DUKE UNIVERSITY
Department of Computer Science

Test 2: CompSci 100

Name (print): _____

Community Standard acknowledgment (signature): _____

	value	grade
Problem 1	8 pts.	
Problem 2	8 pts.	
Problem 3	8 pts.	
Problem 4	14 pts.	
Problem 5	28 pts.	
Problem 6	9 pts.	
TOTAL:	75 pts.	

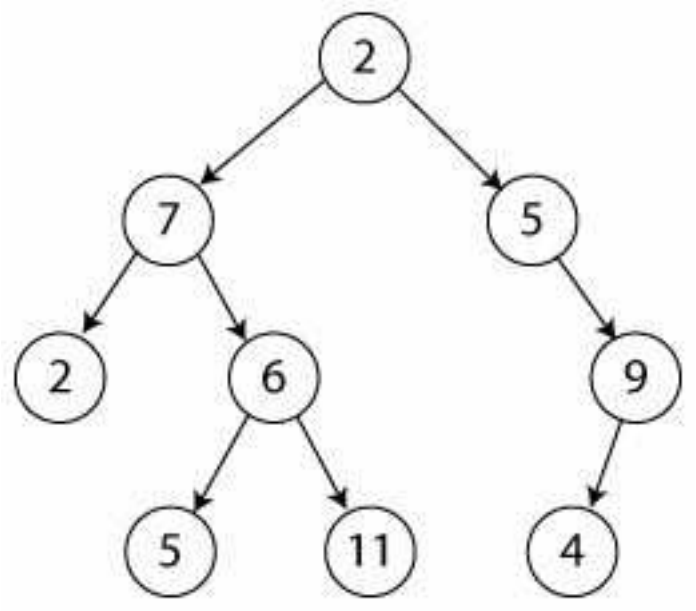
This test has 12 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts only 75 minutes and there are 75 points on the exam. That means you should spend no more than *1 minute per point*.

Don't panic. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious.

If you think there is a syntax error or an ambiguity in a problem's specification, then please ask.

In writing code you do not need to worry about specifying the proper `import` statements. Assume that all libraries and packages we've discussed are imported in any code you write.

PROBLEM 1 : (*Traversals (8 points)*)
Answer the following questions about the binary tree below.



A. What is its *preorder* traversal?

B. What is its *inorder* traversal?

C. What is its *postorder* traversal?

D. Is it height balanced? If not, circle the deepest unbalanced node.

PROBLEM 2 : (*Data structures (8 points)*)

1. Suppose that an array PQ contains a priority queue, represented by a heap (that is, the array represents a tree satisfying the heap property, with the children of the node stored in element i of the array being at positions $2i$ and $2i + 1$). Suppose that there are n **distinct** elements in a binary heap (with the minimum at the root). Which positions (indices in the array) could possibly be occupied by the fourth smallest element? Circle all that apply.

   A. 1

   B. 2 or 3

   C. 4, 5, 6, or 7

   D. 8 through 15

   E. 16 and higher

2. What is the primary reason to use a AVL tree (balanced BST) instead of a binary heap?

   A. Faster average-case delete min

   B. Faster insert

   C. Less space usage

   D. Faster search

   E. Faster worst-case delete min

3. Assume you are reading words from a file and storing them in a data structure. Give a (1) case where a BST would be more appropriate than a trie.

PROBLEM 3 : (*Ordering (8 points)*)

A. Suppose that the following keys, with the given hash values, are inserted into an initially empty table of size 7 using linear-probing hashing.

```
 key:  A  B  C  D  E  F  G
hash:  3  5  3  4  5  6  3
```

Circle the lines on the list below that could not possibly result from inserting these keys, no matter in which order they are inserted.

0	1	2	3	4	5	6
E	F	G	A	C	B	D
C	E	B	G	F	D	A
B	D	F	A	C	E	G
C	G	B	A	D	E	F
F	G	B	D	A	C	E
G	E	C	A	D	B	F

B. Circle the sequences from the following list that produce the same BST as

```
C  E  A  G  B  D  F
```

when inserted in the order given into an initially empty tree.

C	A	B	E	G	D	F
C	A	E	G	D	B	F
C	E	B	G	A	D	F
C	E	G	A	F	D	B
C	E	G	D	F	A	B
E	A	G	B	D	F	C

C. (**Extra credit**) How many of $7! = 5040$ orders in which to insert the keys produce the same tree as

```
C  E  A  G  B  D  F
```

when inserted into an initially empty tree?

PROBLEM 4 : (*Merge (14 points)*)

Assume that the type `ListNode` is defined as on the back page of the exam. The procedure `mergeSet`, described below, is supposed to merge an arbitrary number (M) of sorted lists, as opposed to a simple pair of lists. It is possible to do this in time time $O(N \log M)$, where N is the total number of elements in all lists. It does not suffice to concatenate all the lists together and then sort them, since that will require time $O \log N)$ in the worst case (we assume that we can only compare items on the list, so hashing is out).

