CSE 12 SP24 Final Exam

Do not begin until instructed.

This exam is closed book, closed notes. No study aids or electronic devices are allowed. Turn off your phone and have no bags open.

You must hand in all sheets, including scratch paper, at the end of the exam.

This exam is designed to assess **your** understanding of topics from CSE12. You cannot communicate with other students during the exam. You cannot discuss this exam with anyone from now until you receive your grade on the exam. This includes posting about the exam on Piazza or elsewhere online.

Please write “I excel with integrity” in the box below:

Sign your name: _____

Print your name: _____

Print your email: _____

PID: _____

You have 175 minutes to complete the exam. The exam is in three parts. Each part corresponds to one of the midterms. For each part, we will take the highest percentage between the part and the corresponding midterm exam. If you wish to only use the midterm exam score for a part, you may skip that part and only take the parts you need to improve your grade.

There are three answer sheets. **Fill in your name and PID on all answer sheets even if you are skipping a part. Write all your answers on the correct answer sheet.** Work to maximize points. If you get stuck, work through other problems, and come back to it.

Once the exam starts, you can rip off/remove the answer sheets.

In general, if you think you've spotted a typo in the exam, do your best to answer in the spirit of the question. Keep in mind that some questions have interesting code examples with intentional bugs for you to find as part of the question. If you ask a question during the exam, we may decline to answer. In general, assume that any necessary libraries (JUnit, ArrayList, List, and so on) have been imported.

Stay calm – you can do this!

CSE 12 SP24 Final – Part 1 – Answer Sheet

Name: _____ PID: _____

Write all answers below. There are 23 total points.

Question 1 (*8 pts, 3 pts each for Part 1-2 & 1 pt each for Part 3-4)*

1:. implementation of maybeEnqueue()

2: implementation of dequeue()

3	**4**

Question 2 *(2 pts, 1 pt each)*

1	**2**

Question 3 *(7 pts, 5 pts for Part 1 & 2 pts for Part 2)*

1: implementation of replaceValue()

2: implementation of testStack()

Question 4 *(4 pts, 1 pt each)*

1	**2**	**3**	**4**

Question 5 *(1 pt)*

Question 6 *(1 pt)*

CSE 12 SP24 Final – Part 2 – Answer Sheet

Name: _____ PID: _____

Write all answers below. There are 25 total points.

Question 1

Part 1 *Write Yes or No for each output (3 pts, 0.5 pts each)*

A	B	C	D	E	F

Question 2

Parts 1-2 *Give the tight Θ bound (2 pts, 1 pt each)*

1	2

Part 3 *Write letter X, Y, or both as appropriate (5 pts, 1 pt each)*

A	B	C	D	E

Question 3

Part 1 *Write the letter(s) to fill in each spot in the code (6 pts, 1 pt each)*

1	2	3	4	5	6

Parts 2-3 *Give the tight Θ bound (2 pts, 1 pt each)* Part 4 *Write A or B (1 pt)*

2: Combine	3: Sort	4: Worst case

Question 4

*Write the **output** for the given Hasher (6 pts, 3 pts each)*

HasherA	HasherB

CSE 12 SP24 Final – Part 3 – Answer Sheet

Name: _____ PID: _____

Write all answers to Part 3 below. There are 22 total points.

Question 1

Part 1 *Array contents after poll() (2 pts)*

Part 2-4 *(min heap 2 pts, max heap 2 pts, poll 1 pt)*

2: Min Heap *(Write letter)*	**3: Max Heap** *(Write letter)*	**4: poll()** *(Write letter)*

Question 2

*Part 1-6 Write **BST**, **MIN**, **MAX**, or **NONE** (1.5 pts each)*

1	**2**	**3**	**4**	**5**	**6**

Question 3

Write sequence of letters (3 pts)

Next()

Question 4

Write sequence of letters (3 pts)

sortInPlace()

CSE 12 SP24 Final – Part 1

The following section is for Part 1 of the Final Exam. It corresponds to the Exam 1 midterm. We will take whichever grade is higher. If you wish to use your Exam 1 midterm grade, you can skip this part of the final exam.

