Name: _____BU ID _____

State here the problem you are eliminating:

# CS 112—Final Exam—Summer Two, 20120

There are 8 problems on the exam. The first and last are mandatory, and **you *must* eliminate any one of problems 2 – 7 by drawing an X through it.** Problem 1 is worth 10 points, and all other problems are worth 15. Please write in pen if possible. Circle answers when they occur in the midst of a bunch of calculations. If you need more room, use the back of the sheet and tell me this on the front sheet.

**Problem One. (True/False – MANDATORY – 10 points)** Write True or False to the left of each statement.

1. A tail-recursive method is one in which the last statement in the method involves a recursive call

2. A open-addressing hash table can never have a load factor larger than 1.0.

3. Merge sort performs the same number of comparisons of keys for all possible inputs.

4. If a linked list contains an sequence of integers in order, then you can use binary search to find whether a particular integer is in the list.

5. If x and y are *any* two keys in leaf nodes in a 2-3 tree, then it will take the same number of comparisons of keys to determine whether x and y are in the tree.

6. Mergesort is difficult to implement "in place" (i.e., without using an extra array).

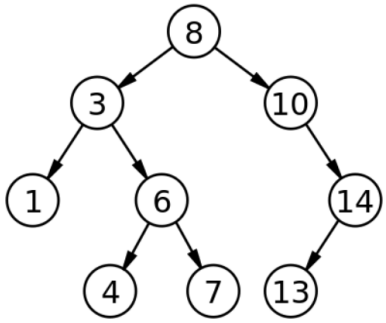
7. If a linear-probing hash table has a load factor of 0.5, then the worst-case lookup time for a key will be O( N ), where N is the number of keys.

8. When performing a *narrowing conversion* you must cast the value to the new type or else you will get an error.

9. In the Boolean expression ( A && B ), if the expression A evaluates to false, then the expression B will not be evaluated.

10. I have put my name on the exam and indicated which problem I will skip.

**Problem Two (Binary Search Trees – 15 points).** This problem has multiple parts, all referring to the diagram of the tree on the right; each part is independent and refers to the original diagram, i.e., start all over again with the original tree diagram for each part.




**(A)** Give a reverse post-order traversal of this tree.

**(B)** What is the worst case time (in term of the number of comparisons) to find a key in this tree? What is the average case (you may use a calculator to figure out the answer).

**(C)** Show the resulting tree after performing the following operations (always delete by moving nodes from the right side of the (sub)tree, do not alternate):

insert(5);

insert(11);

delete(8);

delete(3);

delete(6);

**Problem Three (Hash Tables – 15 points).** For a hash table implemented using the technique of **Linear Probing**, assume that you have an array size of 7 and a hash function

```
int hash(int key) {  return  ( 5 * key % 7 );  }
```

Suppose that you want to perform the following sequence of operations:

```
insert( 2 );

insert( 7 );

insert( 0 );

insert( 1 );

insert( 4);

insert( 6 );

delete( 7 );

delete( 0 );

insert( 11 );
```

(A) Show in the space above the result of performing these operations on an initially-empty table. Use -1 as a sentinel for "never used" and -2 for "used but currently empty."

(B) Recall that a hash function has the form ( P * key % M ) for suitable P and M. What would be the effect on a separate-chaining hash table's performance if P = M // 2, where M is an even number (NOT a prime) ? Be specific about the structure of the table and give the O(..... ) estimated cost for looking up an item in such a table.

(C) Precisely how many slots in the array do you have to examine in each of the following tests for the hash table you got at the end of (A)?

member( 1 )

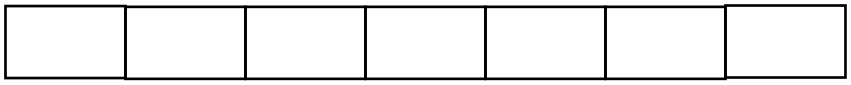
member( 4 )

member( 5 ) [unsuccessful test]

**Problem Four (Max Heaps – 15 points).** Show the Max Heap that would result after the following series of operations; you should work this out by using a tree structure to do the operations, and then (A) fill in the array representation for this final tree, (B) draw your final tree, and (C) answer the general questions below.

