Name: _Answer Key_____ Section: _____

KEY – CSC 210: Data Structures – KEY

Fall 2025 Midterm Exam
All Sections

Self-scheduled exam, Oct. 17-19 2025

- You have 75 minutes to complete this exam, unless you have a documented accommodation for additional time.
- The exam is closed book. You are allowed one 8.5”x11” page of notes (double-sided is ok).
- If you think there is a problem with the exam, do your best to interpret what is requested, and document your choice with a note.
- You may not communicate or consult about the exam with anyone other than the professors until after the weekend is over.
- If you are unable to make progress on any part of the exam, tell me what you know and what you tried: describe your thought process. Partial credit may be possible.

Please copy, sign, and date the statement below: “I have read the instructions above and certify that I have adhered to the Smith Honor Code while completing this exam.”

_____
_
_____
_
_____
_
_____
_
_____
_

Signature: _____

Abstract Datatypes (12 points)

We wish to create two ADTs, one for a linked list and one for an array. Below right is a table with several operations from Java's **List** interface shown. (Note that this interface is implemented by both **ArrayList** and **LinkedList**.) For each empty cell, fill in **Yes** if the operation can be supported by the corresponding underlying data structure without looping or recursion, and **No** if it cannot. In other words, can the operation always be completed in a constant number of steps? All operations should be interpreted as they are in Java's **List** interface. Assume that the array is allocated with extra space for growth at the end.

Operation	ArrayADT	ListADT
set(index,value)	yes	no
add(index,value)	no	no
add(value)	yes	yes
contains(value)	no	no
remove(index)	no	no
get(index)	yes	no

Iterators (10 points)

Predict the printed output of the following code snippets, assuming that `list` is a `LinkedList<String>`, and for each individual part below it begins with initial contents `[R, S, T]`. The variable `iter` has type `ListIterator<String>`. If the code throws an exception, give any output that would be produced first, and then write **<Exception>**. You don't need to specify the type of exception.

```
// Part A
iter = list.listIterator();
iter.next();
```

```
System.out.println(iter.next());
System.out.println(iter.nextIndex());
```

Output:

```
S
2
```

```
// Part B
iter = list.listIterator(3);
iter.next();
System.out.println(iter.next());
System.out.println(iter.nextIndex());
```

Output:

```
<Exception>
```

```
// Part C
iter = list.listIterator();
iter.next();
iter.next();
iter.remove();
System.out.println(iter.next());
System.out.println(iter.nextIndex());
```

Output:

```
T
2
```

```
// Part D
iter = list.listIterator(3);
iter.previous();
iter.previous();
System.out.println(iter.previous());
System.out.println(iter.nextIndex())
```

Output:

```
R
```

0

```
// Part E
iter = list.listIterator();
iter.next();
iter.add("D");
System.out.println(iter.next());
System.out.println(iter.nextIndex());
```

Output:

S
3

Stacks & Queues (12 pts)

1. Your friend has made a Java puzzle for you: a class called *nextInt* that is either a stack or a queue. They want you to guess which it is. They tell you it has two methods: *give* and *take*. Can you guess whether it's a stack or queue based on its behavior? Please provide a short justification (a few words are fine) for your choice. **Assume parts a and b are independent of each other.**
   a. You call nextInt.give(2), then nextInt.take(), which returns 8.
      More likely to be:

      ☐ Stack X Queue

      Why?

      If it was a stack, the take operation would have returned 2 (LIFO). So it must be a queue.

   b. You call nextInt.give(2) and then nextInt.give(10). Then you callnextInt.take(), which returns 10, followed by another nextInt.take(), which returns 2. Then you call nextInt.isEmpty(), which returns true.
      More likely to be:

      X Stack ☐ Queue

      Why?

      The order of the elements is reversed when you take them off. This is LIFO behavior,

which is characteristic of stacks.

2. You are developing your own implementation of a **stack using the Linked List class you developed for A3.** Choose the method that you would call to implement each of these core functions, and provide a short justification.