Reference

Module java.base

Package java.util

Class ArrayList<E>

Type Parameters:

`E` - the type of elements in this list

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

Module java.base

Package java.util

Interface List<E>

`void`	**`add`**`(int index, E element)`	Inserts the specified element at the specified position in this list.
`boolean`	**`add`**`(E e)`	Appends the specified element to the end of this list.
`E`	**`get`**`(int index)`	Returns the element at the specified position in this list.
`int`	**`indexOf`**`(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E`	**`remove`**`(int index)`	Removes the element at the specified position in this list.
`boolean`	**`remove`**`(Object o)`	Removes the first occurrence of the specified element from this list, if it is present.
`int`	**`size`**`()`	Returns the number of elements in this list.

Question 1 (Part 1)

Consider implementing the interface `IncrementalPQueue`:

```
interface IncrementalPQueue<E> {
  int size();
  int getStartNumber();
  void maybeEnqueue(E element, int number);
  E dequeue();
}
```

An `IncrementalPQueue` is like a regular Queue, but each element has an associated number when being enqueued to the Queue. Only the elements with an associated number **exactly one greater than** the number of the element in the back of the queue can be enqueued (by maybeEnqueue). If the entry doesn't satisfy this condition, then we would simply not add the entry to the queue. For example, if the back of the queue has an element `("blue", 4)`, then `maybeEnqueue("green", 6)` would do nothing, but `maybeEnqueue("green", 5)` would add this entry to the back of the queue. During initialization of the Queue, we provide an initial start number, and the first element enqueued in the Queue can only have the associated number as `startNumber + 1`.

Dequeue for `IncrementalPQueue` should dequeue the elements like a regular Queue, and it should explicitly throw some exception if the queue is empty.

```
class ALIPQueue<E> implements IncrementalPQueue<E> {
  ArrayList<E> contents;
  int startNumber;

  public ALIPQueue(int startNumber) {
    this.contents = new ArrayList<E>();
    this.startNumber = startNumber;
  }

  public int size() { return this.contents.size(); }

  public int getStartNumber() { return this.startNumber; }

  public void maybeEnqueue(E element, int number) {
    // make sure to answer 1.1 directly in the answer sheet
  }

  public E dequeue() throws NoSuchElementException{
    // make sure to answer 1.2 directly in the answer sheet
  }
}
```

1. Provide an implementation of **`maybeEnqueue`** that matches the description above.
2. Provide an implementation of **`dequeue`** that matches the description above.
3. **True or false (Do not write T/F, write the complete word):** The `IncrementalPQueue` will **NOT** be sorted in ascending order based on the element E
4. **True or false (Do not write T/F, write the complete word):** Changing the type of the contents field from `ArrayList<E>` to `List<E>` would cause a compile error.

Question 2 (Part 1)

1. Consider this test case:

```
@Test
public void testStack() {
  List m = new MyStack();
  m.add(5);
  m.add(8);
  m.remove();
  m.add(7);
  ____1____ // make sure to answer directly in the answer sheet
  m.add(2);
  m.remove();
  int x = m.remove();
  assertEquals(x, 5);
}
```

Given that MyStack is a Stack, and assuming this test passes, Which **one or more** of the following are valid statements that can be filled in the blank? **Select all that apply**:

A. `m.add(7)`
B. `m.add(5)`
C. `m.remove()`
D. `m.add(3)`

2. Consider this test case:

```
@Test
public void testQueue() {
  List m = new MyQueue();
  m.add("z");
  m.add("x");
  m.add("y");
  m.remove();
  m.remove();
  m.add("w");
  int y = m.remove();
  ____2____ // make sure to answer directly in the answer sheet
}
```

Given that MyQueue is a Queue, write an **`assertEquals`** test to check that the letter that was removed and stored in variable **y** is the expected letter.

