



# University of Maryland College Park

## Department of Computer Science

### CMSC132 Spring 2025

### Exam #2

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

**KEY**

STUDENT ID (e.g., 123456789):

#### Instructions

- Please print your answers and use a pencil.
- This exam is a closed-book, closed-notes exam with a duration of 50 minutes and 100 total points.
- **Do not remove the exam's staple.** Removing it will interfere with the scanning process (even if you staple the exam again).
- Write your directory id (e.g., terps1, not UID) at the bottom of pages with **DirectoryId**.
- Provide answers in the rectangular areas.
- Do not remove any exam pages. Even if you don't use the extra pages for scratch work, return them with the rest of the exam.
- Your code must be efficient and as short as possible.
- If you continue a problem on the extra page(s) provided, make a note on the particular problem.
- You don't need to use meaningful variable names; however, we expect good indentation.
- **You must write your name and id at this point (we will not wait for you after time is up).**
- You must stop writing once time is up.

#### Grader Use Only

#1	Part #1 (Short Answer)	22	
#2	Part #2 (Short answer)	21	
#3	Part #3 (Code 2)	57	
<b>Total</b>	Total	100	

**Part #1 (Short answer – 2 points each)**

1. **Throwable** is the base class for all errors and exceptions in Java.
2. **Enum** are special data types in Java used to define collections of constants.
3. **Annotations** are a form of metadata that provide additional information about the program to the compiler or runtime environment.
4. Binary search has a pre-requisite condition of the data being **sorted** before it can search.
5. Bubble sort runs in  $O(n)$  time in the **best** case.
6. (12 pts @ 2 each) Assume the class **Chips** extends **JunkFood** which extends **Food**. Further, assume that each of the 3 classes has a default constructor that takes no argument. For part a to f, assume the only code that appears in the main method is the given code in the question. Simply circle C if it will compile, CE if it will compile but throw an exception, and NC if it will not even compile.

a.	<pre>ArrayList food = new ArrayList(); food.add("food");</pre> <p style="text-align: center;"><b>C</b>      CE      NC</p>
b.	<pre>ArrayList &lt;Food&gt; ol = new ArrayList&lt;Food&gt;(); ArrayList &lt;? super Chips&gt; list = ol; list.add(new Chips());</pre> <p style="text-align: center;"><b>C</b>      CE      NC</p>
c.	<pre>ArrayList &lt;T extends Chips&gt; list = new ArrayList&lt;Chips&gt;();</pre> <p style="text-align: center;">C      CE      <b>NC</b></p>
d.	<pre>ArrayList &lt;Chips&gt; cl = new ArrayList&lt;Chips&gt;(); cl.add(new Chips()); ArrayList &lt;? extends Food&gt; fl = cl; Food c = fl.get(0);</pre> <p style="text-align: center;"><b>C</b>      CE      NC</p>
e.	<pre>ArrayList&lt;?&gt; list = new ArrayList&lt;Chips&gt;(); list.add(null); Object o = list.get(0);</pre> <p style="text-align: center;"><b>C</b>      CE      NC</p>
f.	<pre>ArrayList &lt;JunkFood&gt; jl = new ArrayList &lt;JunkFood&gt; (); jl.add(new Chips()); jl.add((JunkFood)new Food());</pre> <p style="text-align: center;">C      <b>CE</b>      NC</p>

**Part #2 (Short answer – 3 points each)**

7. List the following Big O expressions in order of asymptotic complexity (lowest complexity first).

$O(5n\log(n))$

$O(n^3)$

$O(20\log(n))$

$O(10n^2)$

$O(2^n)$

$O(20\log(n)), O(5n\log(n)), O(10n^2), O(n^3), O(2^n)$

8. Explain why selection sort is still  $O(n^2)$  in the best case when the data is already sorted in an array. No more than 2 sentences.

It will still do the same number of comparisons as any other case.

9. Indicate the algorithm complexity of the following expression using the best one-term bound:

$$70n + 2n\log(n) + n^3$$

$O(n^3)$

**Consider the following scenario for question 10 and 11:**

- Algorithm A has a time complexity of  $O(n)$
- Algorithm B has a time complexity of  $O(n^2)$
- There is a third algorithm that calls Algorithm A and Algorithm B in sequence, one after the other.

10. Does the overall time complexity of the third algorithm change depending on whether Algorithm A is called before or after Algorithm B? Explain in no more than 2 sentences.

No, because the time complexity of the third algorithm is just the sum of the two and addition is commutative.

11. What is the overall time complexity of the third algorithm using the best bound (only if it is possible to be a one term bound, should that be the answer)?

$O(n^2)$

12. Assume the following code from your project 3:

```
int[] nums = {5, 1, 3, 2, 4, 6, 7, 8};
int k = 4;
int[] result = slidingWindowMax(nums, k);
```

Fill in the **element values** after the code runs:

5	4	6	7	8
---	---	---	---	---

13. Assume the following array:

51	12	37	81	4
----	----	----	----	---

What is the content of the array after the second to last swap occurs in bubble sort?