A. The following also does not meet the desired time constraint:

```
static ListNode mergeSet(ListNode[] L)
{
    ListNode result = null;
    for (int i = 0; i < L.length; i += 1)
        result = merge2(result, L[i]);
    return result;
}
```

(where `merge2` is below).

```
ListNode merge2(ListNode a, ListNode b)
{
  if (a == null)
    return b;
  if (b == null)
    return a;

  // both non-empty
  int comp = a.info.compareTo(b.info);
  if (comp < 0)
  {
    a.next = merge2(a.next, b);
    return a;
  }
  else
  {
    b.next = merge2(a, b.next);
    return b;
  }
}
```

What is the worst-case time bound on this implementation and why?

B. Fill in the body below to fulfill the comment. Keep your solution at as high a level as possible. You may assume that doubly-linked lists, queues, deques, stacks, hash tables, balanced binary search trees, and priority queues are provided, if any of these are useful to you; again, just state your assumptions. Again, your program must run in $O(N \log M)$ time.

```
/** Assumes that linked lists, L[0],... are sorted.  Returns the result of
 *  merging the linked lists L[0],...,L[M-1] into a single sorted linked list.
 *  The operation may destructive; the original lists may be destroyed. */
static ListNode mergeSet(ListNode[] L)
{
   // FILL IN
```

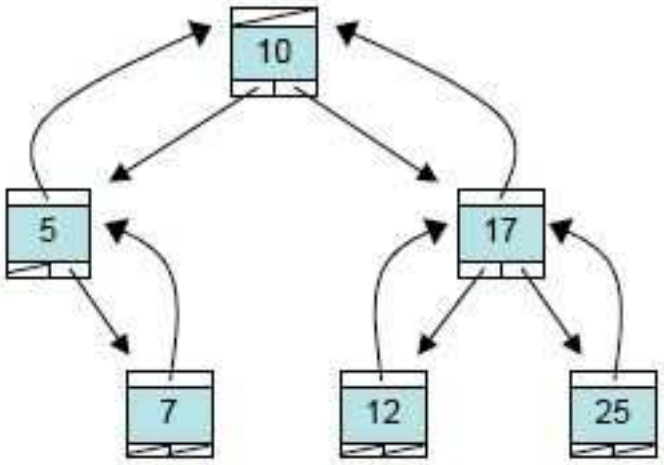
PROBLEM 5 : (*Trees (28 points)*)

A common variation on the standard binary search tree is to add an explicit parent pointer to each node to enable certain operations to be accomplished more efficiently. Consider this node structure that includes a parent pointer:

```
public class BSTNode {
  int info;
  BSTNode parent;
  BSTNode left;
  BSTNode right;

  BSTNode(int e, BSTNode p, BSTNode l, BSTNode r) {
    info = e; parent = p; left = l; right = r;
  }
  BSTNode(int e) {
    this(e, null, null, null);
  }
  public void insert(int x)
  {
    if (x <= info)  {
        if (left == null)
          left = new BSTNode(x, this, null, null);
        else
          left.insert(x);
    } else {
        if (right == null)
          right = new BSTNode(x, this, null, null);
        else
          right.insert(x);
    }
  }
```

A binary tree with parent pointers is shown below (left and right pointers are shown on the bottom of each node, parent pointers are on the top).



A. Write a method `addParentPointers` for the class `BSTNode` that takes a binary search tree with correctly assigned left, right, and info fields and assigns the parent pointers for each node. The parent of the root node should be set to NULL. You may write any helper functions you like.

```
public void addParentPointers()
{
```

B. What is the big-Oh of your solution? Briefly justify.

C. In any collection of elements, the successor of an element is the one immediately following it in increasing order. For example, in the previous tree:

```
5.successor() -> 7
7.successor() -> 10
10.successor() -> 12
25.successor() -> null
```

We would like to implement successor for our `BSTNode` which returns the next BSTNode from the tree in order. For all BSTNodes i, `i.successor().info >= i.info`.

Write a method `successor` to give the successor or the next element in order. You cannot add any new instance variables in addition to the ones already given (`parent, left, right,` and `elem`). Your routine should create no new nodes and it should not change the tree. You can assume that you are given:

```
/* returns the minimum node from tree */
public BSTNode findMin() { ... }
/* returns the maximum node from tree */
public BSTNode findMax() { ... }


public BSTNode successor() {
```