Question 3 (Part 1)

```
class Node<E> {
  E value;
  Node<E> next;
  public Node(E value, Node<E> next) {
    this.value = value;
    this.next = next;
  }
}
```

A `MusicStack` is similar to a standard stack. However, it introduces the functionality of `replaceValue()`, which finds the element with the specified value in the stack and replaces it with the new value. This operation is performed in addition to the typical stack operations such as `push()`, `pop()`, and `peek()`. Assume that all elements in the stack are unique. Note that the LinkedList will have a dummy node top as used in the PAs.

```
public class MusicStack<E> {
  Node<E> top;               // dummy node
  int size;

  public MusicStack() {
    this.top = new Node<E>(null, null);
    this.size = 0;
  }

  /**
  * Finds the element with the value val, and replaces it with the new value newVal
  * Throws an exception if the element with value val does not exist in the stack.
  */
  void replaceValue(E val, E newVal) throws IllegalArgumentException {
    // make sure to answer 3.1 directly in the answer sheet
  }

  // Pushes element elt onto the top of the stack
  void push(E elt) { // Assume implemented correctly }

  // Pops the top element from the stack and returns it
  E pop() { // Assume implemented correctly }

  // Peeks at the top element of the stack without removing it
  E peek() { // Assume implemented correctly }

  @Test
  public void testStack() {
    MusicStack<Integer> testStack = new MusicStack<>();
    // make sure to answer 3.2 directly in the answer sheet
  }
}
```

1. Provide an implementation of `replaceValue()` that matches the description above.

2. Write a test with a simple case to show that `replaceValue()` correctly replaces the value of an element in a stack. This test case **shouldn't** invoke an exception.

Question 4 (Part 1)

Consider this class and interface hierarchy:

```
interface ExamInterface {
    int getQuestions();
    int getAnswers();
    int calculateMeanScore();
}

class StarExam implements ExamInterface {
    /* questions about this class body below */
}

class CheckExam implements ExamInterface {
    /* questions about this class body below */
}
```

1. Given that the `StarExam` and `CheckExam` bodies compile, which of the following lines will **NOT** compile? **Select all that apply**:

- **A.** `ExamInterface obj1 = new StarExam();`
- **B.** `ExamInterface obj3 = new ExamInterface();`
- **C.** `StarExam obj2 = new CheckExam();`

2. **True or false (do <u>not</u> write T/F, write the complete word)**: Given that the `StarExam` and `CheckExam` bodies compile, the maximum number of methods in classes `StarExam` and `CheckExam` will be three.

3. Suppose we create an object:

```
ExamInterface obj = new StarExam();,
```

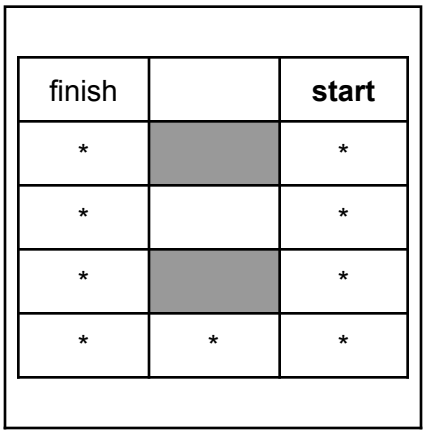
Which of the following statements is true? **Select all that apply**:

- **A.** `obj` can access methods in both `ExamInterface` and `StarExam`.
- **B.** `obj` can access all methods in `StarExam`.
- **C.** `obj` can only access the methods defined in `ExamInterface`.
- **D.** None of the above.

4. **True or false (do <u>not</u> write T/F, write the complete word)**: Given that the `StarExam` and `CheckExam` bodies compile, both the classes can only have **<u>exactly one</u>** method named `getQuestions()` with the `int` return type.

Question 5 (Part 1)

The maze below was solved with a **stack worklist/DFS**. The asterisks mark the path from the finish back to the start by following previous references.



