FillTask left = new FillTask(curr.left);
left.fork();
FillTask right = new FillTask(curr.right);
curr.sum = curr.data + right.compute() + left.join();
return curr.sum;
```


(5 pts)**Question 19: Parallel Prefix**
For this question we will be running the parallel prefix sum algorithm discussed in class

1. Recall that the first pass of the algorithm constructs a tree object. Each node in this tree has 4 fields - the range represented as fields `lo` and `hi`, `sum`, and `leftSum`. Of these fields, which are filled in during the first pass?

   lo, hi, and sum

2. Using the input array `[1, 2, 3, 4, 5, 6, 7, 8]` and a sequential cutoff of 2 (therefore your tree will have 4 leaves), draw the tree object as it would appear after finishing the second pass of the algorithm. Your drawing should make clear the range, `sum`, and `leftSum` for each node.



(3 pts)**Question 20: Amdahl's Law**

Suppose we have a program in which a $\frac{5}{6}$ proportion can be parallelized with perfect linear speedup. Using $T_1 = 1$, answer the following using Amdahl's Law:

1. What is T_2?

   $\frac{7}{12}$

2. What is T_5?

   $\frac{1}{3}$

3. What is T_{10}?

   $\frac{1}{4}$

Section 6: Concurrency

The next two questions reference the code below. This code implements an operation to swap values in an array list. You should assume there are no concurrency issues related to any other methods.

```
1  public class ParallelArrayList{
2      // Suppose other methods and fields are here.
3      public void swap(int a, int b){
4          Object aObj = get(a);
5          Object bObj = get(b);
6          synchronized(aObj){
7              synchronized(bObj){
8                  set(a, bObj);
9                  set(b, aObj);
10             }
11         }
12     }
13 }
```

(2 pts)**Question 21: Deadlock**
The If two threads are running `swap` there is a potential for deadlock. Describe an interleaving that results in deadlock.

if two threads call swap with opposite choice of a and b, then each locks their respective aObj before the other acquires bObj.

(3 pts)**Question 22: Is There Still Deadlock?**
Suppose we rewrite `swap` to try and fix the potential for deadlock. For each option below, indicate whether or not there is still potential for deadlock by writing "Yes" or "No" in the box provided.

```
public void swap(int a, int b){
    Object aObj = get(a);
    Object bObj = get(b);
    if(a < b){
        synchronized(aObj){
            synchronized(bObj){
                set(a, bObj);
                set(b, aObj);
            }
        }
    }
        else{
        synchronized(bObj){
            synchronized(aObj){
                set(a, bObj);
                set(b, aObj);
            }
        }
    }
}
```

1. Still has Deadlock?

Yes

```
public void swap(int a, int b){
    Object aObj = get(a);
    Object bObj = get(b);
    if(aObj.hashCode() < bObj.hashcode()){
        synchronized(aObj){
            synchronized(bObj){
                set(a, bObj);
                set(b, aObj);
            }
        }
    }
    else{
        synchronized(bObj){
            synchronized(aObj){
                set(a, bObj);
                set(b, aObj);
            }
        }
    }
}
```

2. Still has Deadlock?

Yes

```
public synchronized void swap(int a, int b){
    Object aObj = get(a);
    Object bObj = get(b);
    synchronized(aObj){
        synchronized(bObj){
            set(a, bObj);
            set(b, aObj);
        }
    }
}
```

3. Still has Deadlock?

No

The remaining questions within this section reference the code below (this code is also available on a detachable page at the end of the exam). This code implements a class called `GuestList`. The `GuestList` class maintains two lists. The first is called `invited`, which has a list of people who have been invited to an event. The second is called `attending`, and these are the people who have accepted the invitation and therefore will attend the event. When run sequentially, all guests in `attending` must also be in `invited`. The questions below relate to one or more race conditions that could cause this to be untrue when using parallelism.

