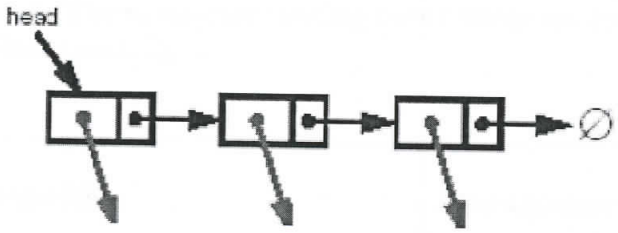
Nothing written on the back of this page will be marked.

1 – Arrays and Linked Lists (5 pts)

Consider a set of ***n*** distinct integers stored in either of the following data structures:

1. an array of size N ($N > n$)
2. a singly linked list whose front is kept track by a pointer "head" as shown below



Give the running time of each of the following methods (operations) in terms of the parameter ***n*** and the big-Oh notation using the above two data structures. Efficiency is important in all operations.

- **insertion**(k): insert a non-duplicate integer k into the data structure.
- **deletion**(k): remove integer k from the data structure. Return false if k is not found in the data structure. Return true if the deletion operation is successful.
- **search**(k): return true if integer k is in the data structure. Return false if k is not found.
- **find_min**( ): return the smallest integer stored in the data structure.
- **find_max**( ): return the largest integer stored in the data structure.

Fill in the following table. ***NO*** *explanation is required.* A sorted array or linked list is always kept sorted in increasing order. We also maintain a variable (attribute) ***size*** that records the current number of elements in the array or linked list.

Operation	**Sorted array**	**Unsorted array**	**Sorted linked list**	**Unsorted linked list**
insertion				
deletion				
search				
find_min				
find_max				

Nothing written on the back of this page will be marked.

2 – Sorting Algorithms (4 pts)

Fill in the following table with the appropriate running time of each sorting algorithm in each case using the big-Oh notation. ***NO*** *explanation is required.*

Note: You may assume that the input is randomly mixed and has no special property in an average case. The worst-case running time usually results from the input having a special property (e.g., almost sorted).

Algorithm	**Average-case running time**	**Worst-case running time**
Insertion Sort		
Quick Sort		
Merge Sort		
In-place Heap Sort		

Nothing written on the back of this page will be marked.

3 – Binary Tree Traversal (3 pts)

Consider a binary tree T having nodes with keys A, B, C, D, E, F, G, H and I.
A ***postorder*** traversal visits the nodes in the order F I C B D H G A E
An ***inorder*** traversal visits the nodes in the order F D B C I E H A G

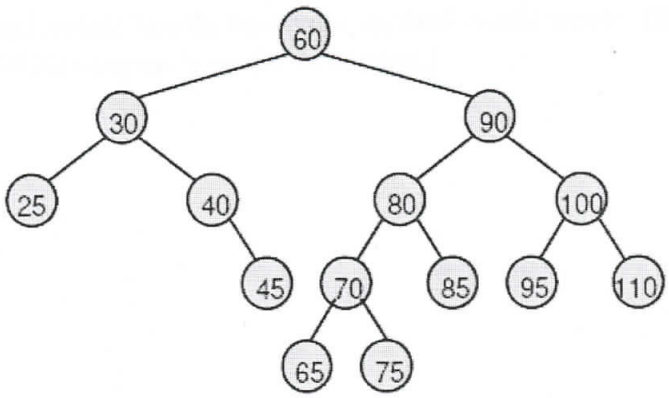
(a) Draw the binary tree T.

(b) Briefly explain the strategy you used to construct the tree (for partial marks in case your answer in (a) is wrong).

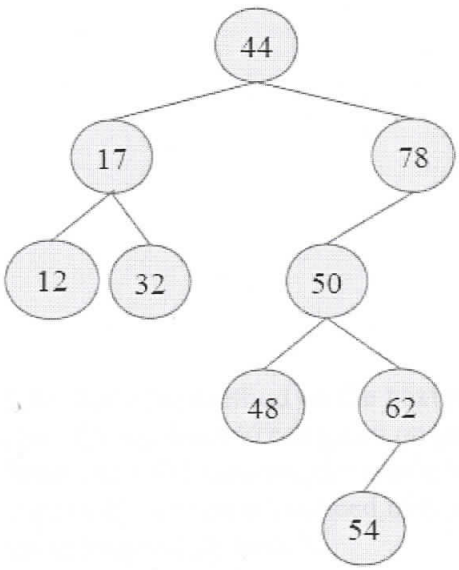
Nothing written on the back of this page will be marked.

4 – Removal Operation of Binary Search Trees (2 pts)

(a) Consider the following BST with integer keys. Redraw the tree as it would look if the node with key 90 were removed. (Note: The external dummy nodes are not shown in the diagram.)



(b) Consider the following BST with integer keys. Redraw the tree as it would look if the node with key 78 were removed. (Note: The external dummy nodes are not shown in the diagram.)



Nothing written on the back of this page will be marked.