For the above case, various **orders** have been chosen for adding the available neighbors to the worklist. You will answer which **order** matches with the above maze solution. **Choose from the answers below**. A reminder that on the page, **West** is left, **East** is right, **North** is up, and **South** is down. **There may be more than one correct order, select all that apply:**

A. **WESN** for West then East then South then North
B. **ENWS** for East then North then West then South
C. **ESWN** for East then South then West then North
D. **EWNS** for East then West then North then South

```
initialize wl to be a new empty worklist (stack _or_ queue)
add the start square to wl
mark the start as visited
while wl is not empty:
  let current = remove the first element from wl (pop or dequeue)
  if current is the finish square
    return current
  else
    for each neighbor of current that isn't a wall and isn't visited IN SOME ORDER
      mark the neighbor as visited
      set the previous of the neighbor to current
      add the neighbor to the worklist (push or enqueue)

if the loop ended, return null (no path found)
```

Question 6 (Part 1)

True or false (do not write T/F, write the complete word):
For the maze in question 5, if we use a **queue worklist/BFS** and **ANY** ordering of adding the neighbors to the worklist, we will get the **same path** *(not necessarily the path shown in Q5)* from the finish back to the start for all orderings.

CSE 12 SP24 Final – Part 2

The following section is for Part 2 of the Final Exam. It corresponds to the Exam 2 midterm. We will take whichever grade is higher. If you wish to use your Exam 2 midterm grade, you can skip this part of the final exam.

Question 1 (Part 2)

Consider this specification for partition:

> **int partition(int[] numbers)**
>
> Partition chooses a pivot value from the array at random. Then, change the array so that all elements smaller than or equal to the pivot appear in indices less than the index of the pivot value, and all elements larger than the pivot appear in indices greater than the index of the pivot value. Returns the final index of the pivot value, which may be different than its starting index. All elements in the input should appear at some index in the output.

Consider the following output array states and return values from partition. Which are possible valid outputs of partition for some input? We intentionally don't show the input. Note that each could come from a different implementation of and call to partition. For each letter, write **Yes** directly on the answer sheet if it corresponds to a valid result. If it's not valid, write **No**.

```
A. Resulting array: { 38, 23, 54, 49, 95, 76 }; Return value: 2
B. Resulting array: { 54, 49, 23, 76, 95, 38 }; Return value: 3
C. Resulting array: { 23, 38, 49, 95, 54, 76 }; Return value: 1
D. Resulting array: { 49, 23, 38, 54, 76, 95 }; Return value: 4
E. Resulting array: { 23, 38, 49, 54, 76, 95 }; Return value: 2
F. Resulting array: { 76, 38, 54, 23, 49, 95 }; Return value: 3
```

Question 2 (Part 2)

Recall that Θ represents a tight bound, O an upper bound, and Ω a lower bound.

1. Give a Θ bound for the number of steps the following program takes in terms of `n`, assuming `doSomeWork()` does a **linear** amount *(Θ(n))* of work.

```
for (int i = 1; i < n; i++) {
  int j = 3 * n;
  while (j > -2) {
    for (int k = 1; k < n; k = k++) {
      doSomeWork();
    }
    j--;
  }
}
```

2. Give a Θ bound for the number of steps the following program takes in terms of `n`, assuming `doSomeWork()` does a **constant** amount of work.

```
for (int i = 0; i < n; i++) {
  int j = 2 * n;
  while (i < j) {
    doSomeWork();
    j = j / 2;
  }
}
```

3. Consider the following pairs of functions. For each, indicate whether f *is* $O(g)$ and/or f *is* $\Omega(g)$. On the answer sheet as appropriate, write letter **X** if f *is* $O(g)$, **Y** if f *is* $\Omega(g)$, or both **XY** if f is both.

