2024 Fall Data Structures – Midterm Exam (Example)

Name:

NYU NetID:

A. Please read the following questions and respond with either T or F. (**Correct answers earn +2 points each, incorrect responses result in -1 point, and if left unanswered, no points will be awarded.**)

1. There are 8 primitive types in JAVA. (   )
2. JAVA's reference types are allocated in the Stack memory (   )
3. JAVA employs pass-by-reference. (   )
4. When a recursive function is called, a single function allocated in memory is repeatedly utilized. (   )
5. Arrays allow dynamic resizing. (   )
6. The time complexity (Big-O) of the remove() operation of ArrayList is O(1). (   )
7. A Stack flows the Last In, First Out (LIFO) principle. (   )
8. Linked Lists are more efficient than arrays for insertion and deletion operations. (   )
9. The time complexity (Big-O, the worse case) of insertion sort is O(nlogn). (   )
10. Complete binary tree means a binary tree that is fully occupied. (   )

B. Please read the following questions and options, then check the one that corresponds to the correct answer.

```
void function1(int m, int a[], int n, int b[], int c[]) {
    for (int i=0; i<m; i++) {
        c[i] = a[i];
    }
    for (int j=0; j<n; j++) {
        for (i = m+j-1; i>=0 ; i--) {
            c[i+1] = c[i];
        }
        c[i+1] = item;
    }
}
```

1. What is the time complexity (Big-O, worst case) of the above function when m > n? [3 points]

   ① $O(m+n)$
   ② $O(mn)$
   ③ $O(m)$
   ④ $O(n)$

2. Choose all operations with **the time complexity (Big-O) of n** (e.g., O(n)), where n is the number of elements. Note that **expansion is not considered** in the case of ArrayList. Additionally, consider only implementations using a **singly-linked list** with one head node for LinkedList. [3 points]

   ① Add(index, x) operation of ArrayList
   ② Append(x) operation of LinkedList
   ③ Remove(index) operation of LinkedList
   ④ Get(index) operation of ArrayList
   ⑤ Get(index) operation of LinkedList

3. The following code is the partial implementation of **enqueue()** operation of ArrayQueue. Consider that, to implement ArrayQueue, we view an array as a **circle**. Accordingly, choose the most appropriate code to fill in the blank spaces ([#1] and [#2]). [3 points]

```
public int enqueue(E newItem) {
    if (isFull()){
        return -1;
    }
    else {
        [#1];
        [#2];
        ++numItems;
        return 0;
    }
}
```

① [#1] tail = (tail-1) % queue.length , [#2] queue[tail-1] = newItem;
② [#1] tail = (tail-1) % queue.length , [#2] queue[tail+1] = newItem;
③ [#1] tail = (tail-1) % queue.length , [#2] queue[tail] = newItem;
④ [#1] tail = (tail+1) % queue.length , [#2] queue[tail+1] = newItem;
⑤ [#1] tail = (tail+1) % queue.length , [#2] queue[tail] = newItem;

4. The following code is the implementation of percolateDown() of Heap. Choose the most appropriate code to fill in the blank space in the following code

```
private void percolateDown(int i) {
    int child = 2 * i + 1;
    int rightChild = 2 * i + 2;
    if (child <= numItems - 1) {
        if (/* blank space */)
            child = rightChild;
        if (A[i].compareTo(A[child]) < 0) {
            E tmp = A[i];
            A[i] = A[child];
            A[child] = tmp;
            percolateDown(child);
        }
    }
}
```

① rightChild <= numItems - 1 && A[child].compareTo(A[rightChild]) < 0

② rightChild <= numItems - 1 && A[child].compareTo(A[rightChild]) > 0

③ rightChild < numItems - 1 && A[child].compareTo(A[rightChild]) > 0

④ rightChild < numItems - 2 && A[child].compareTo(A[rightChild]) < 0

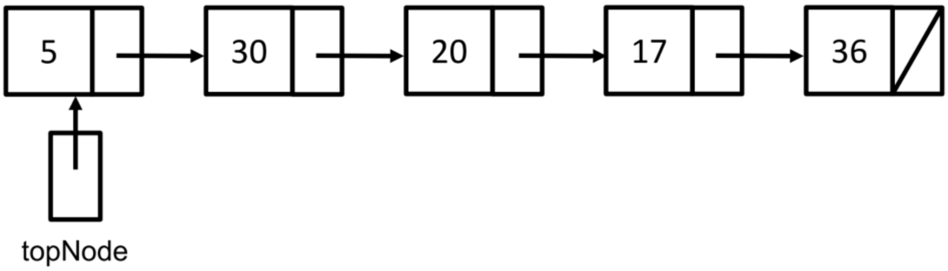
C. Please read the following question and provide your answer.

