Initials:_____

Problem 1 (6 Points, 1 point each)

TRUE or FALSE Questions

a) A binary search tree of size n always has height O(log n).

b) For a given key k that is found in the binary search tree T, the smallest key larger than k (if it exists) lies on the path between the root of the tree and the node containing the key k.

c) Both a singly and doubly Linked Lists have O(n) access (find) time. The expected time (average time) to access is faster for a doubly linked list than for a singly linked list.

d) Given a heap H containing an element x, if that element is removed and then added again (with no other operations performed in between), the resulting heap will be exactly the same as the original heap.

e) Linear probing is equivalent to double hashing with a secondary hash function of $h_2(k)=1$.

f) John Von Neumann was part of the team that invented the transistor.

Problem 2 (5 points)

Short Answer Question

Suppose you have linked list storing instances of an object that has instance variables A and B. Further suppose that you keep the linked list sorted according to A. You now want to **also** be able to efficiently find the top N items according to B. What data structure would you use and why? How much additional space would this data structure require? (No code. Drawings might help your answer)

Problem 3 (18 Points, 1 point each) Complexity

a) For each of the following algorithms, indicate their worst-case running time using the Big-Oh notation. No explanations necessary.

1) Post-order traversal of a binary tree of size n assuming each node visit takes O(n) time.
2) Deleting an entry from an AVL tree of size n.
3) Heapsort on a list of n elements.

b) For each of the following operations, give the expected time in Big-Oh notation (yes, Big-0 is worst case; give the expected time). Your answers should assume that the data structure is performing well, i.e., binary search trees are well balanced and hashtables have a low load factor. No explanations necessary.

1) Delete
    1) unsorted list
    2) sorted list implemented with array
    3) sorted list implemented with linked list
    4) hashtable
    5) balanced binary search tree

2) Contains
    1) unsorted list
    2) sorted list implemented with array
    3) sorted list implemented with linked list
    4) hashtable
    5) balanced binary search tree

3) FindMax
    1) unsorted list
    2) sorted list
    3) binary heap
    4) hashtable
    5) balanced binary search tree

Problem 4 (16 Points: 10 points for part A, 6 for part B) Binary Tree

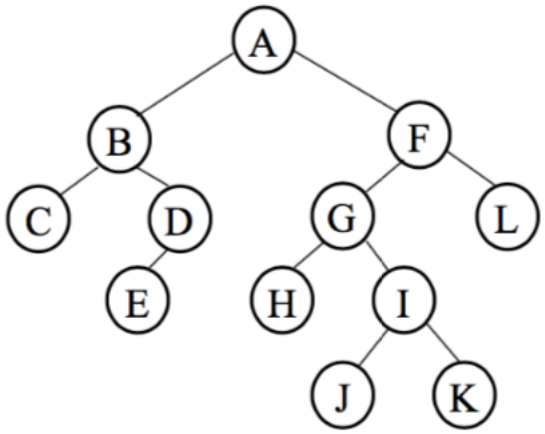
Consider a LinkedBinaryTree, with the method "mystery" given below. (Assume standard offer and poll methods for the Queue. Also assume standard tree functions for the LinkedBinaryTree)

```
public class LinkedBinaryTree<E extends Comparable<E>> {
      private class Node<E> {
            public Node<E> left = null;
            public Node<E> right = null;
            public E element;
            // The Node class also includes constructors, etc
      }
    private Node<E> root = null;

    public void mystery() {
      Queue<Node<E>> q = new Queue<>();
      q.offer(root);
      while(!q.isEmpty()) {
            Node<E> n = q.poll();
            System.out.print(n.getElement()+ " ");
            if (n.right!=null)
                  q.offer(n.right);
            if (n.left!=null)
                  q.offer(n.left);
} }
```

a) Write top-level and line-by-line (as appropriate) documentation for the mystery function. This documentation should show a clear understanding of both what the mystery function does, and how it does that.

Suppose that in the Main method, LinkedBinaryTree<String> object **t** has been constructed and nodes were inserted into the tree so that **t** looks like this:



b) What does a call to t.mystery() print out for the tree above?

