ECE242 Data Structures and Algorithms

Fall 2008

2nd Midterm Examination

(120 Minutes, closed book)

Name: __SOLUTION_SET_____

Student ID: _____

Question	Score
1 (10)	
2 (20)	
3 (25)	
4 (10)	
5 (15)	
6 (20)	

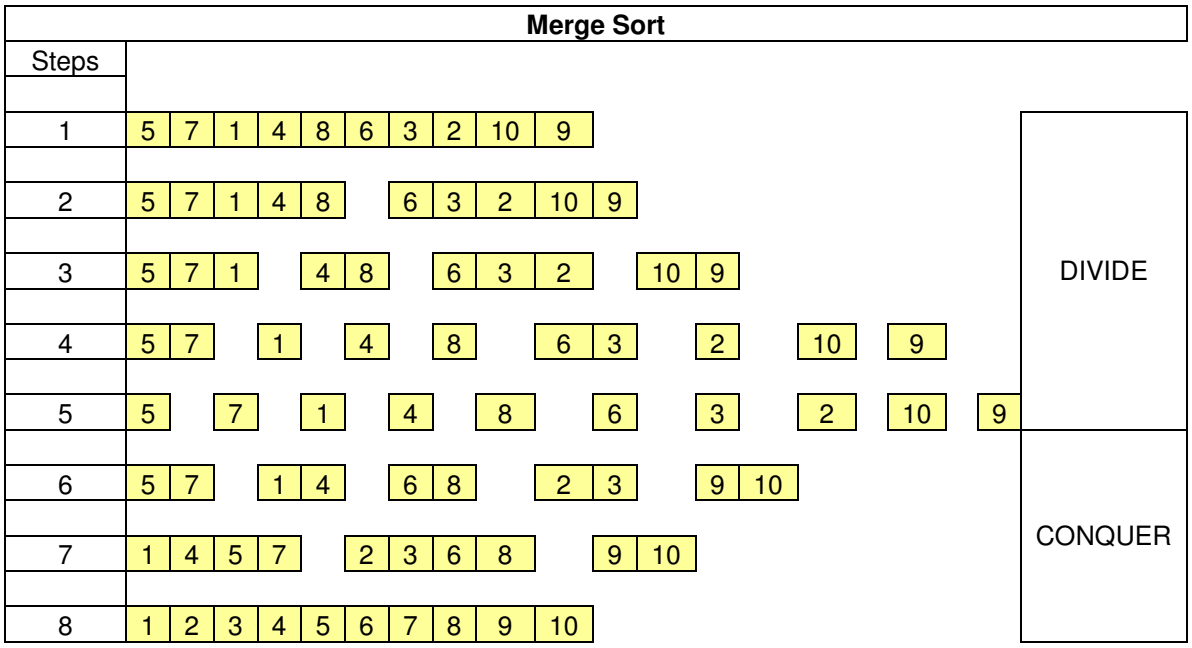
NOTE: Any questions on writing code must be answered in Java using data structures topics covered in lectures 1 - 26.

1. [10pt] Consider the below sequence of integers:

5 7 1 4 8 6 3 2 10 9

Show the steps involved when a **merge** sort algorithm is run on the above sequence of integers to sort them in ascending order (smallest value on the left). *Note: You do not need to write any Java code, however, you need to show each step in the sorting process to receive full credit.*

Solution:



2. [20pt]

a) [17pt] Consider two unsorted arrays, **ArrayA** and **ArrayB** that store **Na** and **Nb** *non-repeating* integers, respectively. Note that some of the integers in ArrayA *could* be present in ArrayB as well. Write a Java method **mergeSortAB** that merges ArrayA and ArrayB into a single **sorted** list, **ArrayC**, such that ArrayC contains integers that are in ArrayA *or* ArrayB but *not* in both. In other words, the ArrayC will ***not*** contain any duplicate elements. *Note: You may create any additional methods you like.*