1. Write the worst case time complexity (Big-O) for the following algorithms
   ① Bubble sort:
   ② Insertion sort:
   ③ Merge sort:
   ④ Selection sort:
   ⑤ Quick sort:

2. Write the best case time complexity for the following algorithms
   ① Bubble sort:
   ② Quick sort:
   ③ Insertion sort:
   ④ Merge sort:
   ⑤ Selection sort:

D. Please read the following question and fill in the blank space with the appropriate code. (Node: Strict adherence to JAVA syntax is not necessary).

1. The below figure illustrates a stack implemented by a LinkedList.



The following code is the implementation of both push() and pop() operations of LinkedStack. Remind that **topNode** refers to the top node of the stack. Accordingly, write the most appropriate code to fill in the blank spaces ([#1] and [#2]). [6 points]

```
public int push(E newItem) {
    topNode = new Node<>(newItem, topNode);
    return 0;
}

public E pop() {
    if (isEmpty( )) return null;
    else {
        [#1];
        [#2];
        return temp.item;
    }
}
```

[Instruction] Write the code in the Java style, and **while strict adherence to syntax is not necessary**, ensure that **the logical flow of the written code is appropriate**. Incorrect logical flow is considered as the incorrect answer.

[Write your code]

2. The below Java code shows the **remove()** function that removes an element from a LinkedList with a dummy head. The remove(index) function deletes the element at the specified index in the LinkedList.

```java
public class LinkedList implements ListInterface{
    private Node head;
    private int numItems;

    public LinkedList(){
        head = new Node(null, null);
    }

    ...

    public void remove(int index){
        if (index >= 0 && index <= numItems-1){
            Node prevNode = getNode(index - 1);
            prevNode.next = prevNode.next.next;
            numItems--;
        }
        else{ /* error handing */}
    }
    ...
```

Now, by adding an additional parameter k to the remove() function, calling it as remove(index, k), we aim to delete a contiguous sequence of k elements (including the element at index) starting from the specified index. If fewer than k elements are remaining starting from the index, we delete only up to the last node. Accordingly, please write the appropriate code in the **blank space** below. [12 points]

```java
public class LinkedList implements ListInterface{
    private Node head;
    private int numItems;

    public LinkedList(){
        head = new Node(null, null);
    }

    ...

    public void remove(int index, int k){
        if (index >= 0 && index <= numItems-1){
            /* --- blank space --- */
        }
        else{ /* error handing */}
    }
    ...
```

[Instruction] Write the code in the Java style, and while **strict adherence to syntax is not necessary**, ensure that **the logical flow of the written code is appropriate**. Incorrect logical flow is considered as the incorrect answer.

[Write your answer code below]

E. Please carefully read the following problem and draw an appropriate diagram. In the case of using a LinkedList, make sure to accurately depict the connections between each node.

1. In the case of using the pop() and push() operations implemented in the ArrayStack, please draw the element map of the ArrayStack after executing the program as follows.

top

```
CustomArrayStack<Integer> stack = new CustomArrayStack<>();

stack.push(1);
stack.push(3);
stack.pop();
stack.push(1);
stack.push(15);
stack.pop();
stack.pop();
stack.push(1);
stack.push(32);
stack.push(9);
stack.pop();
```

2. When the following JAVA code is executed, draw the memory allocation map immediately after the execution of the last code. **Clearly indicate** whether variables are allocated in the **Stack** or **Heap** memory, and for **reference variables**, provide a **clear connection** to the object they are **referring**. Also, if something is allocated in **consecutive memory locations**, should mark it. Note that Circle is a JAVA class that includes an integer variable called radius. [6 points]

```
Circle [] circles = new Circle[5];

for (int i=0; i<5; i++){
    circles[i] = new Circle(i*5);
}

circles[0] = circles[2];
circles[0].radius = 1;
circles[1] = circles[0];
circles[2].radius = 2;
```

Stack memory	**Heap memory**