Problem 5 (10 Points: 2 points for part A, 4 each for parts B and C)

Binary Heap

The following array represents an array-based binary minheap. (For parts b and c, show you work to be eligible for partial credit.)

a) Show the graphical representation of this heap.

0	1	2	3	4	5	6
5	10	8	50	25	9	

b) Show the contents of the **array** after inserting 4, then 0 then 6 into the original heap.

c) Show the contents of the **array** after poll is executed twice on the heap that results from part b. (You can have a correct answer to part c even if your answer to part b is incorrect.)

Problem 6 (14 Points) AVL Trees

a) Show the result of inserting 2, 1, 16, 30, 41, 17, 24, 19, 18 in this order into an initially empty AVL tree that implements a Binary Search Tree. If the insertion of a new entry does not cause rotations, you can show the insertion on the same tree. If an insertion causes rotation(s), draw a new trees to show all intermediate steps. Show each rotation clearly.

b) Show the result of removing 16 then 19, from the AVL tree created in part (a). Use the largest lesser value for replacement. Again, show rotations clearly.

Problem 7 (13 Points: 8 points for part A, 5 points for part B)

A binary tree can be stored in an array in exactly the same manner as a heap. (Root of the tree in position 0, left child of root in position 1, right child in position 2, etc.) For each of the following arrays, is the array a binary MAX heap (that is a heap with every parent greater than its children)? A Binary Search Tree?

Array 1	7	6	5	4	3	2	1
Array 2	500	400	300	200	450	100	150
Array 3	1000	500	2000	250	800	1500	2500
Array 4	1000	700	900	600	1	850	880

	Is Heap (Y/N)	Is BST (Y/N)
Array 1		
Array 2		
Array 3		
Array 4		

PART B:

Although possible, it is unusual to store BSTs in arrays. On the other hand, heaps are almost always stored in arrays. Explain why one data structure (the heap) is well-suited to array storage while the other (BST) is not. (Your explanation should probably include some drawings that illustrate the points you make.)

Problem 8 (20 Points) Linked Lists

You are given a linked list class (named LiLi) that implements the following interface.

```
public interface LLInterface<G> {
    /**
     * @return the first item in the linked list
     * This also resets getNext as documented in getNext below
     * **/
    public G getFirst();

    /**
     * @return the next item in the linked list or null if there is no
     * next item. For instance: getFirst() immediately followed by
     * getNext() will always return the second item in the list (or
     * null if the list had 0 or 1 items).  getFirst() followed by
     * 10 calls to getNext() would return the 11th item (or null if
     * the list had less than 11 items).
     * **/
    public G getNext();

    /**
     * Adds the data item to the linked list, somewhere.
     */
    public void addToList(G g);
}
```

In addition, the LiLi has a constructor that takes no arguments. LiLi may implement other interfaces and have other public methods; use only these.

(problem 8 is continued on next page)

Implement the following function, recursively. DO NOT use any loops. (You may use one — or more — recursive helper functions.)

```
/**
 * Return a new LinkedList containing every Nth item in the
 * original linked list.  For instance, given a linked list
 * containing (a,s,d,f,g,h,j,k,l) and N=3, the returned linked
 * list would contain (a,f,j).  For the same list, with N=2, the
 * function would return (a,d,g,j,l)
 * @param the linked list to select from
 * @param N - the rate of selection as above
 * @return a new linked list containing the selected items
 */
public LiLi subsetter(LiLi source, int N);
```

If you cannot figure out how to write this recursively, write the function using loops for a 20% penalty.

Problem 9 (12 Points: 4 points for each part)

Consider the following two methods. The first method you should recognize; it is the partition method from quicksort. The second method is new.

```
private int partition(int[] arr, int begin, int end) {
    int pivot = arr[end];
    int insertLoc = (begin - 1);
    for (int j = begin; j < end; j++) {
        if (arr[j] <= pivot) {
            insertLoc++;
            int swapTemp = arr[insertLoc];
            arr[insertLoc] = arr[j];
            arr[j] = swapTemp;
        }
    }
    insertLoc++;
    int swapTemp = arr[insertLoc];
    arr[insertLoc] = arr[end];
    arr[end] = swapTemp;
    return insertLoc;
}

public void m(int[] arr) {
    int b = 0;
    int e=arr.length-1;
    int p=-1;
    int m = (arr.length - 1) / 2;
    int r = 0;
    while (true) {
        p = partition(arr, b, e);
        if (p == m)
            break;
       if (p<m) {
            b=p+1;
        } else {
            e = p - 1;
        }
        r++;
    }
    System.out.println(r + " " + p + " " + arr[p]);
}
```

Part A: In one sentence (or two at most), what does the method m compute?

