Name: _Christopher Welch_____ Email: _cwelch25@msudenver.edu_____
Print legibly!

CS 2050
Computer Science II

Final Exam
Spring 2020

Exam instructions (please read them):

- this exam is being applied remotely due to the current covid-19 crisis;
- at the scheduled time, download the exam from Blackboard;
- you may choose between two formats: PDF or Word;
- plan ahead and decide how are you going to write your answers;
- you can choose to do it directly on your computer (using an appropriate text editor) or to print and write your answers by hand.
- should you choose to write your answers by hand, make sure you reserve enough time before the exam ends to scan your answers and submit them through Blackboard;
- during the two hours scheduled for the exam you must remain logged in on the Microsoft Teams' platform with your camera (and audio) ON, microphone muted;
- if you can't have a functional camera you will not be allowed to take the exam;
- the instructor must be able to see the student throughout the exam;
- this exam is closed book and notes;
- you are not allowed to use the internet or any external software tools (like IDEs or compilers, for example);
- you are not allowed to use headphones or headsets;
- you are not allowed to use hats, beanies, or hoods;
- you should not seek external help;
- you should not communicate with others during the exam.

Failure to follow these instructions will result in a zero for the exam.

CLOSED BOOK - CLOSED NOTES

YOUR SCORE: _____ POINTS

Q1 [20 points] Answer the multiple choice questions by circling (or highlighting) your choice.

1) Consider the linked list `[4, 10, 2, 8]`. If `current` (a reference to a node) is set to the *tail* node, what happens if you attempt to do `current.getNext().getNext()`?
a) you will get a reference to node 4
b) you will get a reference to node 8
c) you will get a compiler error
d) you will get a runtime error

2) Consider a singly linked list. If `current` (a reference to a node) is set to the *head* node, what best describes the procedure to *clear* the list?
a) set `current` to `null`
b) while `current` is not `null`, assign `current`'s next to `temp`, set `current`'s next to `null`, assign `temp` to `current`;
c) same as b, but when the loop is over, assign *head* to `null`
d) none of the above

3) Consider a singly linked list. What best describes the procedure to *clone* the list?
a) instantiate a new list and return it
b) instantiate a new list, append (tail insert) all of the elements from the original list into the new list, and then return the new list
c) instantiate a new list, *add* (head insert) all of the elements from the original list into the new list, and then return the new list
d) none of the above

4) Consider a singly linked list and the `set` method implementation shown below (assume that index is always valid).

```
public void set(int index, T data) {
    int i = 0;
    Node<T> current = head;
    while (i < index) {
        i++;
        current = current.getNext();
    }
    current.setData(_________);
}
```

Which option correctly completes the blank?
a) data
b) current
c) index
d) i

5) Consider a generic implementation of a singly linked list. Which option correctly instantiates a list of `String` objects, saving the returned reference to a variable named `lst`?

a) `LinkedList lst = new LinkedList();`
b) `LinkedList<> lst = new LinkedList();`
c) `LinkedList<String> lst = new LinkedList();`
d) `LinkedList<String> lst = new LinkedList<>();`

6) Consider an implementation of a singly linked list of integers with the `doIt` method described below.

```
public void doIt(LinkedList other) {
   if (size() != other.size())
       return;
   Node curr      = head;
   Node currOther = other.head;
   while (curr != null) {
       currOther.setData(curr.getData() + currOther.getData());
       curr = curr.getNext();
       currOther = currOther.getNext();
   }
}
```

If `lstA = [5, 7, 2]` and `lstB = [3, 8, 1]`, what is the content of `lstA` and `lstB` after `lstB.doIt(lstA)` is called?

a) `lstA = [5, 7, 2]` and `lstB = [3, 8, 1]`
b) `lstA = [8, 15, 3]` and `lstB = [3, 8, 1]`
c) `lstA = [5, 7, 2]` and `lstB = [8, 15, 2]`
d) `lstA = [14]` and `lstB = [3, 8, 1]`

7) Consider an implementation of a singly linked list of integers with the `doIt` method described below.

```
public void doIt() {
   Node current = head;
   while (current.getNext() != null) {
       current.setData(current.getData() - current.getNext().getData());
       current = current.getNext();
   }
}
```

If `lstA = [5, 7, 2]`, what is the content of `lstA` after `lstA.doIt()` is called?

a) `lstA = [5, 7, 2]`
b) `lstA = [5, 2, 0]`
c) `lstA = [-2, 5, 2]`
d) `lstA = [2, -5, 2]`