A. $f(x) = log(x)$ $\quad\quad$ $g(x) = log(log(x)) + log(x)$

B. $f(x) = 100! + 2^x$ $\quad\quad$ $g(x) = x^{100} + \frac{x^4+5}{x^2}$

C. $f(x) = x^{50x} + x^2$ $\quad\quad$ $g(x) = 300 * x! + log(x) * log(x)$

D. $f(x) = \frac{x(3-x)}{x^2}$ $\quad\quad$ $g(x) = \frac{(x+log(x))^3}{x^2}$

E. $f(x) = log(x) * log(x)$ $\quad\quad$ $g(x) = \sqrt{\frac{x(x+3)}{2}}$

This page is intentionally left blank. You may use it as scratch paper.

Question 3 (Part 2)

Consider the below **incomplete** MergeSort implementation taken from lecture:

```
public class MergeSort {
    /* Takes two arrays that are already sorted, and combines them into a single array
       in sorted order */
    public static int[] combine(int[] part1, int[] part2) {
        int index1 = 0, index2 = 0;
        int[] combined = new int[____1____];

        while(____2____) {
            if(____3____) {
                combined[____4____] = part1[index1];
                index1 += 1;
            }
            else {
                combined[____4____] = part2[index2];
                index2 += 1;
            }
        }

        while(____5____) {
            combined[index1 + index2] = part1[index1];
            index1 += 1;
        }

        while(____6____) {
            combined[index1 + index2] = part2[index2];
            index2 += 1;
        }

        return combined;
    }

    public static int[] sort(int[] arr) {
        if(arr.length <= 1) { return arr; }
        else {
            int[] part1 = Arrays.copyOfRange(arr, 0, arr.length / 2);
            int[] part2 = Arrays.copyOfRange(arr, arr.length / 2, arr.length);
            int[] sortedPart1 = sort(part1);
            int[] sortedPart2 = sort(part2);
            int[] sorted = combine(sortedPart1, sortedPart2);
            return sorted;
        }
    }
}
```

1. Choose from the pieces of code below to complete the merge method, `combine()`. Each incomplete line only requires **one** of the below pieces of code. For each incomplete line, write the letter corresponding to the line of code you chose.

   A. `index1`
   B. `index2`
   C. `index1 + index2`
   D. `(index1 + index2) / 2`
   E. `index1 < part1.length`
   F. `index1 < combined.length`
   G. `index2 < part2.length`
   H. `index2 < combined.length`
   I. `part1.length`
   J. `part2.length`
   K. `part1.length + part2.length`
   L. `part1`
   M. `part2`
   N. `combined`
   O. `index1 < combined.length && index2 < combined.length`
   P. `index1 < combined.length || index2 < combined.length`
   Q. `index1 < part1.length && index2 < part2.length`
   R. `index1 < part1.length || index2 < part2.length`
   S. `part1[index1] < combined[index2]`
   T. `part1[index2] < combined[index1]`
   U. `part1[index1] < part2[index2]`
   V. `part1[index1] > part2[index2]`

2. Give a Θ bound (a tight bound) for the number of steps the `combine()` method takes in terms of **`n`** `(part1.length)` and **`m`** `(part2.length)`.

3. Give a Θ bound (a tight bound) for the number of steps the `sort()` method takes in terms of **`n`** `(arr.length)`.

4. Consider the MergeSort and QuickSort algorithms that you learned in the lecture. In terms of worst-case time complexity, which algorithm performs better? Write **A** for MergeSort and **B** for QuickSort.

Question 4 (Part 2)

Consider this implementation of a hash table based on one from the lecture. Bolded portions are especially relevant.

```
interface Hasher<K> { int hash(K key); }

class HashTable<K,V> {
  class Entry {
    K k; V v;
    public Entry(K k, V v) { this.k = k; this.v = v; }
  }
  List<Entry>[] buckets; // An array of Lists of Entries
  int size;
  Hasher<K> hasher;

  public HashTable(Hasher<K> h, int startCapacity) {
    this.size = 0;
    this.hasher = h;
    this.buckets = (List<Entry>[])(new List[startCapacity]);
  }

  public V get(K k) {
    int hashCode = this.hasher.hash(k);
    int index = hashCode % this.buckets.length;
    if(this.buckets[index] == null) {
      return null;
    }
    else {
      for(Entry e: this.buckets[index]) {
        if(e.k.equals(k)) { return e.v; }
      }
      return null;
    }
  }

  public void set(K k, V v) {
    int hashCode = this.hasher.hash(k);
    int index = hashCode % this.buckets.length;
    if(this.buckets[index] == null) {
      this.buckets[index] = new ArrayList<Entry>();
      this.buckets[index].add(new Entry(k, v));
    }
    else {
      for(Entry e: this.buckets[index]) {
        if(e.k.equals(k)) { e.v = v; return; }
      }
      this.buckets[index].add(new Entry(k, v));
    }
    this.size += 1;
  }
}
```