Part B: For an array of length 1000 that contains each of the numbers 0 … 1000 in random order, what is the value of p when the method completes?

Part C: What is the expected runtime of the method m. What is the worst case runtime for the method? Give an array of length 10 on which the worst case runtime occurs.

Extra Credit: (2 points) For a given value N, on the array from part B, when the function completes what do you know about the location of N? Why? (For example, what do you know for certain about the location of 151. One possible, but wrong, answer is the N must be in position N-1)

Problem 1 (6 Points, 1 point each)

TRUE or FALSE Questions

a) A binary search tree of size n always has height O(log n).

FALSE, the height is between lgn and n

b) For a given key k that is found in the binary search tree T, the smallest key larger than k (if it exists) lies on the path between the root of the tree and the node containing the key k.

False, inorder successor is down right, then all left

c) Both a singly and doubly Linked Lists have O(n) access (find) time. The expected time (average time) to access is faster for a doubly linked list than for a singly linked list.

False, adding the links does not affect time

d) Given a heap H containing an element x, if that element is removed and then added again (with no other operations performed in between), the resulting heap will be exactly the same as the original heap.
FALSE, the remove and add operations may move stuff around.

e) Linear probing is equivalent to double hashing with a secondary hash function of $h_2(k)=1$.
TRUE

f) John Von Neumann was part of the team that invented the transistor.
FALSE (we did not talk about him in class this semester)

Problem 2 (5 points)

Short Answer Question

Suppose you have linked list storing instances of an object that has instance variables A and B. Further suppose that you keep the linked list sorted according to A. You now want to **also** be able to efficiently find the top N items according to B. What data structure would you use and why? How much additional space would this data structure require? (No code. Drawings might help your answer)

Lots of options. But I will say a linked list that is sorted according to B. In this case, the nodes in the second linked list, can point to the same objects as the first linked list. So the extra space required for the second linked list is the space for N nodes, each of which has 2 pointers, one to the data objects tat are also pointed to by the first linked list and one for the link to the next node. Hence, the space is really just the space required for 2N pointers

Problem 3 (18 Points, 1 point each) Complexity

a) For each of the following algorithms, indicate their worst-case running time using the Big-Oh notation. No explanations necessary.

1) Post-order traversal of a binary tree of size n assuming each node visit takes O(n) time.
   O(n)
2) Deleting an entry from an AVL tree of size n.
   O(lg n)
3) Heapsort on a list of n elements.
   O(n * lg(n))

b) For each of the following operations, give the expected time in Big-Oh notation (yes, Big-0 is worst case; give the expected time). Your answers should assume that the data structure is performing well, i.e., binary search trees are well balanced and hashtables have a low load factor. No explanations necessary.

1) Delete
   1) unsorted list — O(n/2)
   2) sorted list implemented with array — O(lg n)
   3) sorted list implemented with linked list — O(n/2)
   4) hashtable — O(n/2)
   5) balanced binary search tree — O(lg n)

2) Contains
   1) unsorted list — O(n/2)
   2) sorted list implemented with array — O(lg n)
   3) sorted list implemented with linked list — O(lg n)
   4) hashtable — O(n/2)
   5) balanced binary search tree — O(lg n)

3) FindMax
   1) unsorted list — O(n)
   2) sorted list O(1)
   3) binary heap O(1) or O(n) depending on direction of heap
   4) hashtable — O(n)
   5) balanced binary search tree — O(lg n)