   Push:

   addFirst addLast removeFirst removeLast

   Pop:

   addFirst addLast removeFirst removeLast

   Why? *Stacks add and remove from the same point. (We could also have chosen addLast and removeLast).*

3. Would it be more **efficient** (in terms of computational complexity, or run time) to use a base-type array T[ ] or a Linked List for your **stack** implementation? Please provide a short justification for your choice. Assume your Linked List class tracks head but not tail.

   ☐ Array ☐ Linked List X No major difference in run time

   Why? *We have efficient implementations using either.*

4. You are developing your own implementation of a **queue using the Linked List class you developed for A3.** Choose the method that you would call to implement each of these core functions.

   Add:

   addFirst addLast removeFirst removeLast

   Remove:

addFirst addLast removeFirst removeLast

Why? *Queues add and remove from opposite ends. (We could also have chosen addFirst and removeLast.)*

5. Would it be more **efficient** (in terms of computational complexity, or run time) to use a base-type array T[ ] or a Linked List for your **queue** implementation? Please provide a short justification for your choice. Assume your Linked List class tracks head but not tail.

   X Array ☐ Linked List ☐ No major difference in run time

   Why? *Without direct access to the tail in LinkedList, operations on it would require traversing the whole list.*

Arrays (10 points)

Suppose we have a DynamicArray class defined like the one from assignment A1, which implements the interface below:

```
public interface DynamicArrayADT<T> {
    public T set(int index, T value);
    public T get(int index);
    public int size();
    public void add(int index, T value);
    public void add(T value);
    public T remove(int index);
}
```

Assuming that the class is backed by a standard Java array, and we have an instance **`darr`** whose starting configuration looks like this for each question below:

C	S	C	2	1	0

From the bank of possible answers below, choose the configuration (one of the options P through V) that would result from each of the operations shown, or say “Other” if the correct configuration is not among the options shown. Remember, each question part starts from the configuration above.

P:

C	S	C	3	2	1	0

Q:

C	S	C	2	1	0

R:

C	C	C	2	1	0

S:

C	C	C	3	1	0

T:

C	S	3	2	1	0

U:

C	C	S	C	2	1	0

V:

C	S	C	C	2	1	0

Questions:

```
        // Part A
        darr.set(3,'3');
```

Other

```
        // Part B
        darr.set(1,darr.get(2));
```

R

```
        // Part C
        darr.remove(2);
        darr.set(2,'3');
```

Other

```
        // Part D
        darr.add(2,'3');
```

Other

```
        // Part E
        darr.add(3,'3');
        darr.set(3,'C');
```

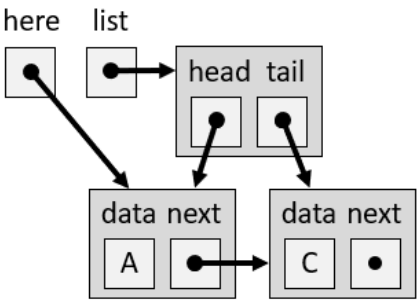
V

Lists (12 points)

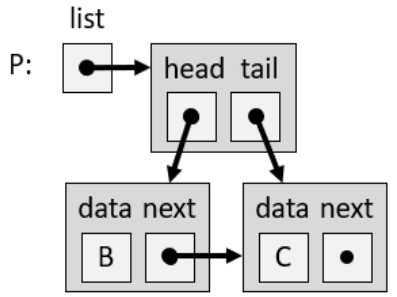
Suppose we have a List class defined similar to the one from assignment A3, which implements the interface below:

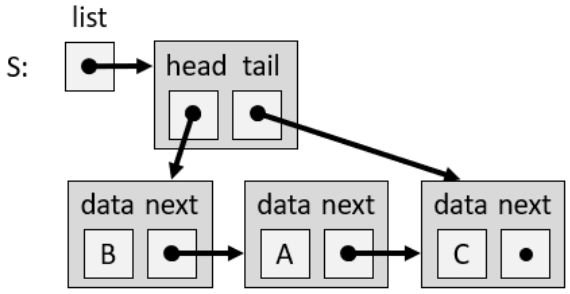
```
public interface SimpleListADT<T> {
  public NodeSL<T> getHead();
  public NodeSL<T> getTail();
  public void addFirst(T v);
  public void addLast(T v);
  public T removeFirst();
  public T removeLast();
  public T removeAfter(NodeSL<T> here);
  public void addAfter(NodeSL<T> here, T v);
}
```