Consider these implementations of the `Hasher` interface:

```
class HasherA implements Hasher<String> {
  public int hash(String s) { return (s.length() * 3) % 5; }
}

class HasherB implements Hasher<String> {
  int num;
  public HasherB() { this.num = 0; }
  public int hash(String s) {
       int hashValue = num + 1;
       int i = s.length() - 1;
       if (Character.isUpperCase(s.charAt(i))) {
          num += 2;
       }
       return hashValue;
  }
}
```

Recall that `s.charAt(i)` retrieves the character at index `i` of string `s`. `Character.isUpperCase(c)` returns true if character `c` is an uppercase character and false otherwise.

Consider this sequence of operations:

```
1   HashTable<String, Integer> ht = new HashTable<>(???, 6);
2   ht.set("pink", 200);
3   System.out.println(ht.size);
4   ht.set("BLUE", 600);
5   ht.set("red", 300);
6   System.out.println(ht.size);
7   System.out.println(ht.get("BLUE"));
8   ht.set("Brown", 400);
9   System.out.println(ht.size);
10  System.out.println(ht.get("Brown"));
11  ht.set("red", 600);
12  System.out.println(ht.size);
```

Consider using each of the two Hasher implementations to replace the **???** above by filling it in with either `new HasherA()` or `new HasherB()`.

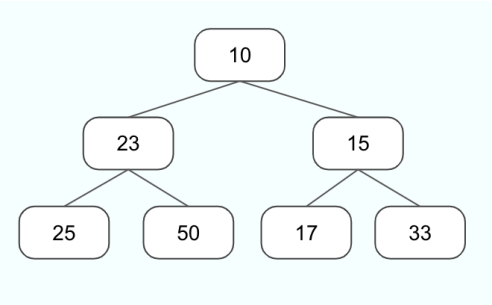
For each Hasher, write what would print directly in the answer sheet (6 lines of output each)

CSE 12 SP24 Final – Part 3

The following section is for Part 3 of the Final Exam. It corresponds to the Exam 3 midterm. We will take whichever grade is higher. If you wish to use your Exam 3 midterm grade, you can skip this part of the final exam.

Question 1 (Part 3)

1. Consider the following array representing a complete tree in **min-heap order**, along with a diagram illustrating its binary tree structure: `10, 23, 15, 25, 50, 17, 33`



Assuming we use the typical strategy of moving the last element to the root and then using **bubbleDown()**, what is the resulting tree after a call to **poll()**? Write the **array's** contents directly in the answer sheet.

2. Assuming we use the typical strategy of adding an element at the end of the array and using **bubbleUp()**. Starting with an empty **min heap**, insert the following **six** elements in the given order: *30, 8, 15, 2, 3, 24*. Write just **the letter** of the array that correctly represents the final **min heap**.

A. `2 3 15 8 30 24`
B. `2 15 3 24 8 30`
C. `2 3 15 24 8 30`
D. `2 15 3 8 30 24`
E. `2 3 15 30 8 24`
F. `2 15 3 8 24 30`

3. Consider the following array representing a complete tree in **max-heap order**: `14, 8, 10, 5, 1`

Insert the element *7* into the array. Write just **the letter** of the array that correctly represents the final **max heap**.