Hint: The mergeSortAB method is exactly the same as the mergeSort method, except with an added feature that eliminates duplicates while merging.

```
/*
 * Algorithm:
 * 1. Sort ArrayA and ArrayB individually using mergeSortAB method
 * 2. Concatenate the ArrayA and ArrayB to form ArrayC
 * 3. Call the mergeAB() method to merge the two sorted arrays, viz.
ArrayA and
 * ArrayB such that duplicates are removed.
 */
public class arrayMergeAB {
      // Initialize size of arrays
      static final int Na = 5;
      static final int Nb = 5;
      // A counter to keep track of total number of non-duplicate
elements
      int curr_datalen;
      //Constructor
      arrayMergeAB() {
      }
      /*
       * Class Methods
       */
      // mergeSortAB: divide and conquer, with duplicate elimination
during merge phase
    private void mergeSortAB (int [] data, int start, int end)
    {
      // Divide until only single elements are left
        if( start < end)
        {
            int mid = (start+end)/2; // Find an approximate mid of the
array
            mergeSortAB(data, start, mid); // Divide the left half
            mergeSortAB(data, mid+1, end); // Divide the right half
            mergeAB(data, start, mid, end);// Start merging the sorted
parts,
```

```
                                                                // with
duplicate elimination
        }
    }

    // mergeAB: conquer with duplicate elimination
    private int mergeAB(int [] data, int start, int mid, int end)
    {
      boolean eq_flag = false;
        int firstCopied = 0, secondCopied = 0, index = 0, length = end-
start +1;
        // Pointers in the left and right parts to be merged
        int [] temp = new int[end-start +1];
        int firstSize = mid-start+1;
        int secondSize = end-mid;
        /*
         * Start with the first element in each part and update the
pointer when
         * the element is merged into the temp array
         */
        while(firstCopied < firstSize && secondCopied < secondSize)
            if (data[start+firstCopied] < data[mid+1+secondCopied])
            {
                temp [index++] = data[start+firstCopied];
                firstCopied++; // This is a smaller element, move to
the next one
            }
            else if (data[start+firstCopied] > data[mid+1+secondCopied])
            {
                temp[index++] = data[mid+1+secondCopied];
                secondCopied++; // This is a smaller element, move to
the next one
            }
            else if (data[start+firstCopied] ==
data[mid+1+secondCopied])
            {
                  temp [index++] = data[start+firstCopied];
                firstCopied++; // Duplicates; move to the next one in
both sets
                secondCopied++;
                length--; // Due to duplicate, total elements is one
less
            }
        // Copy the remaining elements in first set to the temp; they
are sorted
        while(firstCopied < firstSize)
        {
            temp [index++] = data[start+firstCopied];
            firstCopied++;
        }
        // Copy the remaining elements in second set to the temp; they
are sorted
        while(secondCopied < secondSize)
        {
            temp[index++] = data[mid+1+secondCopied];
            secondCopied++;
        }
```

```
        // Update the sorted parts into the original data array
        for(int i=0; i<length; i++)
            data[start+i]=temp[i];
        // Return updated length (after elimination of duplicates)
        return length;
    }

    // MAIN method to verify the mergeSortAB method
    public static void main(String args[])
    {
      // Initialize ArrayA and ArrayB
      int ArrayB [] = {5,4,1,2,6,12};
      int ArrayA [] = {5,1,12};

      int [] ArrayC = new int [(ArrayA.length + ArrayB.length)];
      // Instantiate the arrayMergeAB class
      arrayMergeAB ab = new arrayMergeAB();

      // call mergeSort(...); for sorting ArrayA
        int start = 0;
        int end = ArrayA.length-1;
        ab.mergeSortAB(ArrayA, start, end); // O(nlogn)

        // call mergeSort(...); for sorting ArrayB
        start = 0;
        end = ArrayB.length-1;
        ab.mergeSortAB(ArrayB, start, end); // O(nlogn)

        // Concatenate sorted ArrayA and sorted ArrayB into ArrayC
      int j=0;
      for (int i=0; i<ArrayA.length;i++) {
            ArrayC[j]= ArrayA[i]; //O(n)
            j++;
      }

      for (int i=0; i<ArrayB.length;i++) {
            ArrayC[j]= ArrayB[i];  //O(n)
            j++;
      }
      // Print out the values to verify
      for (int i=0; i<ArrayC.length;i++) {
            System.out.println(ArrayC[i]);
      }
      // Call mergeAB to merge ArrayA and ArrayB w/o duplicates
      // len gives us the total number of distinct elements in ArrayA +
ArrayB
      int len = ab.mergeAB(ArrayC, 0, (ArrayA.length-1),
(ArrayC.length-1)); // O(nlogn)
      // Print out the results
      System.out.println("\n");
        System.out.print("{ ");
        for(int i = 0; i < len; i++)
            System.out.print(ArrayC[i]+" ");
        System.out.println(" }");
    }
}
```

b) [3pt] What is the running time of the new merge sort algorithm created in (a)?

Solution:

$O(n\log_2 n)$

-> similar to mergeSort except that there is an extra constant time comparison to eliminate duplicates

3. [25pt] An unsorted list of 8 elements is given in the following array:

A[0….7] =	15	3	9	31	11	17	7	23

Note: You do not need to write any Java code in this question, however, you need to show each step in the sorting process to receive full credit.

a) [10pt] Apply the insertion sort algorithm to sort the above array.