8) Consider the linked list `lstA = [5, 7, 2]`, another linked list `lstB = []` (empty), and an empty stack `stk`. What would be the content of `lstB` after a call to `lstA.doIt(lstB, stk)`?

```
public void doIt(LinkedList other, Stack stack) {
   Node current = head;
   while (current != null) {
       stack.push(current.getData());
       current = current.getNext();
   }
   while (!stack.isEmpty())
       other.append(stack.pop());
}
```

a) `lstB = [5, 7, 2]`

b) `lstB = [2, 7, 5]`

c) `lstB = []`

d) `lstB = [stk]`

9) What can we tell about a linked list that has `head == tail`?

a) the list is empty

b) the list has only one element

c) the list is empty or it has only one element

d) the list has an even number of elements

10) Consider a static stack of integers with a capacity of 4, initially empty, and referred by `stk`. What is the content of the stack after the following operations are executed?

```
stk.push(1); stk.push(4); stk.pop(); stk.push(9); stk.push(7);
stk.push(8); stk.peek(); stk.push(3);
```

a) `(top) 8 7 9 1 (bottom)`

b) `(top) 3 8 7 9 1 (bottom)`

c) `(top) 7 9 4 1 (bottom)`

d) `(top) 3 8 7 9 (bottom)`

11) Consider the arithmetic expression `5 + 2 - 8` which can also be written in postfix notation as `5 2 + 8 -`. The following code uses a stack to evaluate the expression. Assume that the only operations valid are addition and subtraction.

```
String exp[] = "5 2 + 8 -".split(" ");
Stack stk = new Stack();
for (int i = 0; i < exp.length; i++) {
   String term = exp[i];
   if (term.equals("+"))
       stk.push(stk.pop() + stk.pop());
   else if (term.equals("-"))
       stk.push(_____________________________);
   else
       stk.push(Integer.parseInt(term));
}
System.out.println(stk.pop());
```

Which option correctly completes the blank?

a) `stk.pop() * -1 + stk.pop()`

b) `stk.pop() * -1 - stk.pop()`

c) `stk.pop() - stk.pop()`

d) `stk.pop() * 2`

12) Consider that the *selection sort* algorithm is trying to sort `[1, 2, 3, 4, 5]`. How many (highlighted) comparisons will *selection sort* do to complete the sorting procedure? (consider ascending order)

```
public static void selectionSort(int data[]) {
   int i = 0;
   for (int j = 0; j < data.length; j++) {
      int min = j;
      for (int k = j + 1; k < data.length; k++)
         if (data[k] < data[min])
            min = k;
         int temp = data[i];
         data[i] = data[min];
         data[min] = temp;
         i++;
   }
}
```

a) 4

b) 7

c) 9

d) 10

13) Consider that the *insertion sort* algorithm is trying to sort `[1, 2, 3, 4, 5]`. How many (highlighted) comparisons will *insertion sort* do to complete the sorting procedure? (consider ascending order)

```
public static void insertionSort(int data[]) {
   for (int i = 1; i < data.length; i++) {
      int j = i;
      int k = i - 1;
      while (k >= 0) {
         if (data[j] < data[k]) {
            int temp = data[j];
            data[j] = data[k];
            data[k] = temp;
            j = k;
            k--;
         }
         else
            break;
      }
   }
}
```

a) 4
b) 7
c) 9
d) 10

14) Consider that the *merge sort* algorithm is trying to merge `[1, 2, 3]` and `[4, 5]`. How many (highlighted) comparisons will *merge sort* do to complete the merging procedure, called with `begin = 0`, `middle = 2`, and `end = 4`? (consider ascending order)

```
public static void merge(int data[], int begin, int middle, int end) {
   int i = begin;
   int j = middle + 1;
   int size = end - begin + 1;
   int sorted[] = new int[size];
   int k = 0;
   while (i <= middle && j <= end)
      if (data[i] < data[j])
         sorted[k++] = data[i++];
      else
         sorted[k++] = data[j++];
   while (i <= middle)
      sorted[k++] = data[i++];
   while (j <= end)
      sorted[k++] = data[j++];
   for (i = begin, k = 0; k < size; i++, k++)
      data[i] = sorted[k];
}
```