A. `14 10 8 5 1 7`
B. `14 10 8 7 5 1`
C. `14 10 7 8 1 5`
D. `14 8 10 5 7 1`
E. `14 8 10 7 5 1`
F. `14 8 10 5 1 7`

4. Consider the final **max heap** from the previous part (Question 1.3). Assume we make **three** consecutive calls to **poll()**. What does the 3rd subsequent call to **poll()** return?

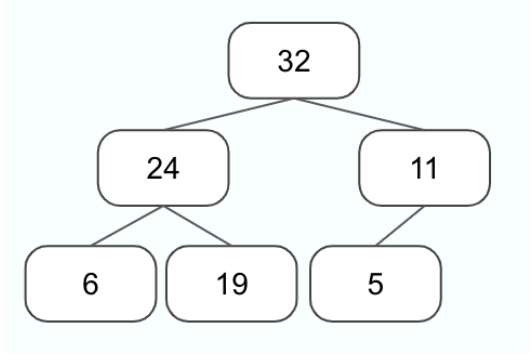
A. `null`
B. `7`
C. `10`
D. `void`
E. `14`
F. `5`
G. `8`

Question 2 (Part 3)

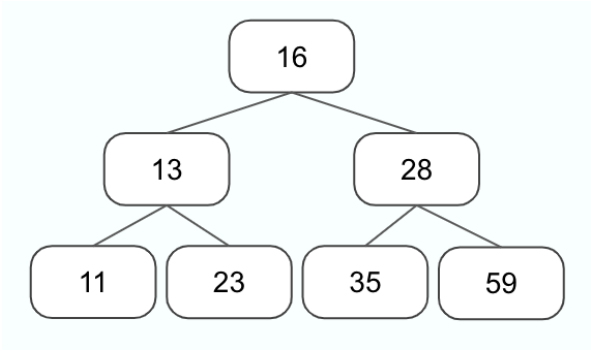
Identify each of the following trees by writing **BST**, **MIN**, **MAX**, or **NONE** in the answer sheet

Answer choices:
BST - Binary Search Tree (BST) - if the example represents a valid binary search tree.
MIN - Min Heap - if the example represents a valid min heap.
MAX - Max Heap - if the example represents a valid max heap.
NONE - None - if the example does not represent a valid BST, valid min heap or a valid max heap.

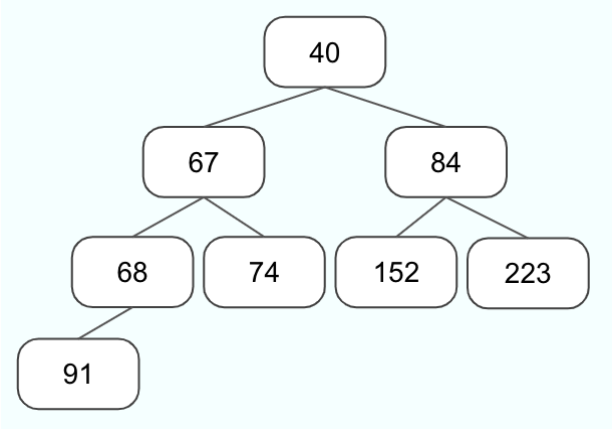
1.



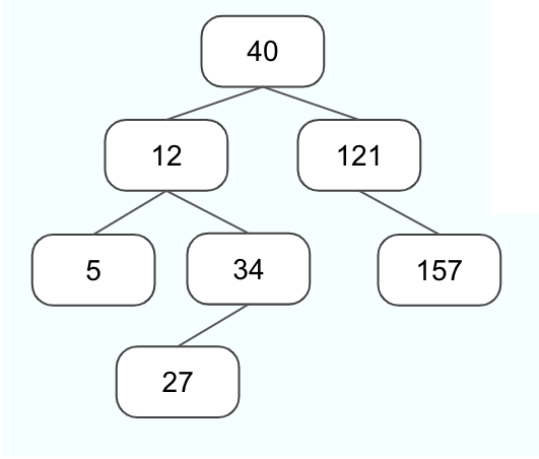
2.