1. insert(4);
2. insert (7);
3. insert (2);
4. insert (8);
5. insert (6);
6. n = getMax(); // delete the root and assign its key to n
7. insert(15);
8. m = getMax(); // delete the root and assign its key to m
9. insert(12);

(A)



next:

Note: The array may not be entirely filled by the operations given.

(B) Draw your final tree:

(C) Suppose we count the exact number of comparisons done during this algorithm when we insert a sequence of numbers $n_1$, $n_2$, $n_3$, …, $n_k$. There is a simple way to describe a worst case sequence which produces the maximum number of comparisons. What is it?

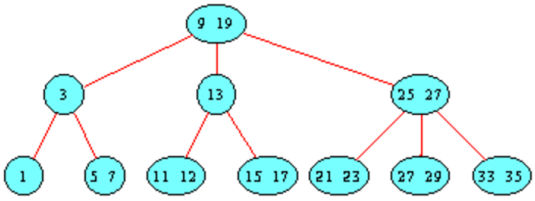
**Problem Five** How many times does the following code print out “X”? Express your answer as a function of N, in terms of Θ(...N...).

```
for( int a = 0; a < N; a = a + 2 ) {

   for( int b = 1; b < 256; b = b * 2 ) {

      for( int c = N; c >= 1; c = c / 2 ){
        System.out.println(“X”);
      }

      for( int m = -N; d < N;  ++m ) {
        System.out.println(“X”);
      }
   }
}
```

**Problem Six (2-3 Trees -- 15 points).** Assume you need to insert the integer keys 1, 2, 3, 4, 5, 10, 9, 8, 7, 6 in that order into an initially-empty 2-3 tree.

1. Show the data structure that would result after inserting these integers (do the work on the next (blank) page and draw the resulting tree on this page).

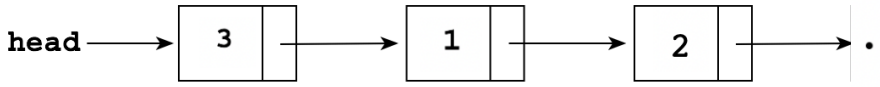
2. Consider the 2-3 tree at the right. Which key in the tree requires the least number of comparisons to find? Which requires the maximum number? Give the keys and the number of comparisons.




3. True or False: The smallest key in a 2-3 tree always occurs at a leaf node.

4. In general what is the maximum number of keys we could to insert into a 2-3 tree to produce a tree whose height is at most 3?

**Problem Seven (Algorithm Trace -- 15 points).** Consider the following linked list:




Show what gets printed out by the following algorithm when applied to this list:

```
void mystery(Node p) {
      if (p == null) {
          ;                  // Do nothing!
      }
      else if (p.item % 2 == 1) {
          mystery(p.next);
          System.out.println(p.item);
          mystery(p.next);
      }
      else {
          mystery(p.next);
          mystery(p.next);
          System.out.println(p.item);
      }
}
```

**Problem Eight. (MANDATORY – 15 points)** Binary heaps are trees which are represented by arrays. However, when discussing the algorithms we simply consider the data structure to be a tree. Suppose we wanted to check if a binary tree with no duplicates has the "heap ordered property" (i.e., each path from the root to a leaf is in descending order). Write a function

```
boolean isHeapOrdered(Node root1) {

      ……

}
```

which takes a binary tree and determines if it has the heap ordered property. Note carefully the following:

- You do NOT need to check if the tree is balanced; all you have to do is check the ordering property;
- You may not use a loop (the function or the helper function must be recursive).
- You may write a helper-function for this if you wish (meaning, the function above is a wrapper around another function).
- The assumed Node class is shown at right.
- You must account for the null pointer.

```
class Node {
   int key;
   Node left;
   Node right;
}
```