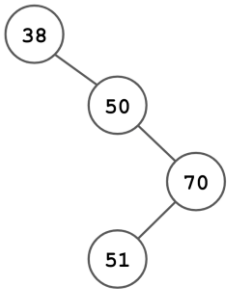
a) 2

b) 3
c) 4
d) 5

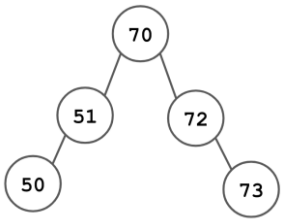
15) The *level order* traversal of a binary search tree is `16, 13, 28, 9, 23, 50`. The *pre-fixed* traversal of the same tree is:
a) `16, 13, 9, 23, 28, 50`
b) `9, 13, 16, 23, 28, 50`
c) `9, 13, 23, 50, 28, 16`
d) none of the above (`16, 13, 9, 28, 23, 50`)

16) Which of the trees is NOT a binary search tree?

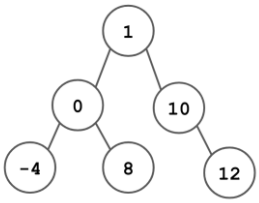
A)



b)

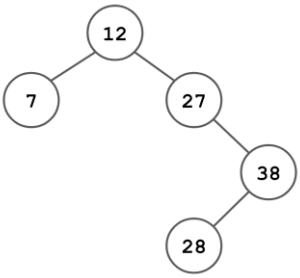


c)



d)

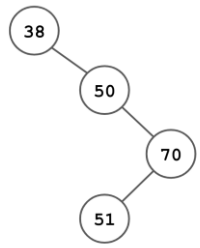
17) Consider the binary search tree below.



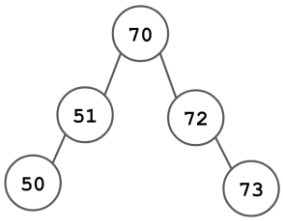
How many *node data* comparisons will be required to search for number 40 (that is evidently NOT in the collection)? Do not consider checking if the current node is null a comparison for your count.
a) 0
b) 1
c) 3
d) 5

18) A tree is considered to be balanced when the difference in height between its subtrees is no greater than one. Assume a recursive definition. According to this definition, which of the trees is NOT considered to be balanced?

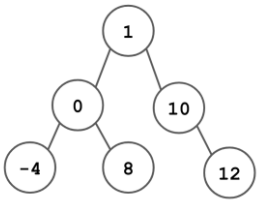
A)



b)

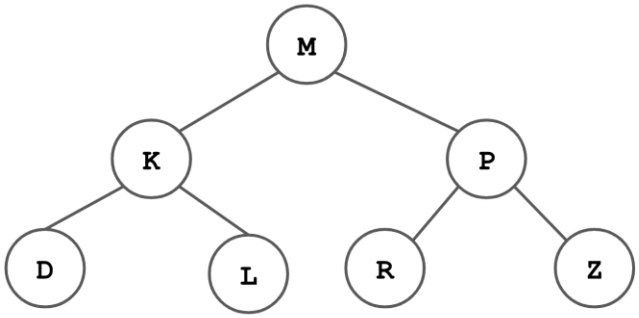


c)



d)

19) Which node invalidates the following binary search tree (consider alphabetical order)?



a) K
b) L
c) P
d) R

20) Using the numbers 4, 9, and 23, how many binary search trees can be built having 23 as the root node?
a) 0
b) 1
c) 2
d) 3

Q2 [10 points] The Java Collections framework has a class called `ArrayList` that implements a list backed up by an array. Consider the semi-complete implementation on the next page of an `ArrayList` of integers to answer the questions that follow.

```
public class ArrayList {
   private int size;
   private int capacity;
   private int data[];
   private static final int INCREMENT_SIZE = 10;
   public ArrayList() {
       capacity = INCREMENT_SIZE;
       data = new int[capacity];
       size = 0;
   }

   public void append(int data) {
       this.data[size++] = data;
       if (                    ) {
           int temp[] = new int[capacity + INCREMENT_SIZE];
           for (int i = 0; i < this.data.length; i++)
               temp[i] = this.data[i];
           this.data = temp;
           capacity = capacity + INCREMENT_SIZE;
       }
   }

   @Override
   public String toString() {
       String str = "";
       for (int i = 0; i < size; i++)
           str += data[i] + " ";
       return str;
   }

   public int getSize() {
       return size;
   }

   public int getCapacity() {
       return capacity;
   }

   public static void main(String[] args) {
       ArrayList al = new ArrayList();
       for (int i = 0; i < 10; i++)
           al.append(i + 1);
       al.append(11);
       System.out.println(al);
```

```
    }
}
```