```
1  public class GuestList{
2      Set<String> invited;
3      Set<String> attending;
4      public GuestList(){
5          invited = new TreeSet<>();
6          attending = new TreeSet<>();
7      }
8      public synchronized void invite(String guest){
9          invited.add(guest)
10     }
11     public synchronized void accept(String guest){
12         if (invited.contains(guest) && !attending.contains(guest))
13             attending.add(guest);
14     }
15     public void uninvite(String guest){
16         synchronized(invited){
17             if(invited.contains(guest))
18                 invited.remove(guest);
19         }
20         synchronized(attending){
21             if(attending.contains(guest))
22                 attending.remove(guest);
23         }
24     }
25 }
```

(3 pts)**Question 23: Race Condition**
Describe a race condition which results in a guest appearing in `attending` but not `invited`.

Consider that there is a guest who is in invited and not attending. Next consider thread 1 running accept and thread 2 running uninvite. This interleaving produces this issue: Thread 2 first does lines 16-19, then Thread 1 does line 9, then Thread 2 does lines 20-24.

(2 pts)**Question 24: Data Race or Bad Interleaving?**
Is the error in this code a Data Race or a Bad Interleaving error? Write either DR or BI in the box to indicate your answer.

BI

(3 pts)**Question 25: Still a Race Condition?**
Below we provide alternative implementations of `invite`, `accept`, `uninvite`. For each, indicate whether it is still possible for a guest to appear in `attending` but not `invited`. For each option we summarize the differences compared to the original.

Difference: We swapped the order that we remove from `invited` and `attending` in `uninvite`.

```
public synchronized void invite(String guest){
    invited.add(guest);
}
public synchronized void accept(String guest){
    if (invited.contains(guest) &&
            !attending.contains(guest))
        attending.add(guest);
}
public void uninvite(String guest){
    synchronized(attending){
        if(attending.contains(guest))
            attending.remove(guest);
    }
    synchronized(invited){
        if(invited.contains(guest))
            invited.remove(guest);
    }
}
```

1. Still possible to be attending but not invited?

Yes

Difference: We changed the locks in `invite` and `accept` to be `invited` and `attending`, respectively.

```
public void invite(String guest){
    synchronized(invited) {
        invited.add(guest); }
}
public void accept(String guest){
    synchronized(attending){
        if (invited.contains(guest) &&
                !attending.contains(guest))
            attending.add(guest); }
}
public void uninvite(String guest){
    synchronized(invited){
        if(invited.contains(guest))
            invited.remove(guest);
    }
    synchronized(attending){
        if(attending.contains(guest))
            attending.remove(guest);
    }
}
```

2. Still possible to be attending but not invited?

No

Difference: We changed the locks in `invite` and `accept` to be `invited` and both of `invited` and `attending`, respectively.

```
public void invite(String guest){
    synchronized(invited) {
        invited.add(guest); }
}
public void accept(String guest){
    synchronized(invited){
        synchronized(attending){
            if (invited.contains(guest) &&
                    !attending.contains(guest))
                attending.add(guest);
    }}
}
public void uninvite(String guest){
    synchronized(invited){
        if(invited.contains(guest))
            invited.remove(guest);
    }
    synchronized(attending){
        if(attending.contains(guest))
            attending.remove(guest);
    }
}
```

3. Still possible to be attending but not invited?

No

Extra Credit

(2 pts)**Question : Well Done!**
Overall, Nathan has felt that this has been an exceptional group of students, and is very proud of you all! More importantly, though, you should be proud of yourselves! Name one thing you accomplished this quarter that you're proud of.

...

Identities

Nothing written on this page will be graded.

Summations

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ for } |x| < 1$$

$$\sum_{i=0}^{n-1} = \sum_{n}^{i=1} = n$$

$$\sum_{i=0}^{n} i = 0 + \sum_{n}^{i=1} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

$$\sum_{i=0}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$$

$$\sum_{i=0}^{n-1} x^i = \frac{1-x^n}{1-x}$$

$$\sum_{i=0}^{n-1} \frac{1}{2^i} = 2 - \frac{1}{2^{n-1}}$$

Logs

$$x^{\log_x(n)} = n$$

$$\log_a(b^c) = c \log_a(b)$$

$$a^{\log_b(c)} = c^{log_b(a)}$$

$$\log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$

Section 6 code

This code is referenced in Questions 23,24, and 25 in section 6 of the exam. The same code also appears within that section. It is provided here so you can detach it for more convenient reference.

```
1  public class GuestList{
2      Set<String> invited;
3      Set<String> attending;
4      public GuestList(){
5          invited = new TreeSet<>();
6          attending = new TreeSet<>();
7      }
8      public synchronized void invite(String guest){
9          invited.add(guest)
10     }
11     public synchronized void accept(String guest){
12         if (invited.contains(guest) && !attending.contains(guest))
13             attending.add(guest);
14     }
15     public void uninvite(String guest){
16         synchronized(invited){
17             if(invited.contains(guest))
18                 invited.remove(guest);
19         }
20         synchronized(attending){
21             if(attending.contains(guest))
22                 attending.remove(guest);
23         }
24     }
25 }
```

Scratch Work

Nothing written on this page will be graded.