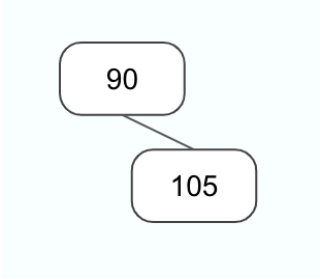
3.



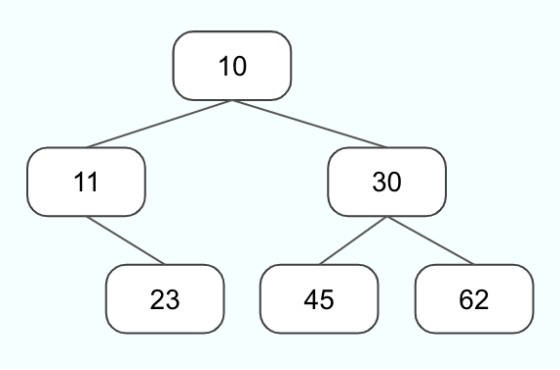
4.



5.



6.



This page is intentionally left blank. You may use it as scratch paper.

Question 3 (Part 3)

Consider the following `GList` class extended to support iteration in the **reverse order**.

```
class Node {
    Integer data;
    Node next;
    Node prev;
    Node(Integer data) { this.data = data; }
}

class GList implements Iterable<Integer> {
    class GListIterator implements Iterator<Integer> {
        Node currentNode;

        public GListIterator(Node tail) {
            this.currentNode = tail;
        }

        public Integer next() {
            /* Implement this */
        }

        public boolean hasNext() {
            return this.currentNode != null;
        }
    }

    Node head;
    Node tail;

    public GList() {
        this.head = null;
        this.tail = null;
    }

    public void add(Integer data) {
        Node newNode = new Node(data);
        if (tail == null) {
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
        }
    }

    public Iterator<Integer> iterator() {
        return new GListIterator(tail);
    }

    /* Other methods not shown (remove, etc.) */
}
```

Choose a sequence of letters from below that constructs a working implementation for the `next()` method that returns the elements of the list in order. Give your answer as a sequence of letters. You can ignore error cases of iterating past the end of the list.

A. `return currentNode.data;`
B. `Integer answer = currentNode.data;`
C. `Node answer = currentNode;`
D. `return answer;`
E. `currentNode = null;`
F. `currentNode = currentNode.next;`
G. `currentNode.data = currentNode.prev.data;`
H. `currentNode = currentNode.prev;`
I. `Integer answer = currentNode.prev.data;`

Question 4 (Part 3)

Using a Priority Queue, implement the method `sortInPlace()`, which sorts the elements in the array in-place in **descending** order.

Example: Given input array [5, 10, 3, 2, 6, 1], calling `sortInPlace()` should result in the input array being modified to [10, 6, 5, 3, 2, 1].

```
public static void sortInPlace(int [] array) {
    // Implement this!
}
```

Choose a sequence of letters from below that constructs a working implementation for the `sortInPlace()` method. Note that the sequence of letters must be in the **correct order** to receive credit:

A. `for (int i = 0; i < array.length; i++) pq.add(i);`
B. `for (int num : array) pq.add(array[num]);`
C. `for (int num : array) pq.add(num);`
D. `PriorityQueue<Integer> pq = new PriorityQueue<>(Integer::compare);`
E. `PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder(Integer::compare));`
F. `for (int i = 0; i < array.length; i++) array[i] = pq.poll();`
G. `for (int i = array.length - 1; i >= 0; i--) array[i] = pq.poll();`
H. `for (int i = array.length - 1; i >= 0; i--) array[i] = pq.peek();`
I. `for (int i = 0; i < array.length; i++) array[i] = pq.peek();`

➢ Recall that using `Integer::compare` results in Min Heap and using `Collections.reverseOrder(Integer::compare)` results in a Max Heap.
➢ There may be more than one correct sequence of letters. Only write one sequence of letters in the answer sheet.

Scratch Paper