5 – Binary Search Tree Algorithms (3 pts)

Assume that you have a binary search tree T containing no duplicate integer keys and having at least 2 nodes. Write a (non-recursive) method named ***findSecondSmallest***( ) that returns the ***second smallest element*** in the BST. For example, if a BST contains the following integers {10, 25, 35, 100, 200, 350} then the method returns integer 25.

(a) Explain briefly how your method would work. Diagrams of different cases might be useful here. (Write your code on the next page.)

5(b) Write the method on the next page. We implement external **dummy** nodes in this problem. Use class *BinarySearchTree* given on page 19 in your implementation.
Note: Do NOT traverse the whole tree to find the second smallest element (i.e., the exhaustive approach). Given a balanced tree, your algorithm must run in O(log***n***) time. No point will be given for the exhaustive approach**.**

Nothing written on the back of this page will be marked.

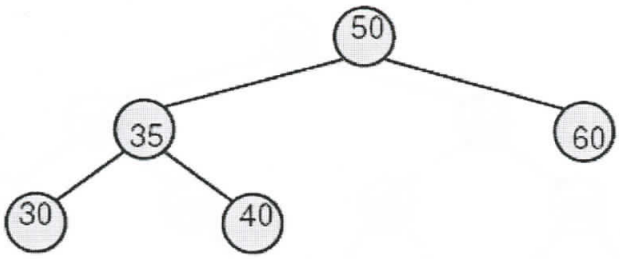
5(b) **Provide comments** in your code to facilitate code reading and understanding.

```
public int findSecondSmallest( )
{
```

Nothing written on the back of this page will be marked.

6(a) – AVL Trees – Insertion (3 pts)

Consider the following AVL tree with integer keys. Redraw the tree as it would look if a node with key **42** and then another node with key **32** were inserted into the tree.
Note: Draw the tree after each insertion to get points for each operation. If tree restructuring is needed, show the involved nodes z, y, x (or the 7 components if using the cut-and-link method) for partial marks in case the final answer is wrong. The external dummy nodes are not shown in the diagram.

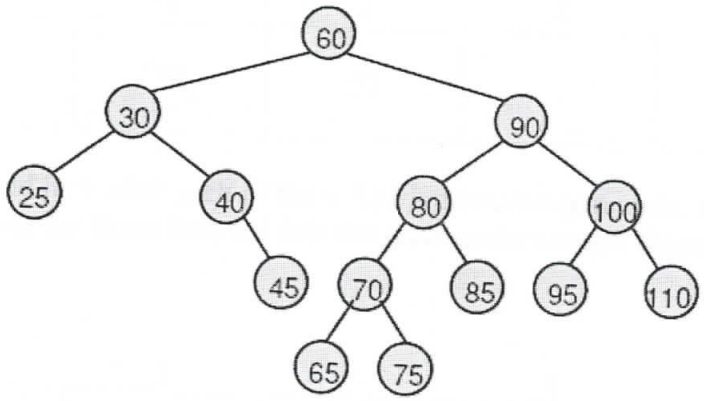


Nothing written on the back of this page will be marked.

6(b) – AVL Trees – Removal (3 pts)

Consider the following AVL tree with integer keys. Redraw the tree as it would look if the node with key **25** were removed.

Note: If tree restructuring is needed, show the involved nodes z, y, x (or the 7 components if using the cut-and-link method) for partial marks in case the final answer is wrong. The external dummy nodes are not shown in the diagram.



Nothing written on the back of this page will be marked.

7(a) – Hashing – Quadratic Probing (3 pts)

Consider a hash table of length 7 and **quadratic probing**. The hash table stores integer keys. The hash function is given by $h(k) = k \bmod 7$.

Below is the table as it looks after the commands *insert*(7), *insert*(52), and *insert*(30).

0	1	2	3	4	5	6
7		30	52			

Draw the table as it would look after each of the following *insert* commands. (Note that the *insert* operations are cumulative; the items inserted into the table so far are still there, so show them too.)

insert(16)

0	1	2	3	4	5	6
7		30	52			

insert(80)

0	1	2	3	4	5	6
7		30	52			

insert(45)

0	1	2	3	4	5	6
7		30	52			

Nothing written on the back of this page will be marked.

7(b) – Double Hashing (3 pts)

Consider a hash table of length 7 and **double hashing**. The hash table stores integer keys. The primary hash function is given by $h(k) = k \bmod 7$.
The secondary hash function is given by $d(k) = 5 - k \bmod 5$.

Below is the table as it looks after the commands *insert*(59), *insert*(14), and *insert*(37).

0	1	2	3	4	5	6
14		37	59			

Draw the table as it would look after each of the following *insert* commands. (Note that the *insert* operations are cumulative; the items inserted into the table so far are still there, so show them too.)

insert(28)

0	1	2	3	4	5	6
14		37	59			

insert(44)

0	1	2	3	4	5	6
14		37	59			

insert(38)

0	1	2	3	4	5	6
14		37	59			

Nothing written on the back of this page will be marked.