Problem 4 (16 Points: 10 points for part A, 6 for part B) Binary Tree

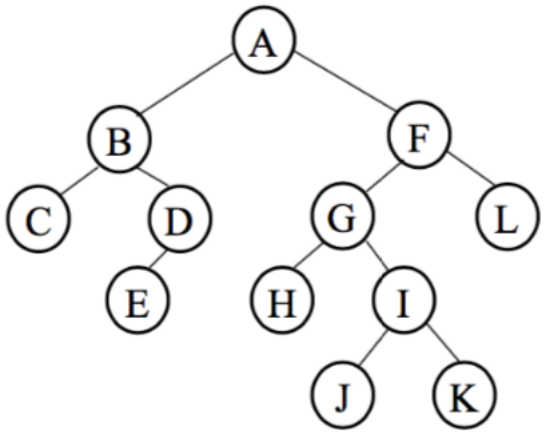
Consider a LinkedBinaryTree, with the method "mystery" given below. (Assume standard offer and poll methods for the Queue. Also assume standard tree functions for the LinkedBinaryTree)

```
public class LinkedBinaryTree<E extends Comparable<E>> {
      private class Node<E> {
            public Node<E> left = null;
            public Node<E> right = null;
            public E element;
            // The Node class also includes constructors, etc
      }
    private Node<E> root = null;

    /**
     * Print a pre-order traversal of the tree, right nodes first.
     * After first adding the root, the loop works by dequeuing the first
element
     * printing it, then adding its children.
     * At any given time, the queue may have a bunch of left node, but it
only briefly has right nodes.  The number of left nodes held is at most the
depth of the branch of the tree being explored.
     **/
    public void mystery() {
      // Create a queue
      Queue<Node<E>> q = new Queue<>();
      // add root to the queue
      q.offer(root);
      while(!q.isEmpty()) {
            Node<E> n = q.poll();
            System.out.print(n.getElement()+ " ");
            if (n.right!=null)
                  q.offer(n.right);
            if (n.left!=null)
                  q.offer(n.left);
} }
```

a) Write top-level and line-by-line (as appropriate) documentation for the mystery function. This documentation should show a clear understanding of both what the mystery function does, and how it does that.

Suppose that in the Main method, LinkedBinaryTree<String> object **t** has been constructed and nodes were inserted into the tree so that **t** looks like this:



b) What does a call to t.mystery() print out for the tree above?
A F L G I K J H B D E C

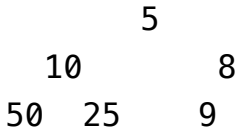
Problem 5 (10 Points: 2 points for part A, 4 each for parts B and C)

Binary Heap

The following array represents an array-based binary minheap. (For parts b and c, show you work to be eligible for partial credit.)

a) Show the graphical representation of this heap.

0	1	2	3	4	5	6
5	10	8	50	25	9	



b) Show the contents of the **array** after inserting 4, then 0 then 6 into the original heap.
0 4 5 6 25 9 8 50 10

c) Show the contents of the **array** after poll is executed twice on the heap that results from part b. (You can have a correct answer to part c even if your answer to part b is incorrect.)