D. With `BSTNode` and `successor`, we can define a class `BSTIter` to give us an `Iterator` in increasing order of the elements of our tree. Fill in BSTIter below.

```
public class BSTIter implements Iterator {
  // reference to current node in iteration
  private BSTNode myCurrent;
  // reference to root of tree
  private BSTNode myRoot;

  /** creates a iterator for BST rooted at root
   */
  BSTIter(BSTNode root) {
    myRoot = root;
    myCurrent = root.findMin()
  }
  /** Tests if this enumeration contains more elements. Returns true if
   *  this iteration has more elements; false otherwise.  */
  public boolean hasNext() {
    // FILL IN




  }
  /* Returns the next node from the tree. Successive
     calls to the nextElement method return successive elements of the
     series. First call to nextElement returns the minimum element in tree.
     Returns null if no more  */
  public Object next() throws NoSuchElementException {
    if (!hasNext())
      throw new NoSuchElementException();
    // FILL IN
```

PROBLEM 6 : (*Vote (9 points)*)

You have been hired by the UN to write election monitor software to be used to keep track of elections in all countries. They provide you with the following class definition.

```
public class Vote {
  ...
  public String getCandidate() {...}  // returns name of candidate voted for
  ...
}
```

They want you to be able to support the following functions as efficiently as possible:

```
public class ElectionMonitor {
  /** Record this vote for the election */
  public void registerVote(Vote v) {...}
  /** Return the current Leader */
  public String getLeader() {...}
  /** Returns a List of candidates in order of votes, from
   *  most to least */
  public List currentResults() {...}
}
```

The UN gives you the following guidelines for your implementation of `ElectionMonitor`.

- The `registerVote` method should work very quickly
- `getLeader` should work very quickly, as people will continually be checking to see who is winning.
- `currentResults` will not be called as often, and hence it is possible that it is slower
- *Do not* assume the number of candidates running is small, or anything else

Explain below (in precise English) how you will design the `ElectionMonitor` class. You can assume the existence of all data structures we discussed in class. You *do not* have to explain how any of the standard methods (e.g. constructing a heap) work. Be specific, however, about which data structures you are using and how these data structures are interconnected. ***Make sure to explain what the running time of the the three methods above will be using your implementation***.

Throughout this test, assume that the following classes and methods are available. These classes are taken directly from the material used in class. There should be no methods you have never seen before here.

ListNode

```
public class ListNode
{
    String info;
    ListNode next;
    ListNode(String s, Node link) {
        info = s;
        next = link;
    }
}
```

String

```
public class String {
    /* Compares this string to the specified object.
     * The result is true if and only if the argument
     * is not null and is a String object that
     * represents the same sequence of characters */
    public boolean equals(Object anObject)

    /* Compares two strings lexicographically.  The
     * result is a negative integer if this String
     * object lexicographically precedes the argument
     * string. The result is a positive integer if
     * this String object lexicographically follows
     * the argument string. The result is zero if the
     * strings are equal */
    public int compareTo(Object o)
}
```

TreeSet/HashSet

```
public class TreeSet {
    // Constructs a new, empty set
    public TreeSet()

    // Returns an iterator over the elements in
    // this set.  The elements are visited in
    // ascending order.
    public Iterator iterator()

    // Returns the number of elements in this set.
    public int size()

    // Returns true if this set contains o
    public boolean contains(Object o)

    // Adds the specified element to this set
    // if it is not already present.
    public boolean add(Object o)
}
```

Iterator

```
public interface Iterator {
    // Returns true if the iteration has more elements.
    boolean hasNext()

    // Returns the next element in the iteration.
    Object next()
}
```

TreeMap/HashMap

```
public class TreeMap {
    // Constructs a new, empty map
    public TreeMap()

    /* Returns the value to which this map maps the
     * specified key.  Returns null is no mapping */
    public Object get(Object key)

    /* Associates the specified value with the
     * specified key in this map.  If the map
     * previously contained a mapping for this key,
     * the old value is replaced. */
    public Object put(Object key, Object value)

    // Returns a Set of the keys contained in this map.
    public Set keySet()

    // Returns the number of key-value mappings in this map
    public int size()
}
```

PriorityQueue

```
public class PriorityQueue
{
    // Constructs abn empty (min) priority queue
    public PriorityQueue()
    /* A new element <code>o</code> is added to the priority queue
     * in O(log n) time where n is the size of the priority queue. */
    public boolean add(Object o)
    /* The smallest/minimal element is removed and returned
     * in O(log n) time where n is the size of the priority queue. */
    public Object remove()
    // returns true if and only if the priority queue is empty
    public boolean isEmpty()
}
```

ArrayList

```
public class ArrayList {
    // Constructs an empty list
    public ArrayList()
    // Returns the number of elements in this list.
    public int size()
    /* Searches for the first occurence of the given
       argument, returns  -1 if not found */
    public int indexOf(Object elem)
    // Returns the element at the specified position
    // in this list.
    public Object get(int index)
    /* Replaces the element at the specified position
     * in this list with the specified element. */
    public Object set(int index, Object element)
    // Appends the specified element to the end of
    // this list.
    public boolean add(Object o)
    // Returns Iterator over elements in list
    public Iterator iterator()
}
```