Name________________________

CSCI 151
Exam 2
December 17, 2021

This is a closed-book, closed-notes, closed-Internet exam.

The exam has 6 numbered questions; each is worth 16 points. You get 4 points for free, for a total of 100 points.

The last page of the exam is blank; you can use this for additional space for any of the exam questions.

After you have finished the exam please indicate whether you followed the Honor Code on the exam.

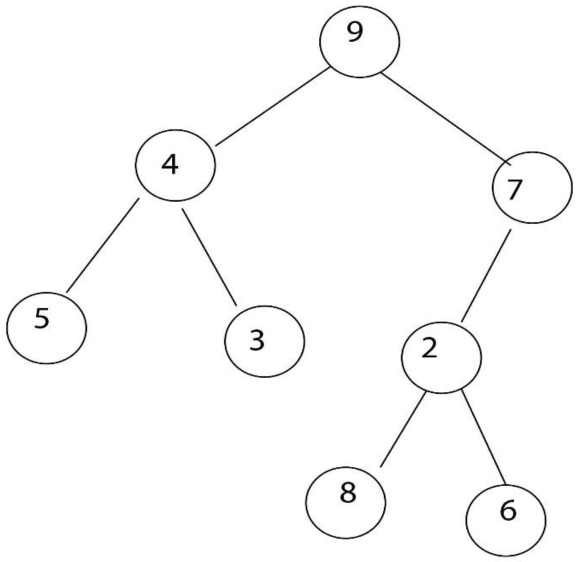
I ☐ did ☐ did not

adhere to the Honor Code while taking this exam.

____

Signature

1. Here is a picture of a binary tree. Give **pre-order, in-order, and post-order traversals** of this tree.



Pre-order: 9 4 5 3 7 2 8 6

In-order: 5 4 3 9 8 2 6 7

Post-order 5 3 4 8 6 2 7 9

2. We have discussed three O(n*log(n)) sorting algorithms this semester: MergeSort, QuickSort, and HeapSort. **Choose one of them – any one you wish..**

A. **Describe in one or two sentences how this algorithm works.**

MergeSort: Split the list into two halves, recursively sort them, then merge them back together.

QuickSort: Choose one element as the "pivot", rearrange the list so that everything less than the pivot is to its left and everything greater than the pivot is to its right, then recursively sort everything to the left of the pivot and recursively sort everything to the right of the pivot.

HeapSort: Consider the list as a tree and "heapify" as a MaxHeap (i.e, largest element is at the root). Then repeatedly poll the heap and put the largest element in place in the list.

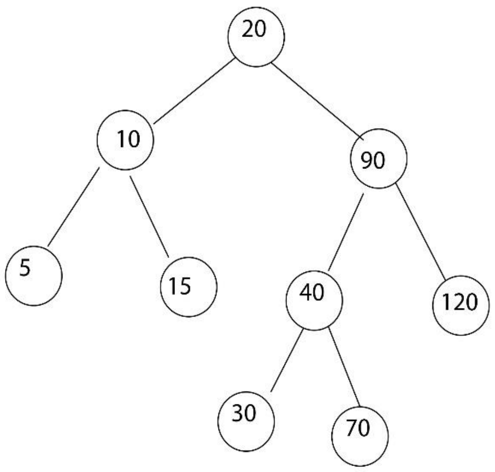
B. What is **one feature** of this algorithm that would either incline you to use it or not to use

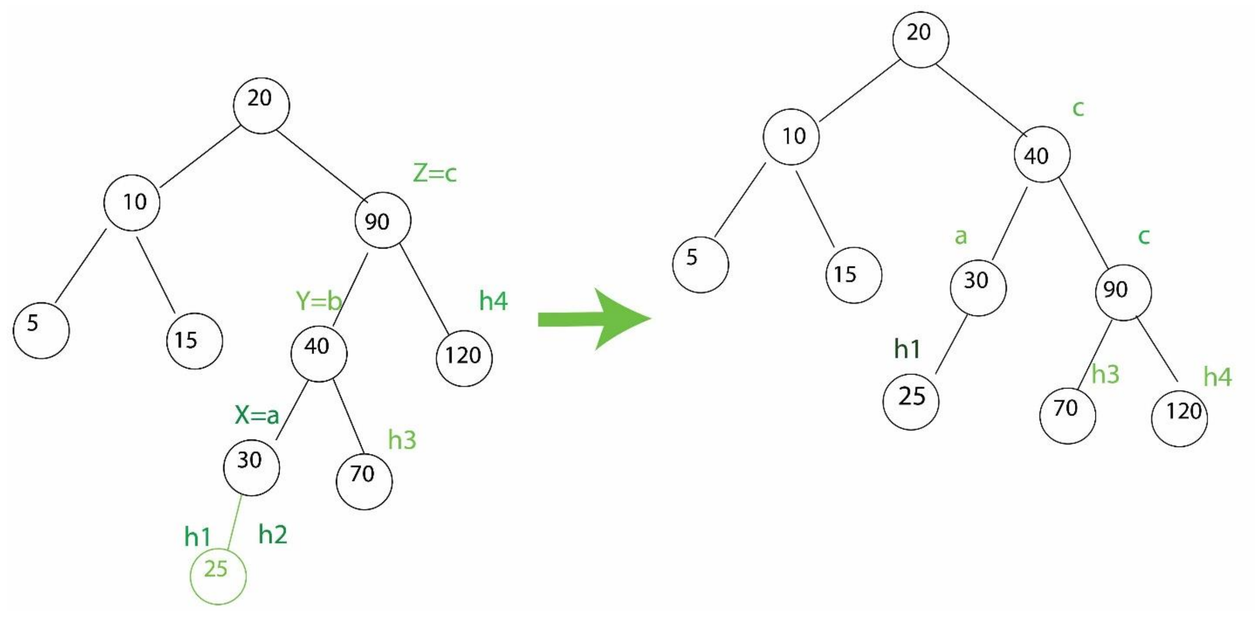
MargeSort. Good: it is fast and easy to code. Always runs in O(n*log n) Bad: it requires an additional list the same size as the one being sorted. Always runs in O(n*log n); doesn't speed up if the list is almost sorted.