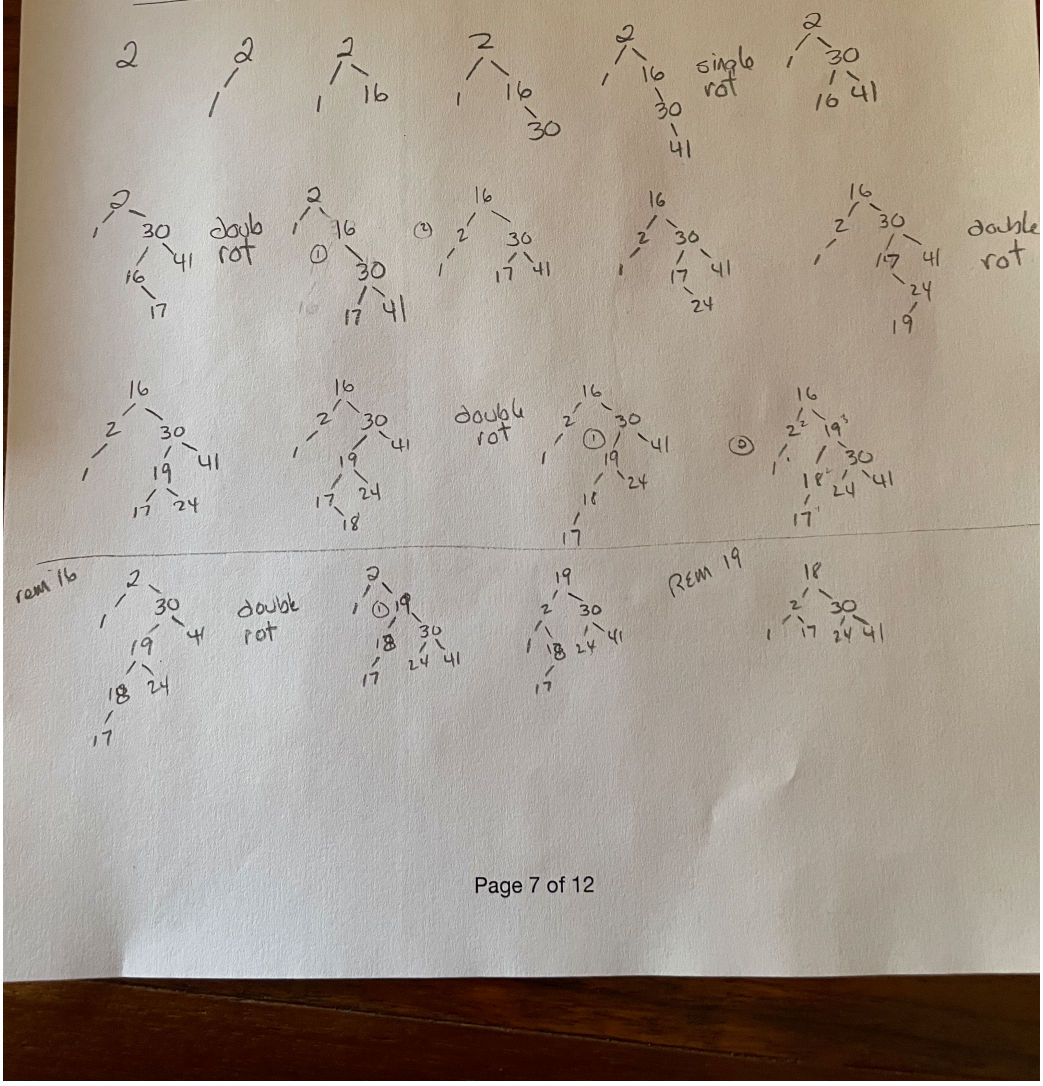
5 6 8 10 25 9 50

Initials:_____

Problem 6 (14 Points) AVL Trees

a) Show the result of inserting 2, 1, 16, 30, 41, 17, 24, 19, 18 in this order into an initially empty AVL tree that implements a Binary Search Tree. If the insertion of a new entry does not cause rotations, you can show the insertion on the same tree. If an insertion causes rotation(s), draw a new trees to show all intermediate steps. Show each rotation clearly.

b) Show the result of removing 16 then 19, from the AVL tree created in part (a). Use the largest lesser value for replacement. Again, show rotations clearly.



Problem 7 (13 Points: 8 points for part A, 5 points for part B)

A binary tree can be stored in an array in exactly the same manner as a heap. (Root of the tree in position 0, left child of root in position 1, right child in position 2, etc.) For each of the following arrays, is the array a binary MAX heap (that is a heap with every parent greater than its children)? A Binary Search Tree?

Array 1	7	6	5	4	3	2	1
Array 2	500	400	300	200	450	100	150
Array 3	1000	500	2000	250	800	1500	2500
Array 4	1000	700	900	600	1	850	880

	Is Heap (Y/N)	Is BST (Y/N)
Array 1	Y	N
Array 2	N	N
Array 3	N	Y
Array 4	Y	N

PART B:

Although possible, it is unusual to store BSTs in arrays. On the other hand, heaps are almost always stored in arrays. Explain why one data structure (the heap) is well-suited to array storage while the other (BST) is not. (Your explanation should probably include some drawings that illustrate the points you make.)