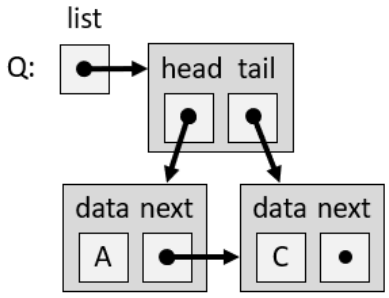
Assuming that the class is backed by a standard singly linked list, and we have an instance **`list`** whose starting configuration looks like this for each question below:

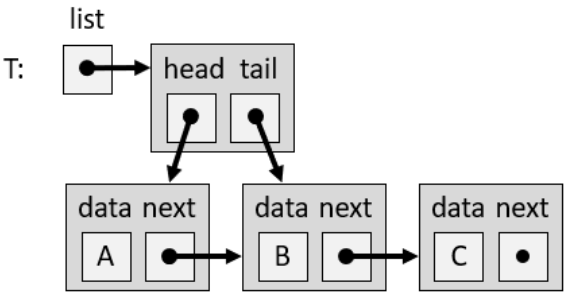


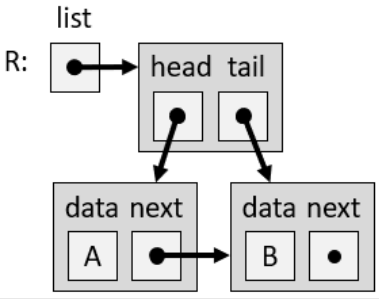
From the bank of possible answers below, choose the configuration (one of the options P through V) that would result from each of the operations shown, or say “Other” if the correct configuration is not among the options shown. Remember, each question part starts from the configuration above.

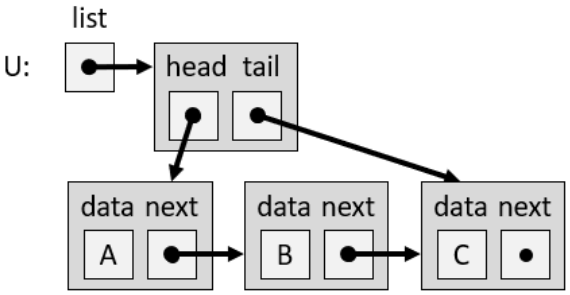












Questions:

```
// Part A
list.addAfter(here, 'B');
```

T

```
        // Part B
        list.addFirst('B');
```
S

```
        // Part C
        list.addLast(list.removeLast());
```
Q

```
        // Part D
        list.removeAfter(here);
        list.addLast('B');
```
R

```
        // Part E
        list.removeFirst('A');
        list.addFirst('B');
```
P

```
        // Part F
        list.getTail().setData('B');
```
R

Loops (8 points)

Imagine you are looking for an element in the following array, *arr*:

42	117	238	304	459	563	678	731	845

Consider the following pseudocode to search for a given element n within arr:

Initialize index to arr.length-1
While index is not negative and arr[index] not equal to n:
    Decrement index
Endwhile
Return index

1. What is the **loop variable** used in this loop?

   index

1. Under what condition(s) will iteration of the loop stop?

   If index is at the location of the target n, or is equal to -1

2. Let's say you are looking for element 731. When you find 731, what portion of the array has been actively processed (the "done" portion of the array)? Shade it in above.

   42 117 238 304 459 563 678 731 845

3. In one sentence, what is the **loop invariant** for your loop?

   The target n is not in the region of the array from index+1 to the end.