1) What's the initial *size* and *capacity* of a newly instantiated `ArrayList`?

a) `0, 10` (respectively)
b) `0, 20` (respectively)
c) `10, 10` (respectively)
d) `10, 20` (respectively)

2) What would be the best choice to complete the blank in the code?

a) `size < capacity`
b) `size = capacity`
c) `size == capacity`
d) `size > capacity`

3) What's the expected result if you run the given code?

a) `ArrayIndexOutOfBoundsException` is thrown
b) `1 2 3 4 5 6 7 8 9 10`
c) `1 2 3 4 5 6 7 8 9 10 11`
d) none of the above

Q3 [10 points] Consider a typical implementation of a hash table that uses the *odd* hash function **`h(k) = | 3k - 6 | / 5`**. , where `| x |` is the absolute value of `x` and `/` is the integer division operator. How many collisions (if any) will happen if we use the following keys: 0, 5, 12, and 10? For each key, show the hash of the key to justify your answer.

`h(0) = | 3(0) - 6 | / 5` = 1
`h(5) = | 3(5) - 6 | / 5` = 1
`h(12) = | 3(12) - 6 | / 5` =6
`h(10) = | 3(10) - 6 | / 5` =4

h(0) & h(5) hashes to the same value of 1, causing a **hash collision**.

Q4 [10 points] Consider the semi-complete implementation on the next page of a `TreeMap` to answer the questions that follow.

```
public class BinNode<K, V> {
   private K  key;
   private V  val;
   private BinNode<K, V> left, right;

   public BinNode(K key, V val) {
       this.key = key;
       this.val = val;
       left = right = null;
   }

   public K getKey() {
       return key;
   }

   public V getValue() {
       return val;
   }

   public BinNode<K, V> getLeft() {
       return left;
   }

   public BinNode<K, V> getRight() {
       return right;
   }

   public void setKey(K key) {
       this.key = key;
   }

   public void setLeft(BinNode<K, V> left) {
       this.left = left;
   }

   public void setRight(BinNode<K, V> right) {
       this.right = right;
   }

   @Override
   public String toString() {
       return "(" + key + "," + val + ")";
   }
}
```

```
public class TreeMap<K extends Comparable<K>, V> {

   private BinNode<K, V> root;

   public TreeMap() {
       root = null;
   }

   public boolean isEmpty() {
       return root == null;
   }

   public void put(K key, V val) {
       root = addRecursively(root, key, val);
   }

   private BinNode<K, V> addRecursively(BinNode<K, V> current, final K key,
final V val) {
       if (current == null)
           return new BinNode<K, V>(key, val);
       else {
           if (_____A_____)
               current.setLeft(addRecursively(current.getLeft(), key, val));
           else if (_____B_____)
               current.setRight(addRecursively(current.getRight(), key, val));
           return current;
       }
   }

   public V get(K key) {
       return _____C_____;
   }

   private V searchRecursively(final BinNode<K, V> current, final K key) {
       if (current == null)
           return null;
       else if (key.compareTo(current.getKey()) == 0)
           return current.getValue();
       else if (key.compareTo(current.getKey()) < 0)
           return searchRecursively(current.getLeft(), key);
       else
           return searchRecursively(current.getRight(), key);
   }
```

```
    public String inOrder(final BinNode<K, V> current) {
        if (current != null)
            return  inOrder(current.getLeft())  +
                    current.toString() + " "     +
                    inOrder(current.getRight());
        return "";
    }

    @Override
    public String toString() {
        return inOrder(root);
    }

    public static void main(String[] args) {
        TreeMap<Integer, String> tm = new TreeMap<>();
        tm.put(4, "Trinity");
        tm.put(1, "Morpheus");
        tm.put(7, "Neo");
        tm.put(0, "Agent Smith");
        System.out.println(tm);
    }
}
```

1) How would you complete the blanks labeled **A** and **B** in `TreeMap`'s `addRecursively` method?

A. `key.compareTo(current.getKey()) <= 0`

B. `key.compareTo(current.getKey()) > 0`

2) How would you complete the blank labeled **C** in `TreeMap`'s `get` method?

```
return root.getValue();
```

3) What's the expected output when you run `TreeMap`?

(0,Agent Smith) (1,Morpheus) (4,Trinity) (7,Neo)

(^The Matrix would say there is no TreeMap.^) Enjoy your Summer Break! Stay Safe!