QuickSort. Good: sorts in place (no additional storage needed. Bad doesn't guarantee O(n*log n) performance.

HeapSort. Good: fast, sorts in place, isn't recursive. Bad: Complicated to code.

3. Here is an AVL tree. **Draw the AVL tree that results from inserting 25 into this tree.**





4. Here is a list of values and their hash codes:

value	4	2	18	19	11	7	9	0	15
hashcode	4	2	0	1	2	7	0	0	6

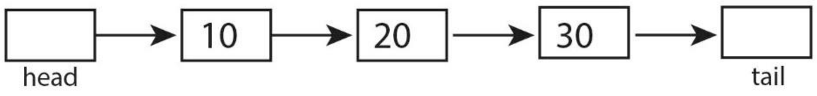
Add these values from left to right (first 4, then 2, then 18 …) to the following hash table. The small numbers are the table indices, so you don't have to count across to see where an index such as 6 occurs.

0	1	2	3	4	5	6	7	8
18	19	2	11	4	9	0	7	15

5. We have a linked list class based on the following structure:

```
        class MyLinkedList {
              Node head, tail;
              MyLinkedList( ) {
                     head = new Node();
                     tail = new Node();
                      head.next = tail;
              }
        ……
      }
```

To save space I am not going to write the Node class. Nodes have an integer value *data* and a link *next*. There are two Node constructors; one takes an integer argument and builds a Node with that value; the other takes no argument and builds an empty Node. You can directly access the fields of a Node: if x and y are Nodes you can say things like x.next = y; and x.data = 23.
Here is a picture of a typical list. Note that head and tail are empty sentinel nodes guarding the front and rear of the list; the list is empty when head points to tail.



Give a method InsertInOrder(int d) for the MyLinkedList class that will insert a new element d into the list so that, if the list is ordered from smallest to largest before this method is called, then d will be inserted at a location that preserves the order. For example, if we call InsertInOrder(25) with the list pictured, 25 will be placed between 20 and 30.

```
void InsertInOrder(int d) {
        Node p = head;
        Node q = p.next;
        while (q != tail &&  q.value < d) {
                q = q.next;
                p = p.next;
        }
        Node r = new Node(d);
        p.next = r;
        r.next = q;
}
```

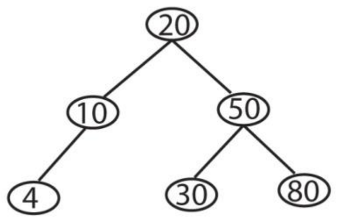
6. Here is a class for Binary Search Trees that hold integer values

```
public class BST {
        int value;
        BST left, right;
        int size; // the number nodes in the tree with this as root
}
```

Below is a picture of a tree using this structure. Note that there is no EmptyTree class; empty trees are represented by null pointers. Write a method for this class

int kth(int k)

that retuns the kth largest value in the tree. For the tree pictured kth(0) is 4, kth(1) is 10, kth(2) is 20 and kth(4) is 50. You can assume k will be between 0 and the size of the tree; you don't have to test for that.



A cheesy way to do this is to put all of the values into a list, sort the list, and return the value at index k. Instead of that, I am looking for a recursive solution.

```
int kth(int k) {
        int lsize, rsize;
        if (left == null)
                lsize = 0;
        else lsize = left.size;
        if (right == null)
                rsize = 0;
        else rsize = right.size;
        if (k == lsize)
                return value;
        else if (k < lsize)
                return left.kth(k)
        else // k > lsize
                return right.kth(k-1-lsize);
}
```

You can use this   page for additional space for any question.