Insertion sort									
Steps	Temp								
1	3	15	3	9	31	11	17	7	23
2	3	15	15	9	31	11	17	7	23
3	--	3	15	9	31	11	17	7	23
4	9	3	15	9	31	11	17	7	23
5	9	3	15	15	31	11	17	7	23

6	--	3	9	15	31	11	17	7	23
7	31	3	9	15	31	11	17	7	23
8	--	3	9	15	31	11	17	7	23
9	11	3	9	15	31	11	17	7	23
10	11	3	9	15	31	31	17	7	23
11	11	3	9	15	15	31	17	7	23
12	--	3	9	11	15	31	17	7	23
13	17	3	9	11	15	31	17	7	23
14	17	3	9	11	15	31	31	7	23
15	--	3	9	11	15	17	31	7	23
16	7	3	9	11	15	17	31	7	23
17	7	3	9	11	15	17	31	31	23
18	7	3	9	11	15	17	17	31	23
19	7	3	9	11	11	15	17	31	23
20	7	3	9	9	11	15	17	31	23
21	--	3	7	9	11	15	17	31	23
22	23	3	7	9	11	15	17	31	23
23	23	3	7	9	11	15	17	31	31
24	--	3	7	9	11	15	17	23	31

b) [10pt] Apply recursive quick sort algorithm to sort the above array. *Assume that the pivot is always the last array element for quick sort.*

Solution:

Quick sort (Last element)

Steps								
1	15	3	9	31	11	17	7	23
2	15	3	9	11	17	7	23	31
3	15	3	9	11	17	7	23	31
4	3	7	15	9	11	17	23	31
5	3	7	15	9	11	17	23	31
6	3	7	15	9	11	17	23	31
7	3	7	15	9	11	17	23	31
8	3	7	9	11	15	17	23	31
9	3	7	9	11	15	17	23	31
10	3	7	9	11	15	17	23	31
11	3	7	9	11	15	17	23	31
12	3	7	9	11	15	17	23	31
13	3	7	9	11	15	17	23	31

c) [5pt] What are the big-O running times of insertion sort and quick sort in terms of input data size *n*? Provide a brief explanation for full credit.

4. [10pts] Demonstrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be *h(k) = k mod 9*.

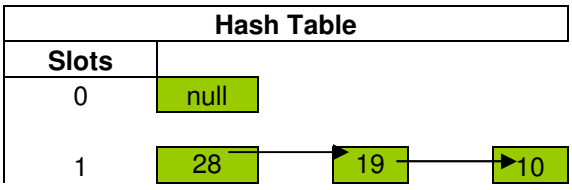
Note: You do not need to write any Java code, however, you need to show each step in the insertion process to receive full credit.

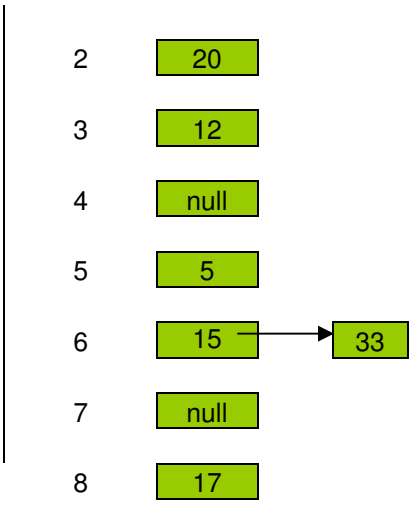
Solution:

First let us calculate the index based on the hash function for each key.

k	h(k) = k mod 9
5	5
28	1
19	1
15	6
20	2
33	6
12	3
17	8
10	1

Now the keys can be inserted into the hash table as follows. Note that the collisions are resolved using chaining.





5. [15pt] You are given a binary search tree (BST) consisting of **N** nodes that store integers as elements. Assume that you have access to the **BSTNode** and the **BST** class with the following methods.

Code for BSTNode

```
private class BSTNode
{
    private int id;
    private BSTNode left;
    private BSTNode right;

    public BSTNode(int key, BSTNode l, BSTNode r) {
        id = key;
        left = l;
        right = r;
    }

}
```

Methods for BST

```
public class BST implements BSTinterface
{
    public void insert(int id);
```

```
    public void remove(int id);

    public void search(int id);

    public void printall();

    public int height();

    public void show();
}
```

a) [6pt] Write a method **findMin** for the class BST above that returns the element with the minimum value in the BST.

```
    // Q 5a. findMin element in the tree
    public int findMin(){
            BSTNode node = root;
            while (node.left!=null){
                  node = node.left;
            }
            return node.id;
      }
```

b) [6pt] Write a method **findMax** for the class BST above that returns the element with the maximum value in the BST.

```
// Q 5b. findMax element in the tree
    public int findMax(){
            BSTNode node = root;
            while (node.right!=null){
                  node = node.right;
            }
            return node.id;
      }
```

c) [3pt] Analyze the running times of **findMin** and **findMax** methods for the best case and the worst case. Explain your answer with a picture.