Copy Depth (14 points)

Consider the code below and answer the questions that follow.

```
char[] a = { 'X', 'Y', 'Z' };
char[] b = { 'X', 'Y', 'Z' };
char[] c = a;
```

```
char[] d = new char[3];
d[0] = a[0];
d[1] = a[1];
d[2] = a[2];
LinkedList<Character> e = new
    LinkedList<Character>(Arrays.asList('X', 'Y', 'Z'));
LinkedList<Character> f = new LinkedList<>(e);
LinkedList<Character> g =
    (LinkedList<Character>) e.subList(0, 3);
LinkedList<Character> h = new LinkedList<>();
h.add(a[0]);
h.add(a[1]);
h.add(a[2]);
```

1. Of the variables **a** through **h**, which are deep copies of each other? List all sets and draw a circle around each one.

   a, b, and d; *e, f, and h*

2. Of the variables **a** through **h**, which are shallow copies of each other? List all sets and draw a circle around each one.

   e and g

3. Of the variables **a** through **h**, which are aliases of each other? List all sets and draw a circle around each one.

   a and c

Recursion (12 points)

Consider the four recursive methods defined below (**plusOne**, **fib**, **listSum**, and **listMaximum** – note that these definitions may differ from similarly named ones you have seen before!) Answer the questions that follow.

```
LinkedList<Integer> plusOne(ListIterator<Integer> iter) {
    if (iter.hasNext()) {
        LinkedList<Integer> list = plusOne(iter);
        list.addFirst(1 + iter.next());
        return list;
```

```
    } else {
        return new LinkedList<Integer>();
    }
}

int fib(int n) {
    if (n == 0) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

private int listSum(ListIterator<Integer> iter, int sum) {
    if (iter.hasNext()) {
        return listSum(iter, iter.next() + sum);
    } else {
        return sum;
    }
}

public int listSum(LinkedList<Integer> list) {
    return listSum(list.listIterator(), 0);
}


public int listMaximum(ListIterator<Integer> iter) {
    if (iter.hasNext()) {
        int val = iter.next();
        int max = listMaximum(iter);
        return (val > max) ? val:max;
    } else {
        throw new NoSuchElementException();
    }
}
```

1. For each of the methods defined above, will the method always return normally when called? If it is possible to reach an infinite recursion or trigger an unhandled exception, you should write “no”. For each “no” answer, give a reason.

    a. plusOne yes/no:__*No*__ Reason: *Recursive call doesn’t simplify the problem first*

b. fib yes/no:__*No*__ Reason: *Call to fib(1) goes to fib(-1) which will make an infinite recursion*

c. listSumyes/no:__*Yes*_ Reason:

d. listMaximum yes/no:__*No*__ Reason: *Base case triggers an exception*

2. For each of the methods defined above, is it an example of tail recursion?

   a. plusOne yes/no:__*No*____

   b. fib yes/no:__*No*____

   c. listSumyes/no:__*Yes*____

   d. listMaximum yes/no:__*No*____

Additional topics:

Back-Up Questions

Add & Remove (LL & arrays)

Consider the runtime performance ("big-O notation") of the following sets of operations, in terms of the problem size *n*. Compare the following operations by circling the appropriate operator (greater than, less than, or approximately equal to, respectively) and provide your rationale:

1. The cost of inserting a new element at the head of a linked list is

   > **<** **≈**

the cost of inserting a new element at the beginning of an array (T[ ])

Why?

2. The cost of inserting a new element at the end of a linked list is

   > < ≈

   the cost of inserting a new element at the end of an array (T[ ])

   Why?

3. The cost of searching through a linked list item by item to find a particular value is

   > < ≈

   the cost of searching through an array index by index to find a particular value

   Why?

4. The cost of looking for an item in a linked list (using any approach) is

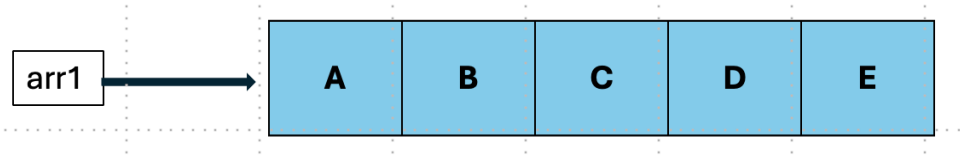
   > < ≈

   the cost of looking for an item in an array (using any approach)

   Why?

Copy Depth

Consider the object *arr1*, as shown below:



For each of the calls below, draw what the resulting object(s) would look like. If any changes are made to arr1, make sure to draw it and label it clearly as arr1. Otherwise, note that arr1 is unchanged.

1. Call to a copy constructor

2. Splice