Heaps in arrays make sense because there are no gaps. Consider a tree with root 100, left 50 no right, then left 25, no right, left 10, then left 5 under the 10. It is a fine BinarySearchTree. The smallest array that could contain this using the heap array placement formula is 16 places, of which only 5 are filled. If you added 2 under 5 you would need 31 places of which only 6

would be filled. More generally, given a tree with N items, you could need an array with 2^N spaces to hold it. This is NOT true of heaps.

Problem 8 (20 Points) Linked Lists

You are given a linked list class (named LiLi) that implements the following interface.

```
public interface LLInterface<G> {
    /**
     * @return the first item in the linked list
     * This also resets getNext as documented in getNext below
     * **/
    public G getFirst();

    /**
     * @return the next item in the linked list or null if there is no
     * next item. For instance: getFirst() immediately followed by
     * getNext() will always return the second item in the list (or
     * null if the list had 0 or 1 items).  getFirst() followed by
     * 10 calls to getNext() would return the 11th item (or null if
     * the list had less than 11 items).
     * **/
    public G getNext();

    /**
     * Adds the data item to the linked list, somewhere.
     */
    public void addToList(G g);
}
```

In addition, the LiLi has a constructor that takes no arguments. LiLi may implement other interfaces and have other public methods; use only these.

(problem 8 is continued on next page)

Implement the following function, recursively. DO NOT use any loops. (You may use one — or more — recursive helper functions.)

```
/**
 * Return a new LinkedList containing every Nth item in the
 * original linked list.  For instance, given a linked list
 * containing (a,s,d,f,g,h,j,k,l) and N=3, the returned linked
 * list would contain (a,f,j).  For the same list, with N=2, the
 * function would return (a,d,g,j,l)
 * @param the linked list to select from
 * @param N - the rate of selection as above
 * @return a new linked list containing the selected items
 */
public LiLi subsetter(LiLi source, int N);
```

If you cannot figure out how to write this recursively, write the function using loops for a 20% penalty.

```
public LiLi subsetter(LiLi source, int N) {
   LiLi rtn = new LiLi();
   subsetterUtil(source, rtn, N, 0);
   return rtn;
}
private void subsetterUtil(LiLi source, LiLi rtn, int
N, int count) {
   G gg;
   if (count==0)
      gg=source.getFirst();
  else
      gg=source.getNext();
  if (gg==null)
      return;
  if (count%N==0)
     rtn.addToList(gg);
  subsetterUtil(source, rtn, N, count+1);
}
```

Problem 9 (12 Points: 4 points for each part)

Consider the following two methods. The first method you should recognize; it is the partition method from quicksort. The second method is new.

```
private int partition(int[] arr, int begin, int end) {
    int pivot = arr[end];
    int insertLoc = (begin - 1);
    for (int j = begin; j < end; j++) {
        if (arr[j] <= pivot) {
            insertLoc++;
            int swapTemp = arr[insertLoc];
            arr[insertLoc] = arr[j];
            arr[j] = swapTemp;
        }
    }
    insertLoc++;
    int swapTemp = arr[insertLoc];
    arr[insertLoc] = arr[end];
    arr[end] = swapTemp;
    return insertLoc;
}

public void m(int[] arr) {
    int b = 0;
    int e=arr.length-1;
    int p=-1;
    int m = (arr.length - 1) / 2;
    int r = 0;
    while (true) {
        p = partition(arr, b, e);
        if (p == m)
            break;
       if (p<m) {
            b=p+1;
        } else {
            e = p - 1;
        }
        r++;
    }
    System.out.println(r + " " + p + " " + arr[p]);
}
```

Part A: In one sentence (or two at most), what does the method m compute?

the median

Part B: For an array of length 1000 that contains each of the numbers 0 … 1000 in random order, what is the value of p when the method completes?

499

Part C: What is the expected runtime of the method m. What is the worst case runtime for the method? Give an array of length 10 on which the worst case runtime occurs.

expected = O(lg 1000)

Worst case would be 1,2,3,4,5,6,7,8,9,10

Extra Credit: (2 points) For a given value N, on the array from part B, when the function completes what do you know about the location of N? Why? (For example, what do you know for certain about the location of 151. One possible, but wrong, answer is the N must be in position N-1)

If N>median it will be in the upper part of the list, if N less than median it will be in the loder part of the list.