Solution:

Let n be the number of nodes in the given binary tree.

Since the given tree is not necessarily a complete (perfect) binary tree, the worst case running times of both are O(n)

Assuming perfect binary tree, we get the best case running time as $O(\log_2 n)$

6. [20pt]

a) [3pt] If a perfect (complete) binary tree has n leaves and all levels are fully populated, how many nodes does the tree have in terms of n?

Solution:

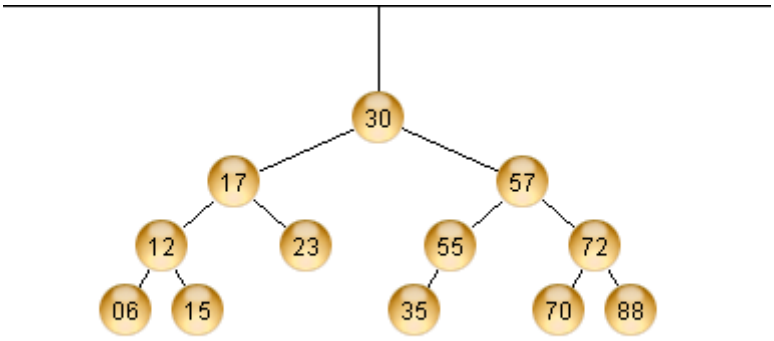
2n-1 nodes

b) [3pt] What is the maximum number of nodes in a binary tree of height h in terms of h?

Solution:

$2^{h+1} - 1$

c) [10pt] Consider the following binary search tree (BST)



i) [3pt] Perform pre-order, inorder and post order traversal on the BST shown in Figure 1 and write the output sequence of each traversal.

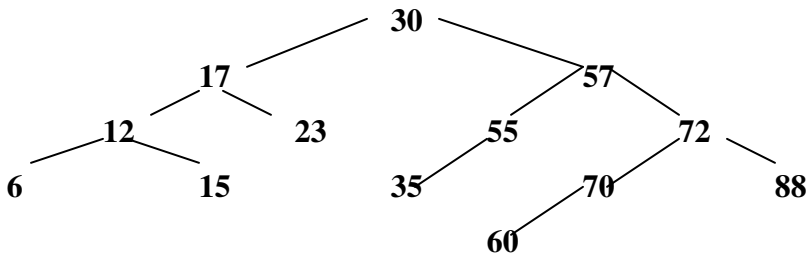
Solution:

Pre-order traversal: 30, 17, 12, 6, 15, 23, 57, 55, 35, 72, 70, 88

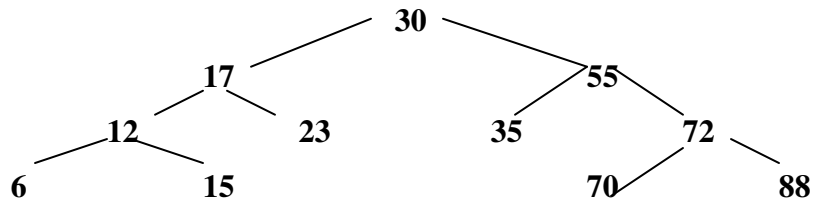
Inorder traversal: 6, 12, 15, 17, 23, 30, 35, 55, 57, 70, 72, 88

Post-order traversal: 6, 15, 12, 23, 17, 35, 55, 70, 88, 72, 57, 30

ii) [3pt] In the BST shown in Figure 1, insert a new node with value 60. Show the resulting tree and all intermediate steps.



iii) [4pt] In the BST shown in Figure 1, remove the node 57 and show the resulting tree using one of the two methods used in lecture. Show all steps.



d) [4pt] Bob claims, "the order in which a fixed set of elements are inserted into a binary search tree does not matter; the same tree results every time". Is Bob correct? If yes, explain why. If not explain why not. *You may use a simple example to prove your answer.*

Solution:

No. First, the root changes every time the insertion is started with a new order. Second, binary search tree is constructed such that the element that is less than a node is supposed to be placed to the left of it and element that is more than the node is supposed to be placed to the right. Hence, if you have a different order, then the tree will be different.