12	4	37	51	81
----	---	----	----	----

### Part #3 (Code)

You will finish 4 methods for the code below and answer some questions about the code. The code implements the Stack ADT using a circular singly linked list data structure. In this circular linked list design:

- The last node (i.e. the tail) points back to the first node (i.e. head) instead of null.
- There is no explicit head field for the circular linked list as the head can be found via tail.next. As you will use the linked list to function as a stack, the top of your stack should be tail.next.
- The linked list (i.e. your stack) is empty when tail is null.
- When there is only one node, tail.next points back to tail (circularly).

For full credit, your implementation must ensure that all 4 methods: push, pop, hasNext method of the iterator, and next method of the iterator, operate in  $O(1)$  time complexity. No additional fields, methods (i.e. no helper method or constructors), or using anything from the Java library. Assume no null value for the data.

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Stack<T> implements Iterable<T> {
    private Node tail; // Tail reference (tail.next is head)

    /**
     * Node class representing each element in the stack.
     * Each node holds a data item and a reference to the next node.
     */
    private class Node {
        T data;
        Node next;

        Node(T data) {
            this.data = data;
        }
    }

    /**
     * Custom exception for empty stack condition.
     */
    public static class StackEmptyException extends RuntimeException {
        public StackEmptyException(String message) {
            super(message);
        }
    }

    public void push(T item) {
        //YOU WILL CODE THIS
    }

    public T pop() {
        //YOU WILL CODE THIS
    }
}
```

```

/**
 * Returns an iterator for the stack.
 * The iterator allows iteration through the stack from top to bottom.
 * Time Complexity: O(1) per iteration
 * @return an iterator for the stack
 */
@Override
public Iterator<T> iterator() {
    return new StackIterator();
}

private class StackIterator implements Iterator<T> {
    private Node current = (tail == null) ? null : tail.next;
    private boolean firstPass = (tail != null);

    @Override
    public boolean hasNext() {
        //YOU WILL CODE THIS
    }

    @Override
    public T next() {
        //YOU WILL CODE THIS
    }
}

public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(10);
    stack.push(20);
    stack.push(30);

    // Iterate over the stack
    for (int item : stack) {
        System.out.print(item + " "); // Output: 30 20 10
    }
    System.out.println();

    System.out.println(stack.pop()); // 30
    System.out.println(stack.pop()); // 20
    System.out.println(stack.pop()); // 10

    try {
        System.out.println(stack.pop());
    } catch (StackEmptyException e) {
        System.out.println("Caught exception: " + e.getMessage());
    }
}
}

```

### OUTPUT

30 20 10

30

20

10

Caught exception: Stack is empty

Directory ID:

14. (2pts) *Circle one:* **StackEmptyException** from the code above is an example of a(n):

- a) Check Exception
- b) Unchecked Exception**
- c) An Unbounded Exception
- d) A Bounded exception
- e) None of the above

15. (2pts) *Circle one:* **StackIterator** from the code above is an example of a(n):

- a) Lambda Expression
- b) Top Level Class
- c) Inner Class**
- d) Local Class
- e) Anonymous Class
- f) Static Nested Class

16. (14 pts) Code the **push** method. Remember that the top of the stack is tail.next. Your code should handle both the case where the stack is empty before the push, and the case where there are already node(s) in the stack before the push. Time Complexity must be  $O(1)$ .

```
public void push(T item) {  
  
    Node newNode = new Node(item);  
    if (tail == null) {  
        tail = newNode;  
        tail.next = tail; // Circular link to itself when stack is empty  
    } else {  
        newNode.next = tail.next;  
        tail.next = newNode;  
    }  
}
```

17. (16 pts) Code the **pop** method. Remember that the top of the stack is `tail.next`. If the stack is empty (`tail` is `null`), a **StackEmptyException** is thrown with the message: `Stack is empty`. Your code should handle both the case where the stack becomes empty after the pop, and the case where there will still be node(s) left in the stack after the pop. Time Complexity must be  $O(1)$ .

```
public T pop() {  
    if (tail == null) {  
        throw new StackEmptyException("Stack is empty");  
    }  
  
    Node head = tail.next; // The top element is tail.next  
    if (head == tail) { // Only one element in stack  
        tail = null; // Stack becomes empty after pop  
    } else {  
        tail.next = head.next; // Remove the top node  
    }  
    return head.data;  
}
```

General note about Iterator implementation: In theory, the user should first call **hasNext()** method once, and if **true**, they should call **next()** method once to get the next element. However, your implementation should be robust in that sense that multiple calls to **hasNext()** should not lead to skipping any elements, and calls to **next()** without calling **hasNext()** should return the next element as long as there are elements left.

18. (8 pts) Code the **hasNext()** method of the iterator. It should return **true** if there are more elements in the stack to iterate, and **false** otherwise. Remember that you can use the fields **current** and/or **firstPass** (already initialized) in your code.

```
public boolean hasNext() {  
    return firstPass ; //This only works depending on how next changes firstPass  
}
```

19. (15 pts) Code the **next()** method of the iterator. It should throw a **NoSuchElementException** (with no string message) if **next** is called and there are no elements left to iterate over. Otherwise, it should return the data of the current node of the stack. Remember that you can use and make changes to the fields **current** and/or **firstPass** in your code. You can also call **hasNext()** if you want.

```
public T next() {  
  
    if (!hasNext()) {  
        throw new NoSuchElementException();  
    }  
  
    T data = current.data;  
    current = current.next;  
    if (current == tail.next) { // If we've looped back, stop iteration  
        firstPass = false;  
    }  
    return data;  
}
```