8 (a) – Heap Operations – Insertion (2 pts)

The following array contains a ***min heap*** with integer keys. Nothing is stored at index 0, so we mark it with an X.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	14	15	18	25	19	27	20	26	35	24	22				

Give the final version of the array as it would look like after ***insert***(**16**) followed by ***insert***(**10**).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X															

In the space below, draw the heap in the form of a binary tree after each insertion for partial marks in case your final answer above is wrong.

Nothing written on the back of this page will be marked.

8(b) – Heaps – *deleteMin( )* (2 pts)

The following array contains a ***min heap*** with integer keys. Nothing is stored at index 0, so we mark it with an X.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	14	15	18	25	19	27	20	26	35	24	22	28			

Give the final version of the array as it would look like after **two** (2) ***deleteMin***( ) operations.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X															

In the space below, draw the heap in the form of a binary tree after each *deleteMin*( ) operation for partial marks in case your final answer above is wrong.

Nothing written on the back of this page will be marked.

8(b) – Heap Construction (2 pts)

Consider a *non-heap* binary tree with integer keys stored in the following array. Nothing is stored at index 0, so we mark it with an X.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	20	17	18	35	15	21	23	26	25	14	22	16	19	37	30

We call method ***buildHeap***( ) to construct a ***min heap*** from the above tree using a bottom-up construction with log ***n*** phases. The final heap is stored in the same array. (*Note*: ***buildHeap***( ) is used in the in-place heap sort algorithm, but we are not interested in sorting the above array. We simply want to build a heap from the binary tree represented by the above array.)

(i) Show the heap in the form of a binary tree after each phase.

(ii) Show the content of the array after ***buildHeap***( ) terminates.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X															

Nothing written on the back of this page will be marked.

9 – Heap Properties (4 pts)

Consider a **min** heap having unique keys and stored in array A. There are 10,000 entries, stored in A[1] … A[10000] . Circle TRUE or FALES for each of the following statements and give a brief explanation or proof. *No point is given without an explanation/proof.*

(i) A[4000] > A[3999] TRUE FALSE

Explanation/proof:

(ii) A[4000] > A[125] TRUE FALSE

Explanation/proof:

(iii) In a postorder traversal of the heap (as a binary tree), the last node visited contains the smallest value. TRUE FALSE

Explanation/proof:

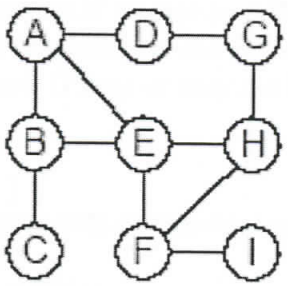
(iv) In a preorder traversal of the heap (as a binary tree), the last node visited contains the largest value. TRUE FALSE

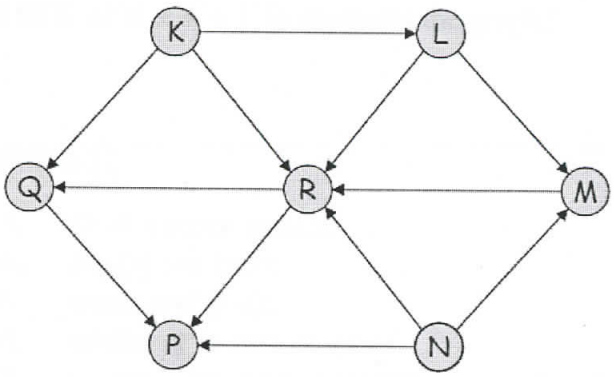
Explanation/proof:

Nothing written on the back of this page will be marked.

10 – Graph Traversal (4.5 pts)

Consider the following graphs.





(a) Given the undirected graph on the left, list the vertices in the order they might be encountered in a ***depth first search*** (DFS) starting at vertex A.

(b) Given the undirected graph on the left, list the vertices in the order they might be encountered in a ***breadth first search*** (BFS) starting at vertex A.

[redacted]

Nothing written on the back of this page will be marked.

11 – Graph Algorithms – Breadth First Search (2.5 pts)

Consider the ***breadth first search*** algorithm shown below. Given an undirected graph ***G*** having ***V*** vertices and ***E*** edges, what is the running time of the ***BFSearch***( ) algorithm in terms of ***V*** and ***E*** and the big-Oh notation if we use an **ADJACENCY MATRIX** of size ***V*** **x** ***V*** to represent the graph? Show detailed calculations and explanations.

Note: The graph ***G*** may or may not be connected.

```
BFSearch( G ) {
i = 1;    /* i represents component number */
for every vertex v
        flag[v] = false;
for every vertex v
        if ( flag[v] == false ) {
          print ( "Component " +  i++ );
          BFS( v );
        }
}
```

```
BFS( s ) {

3.   Q = empty queue;
4.   flag[s] := true;
5.   enqueue(Q, s);
6.   while Q is not empty
7.      do v := dequeue(Q);    output ( v );
8.         for each w adjacent to v
9.            do if flag[w] = false
10.                 then flag[w] := true;
11.                      enqueue(Q, w)
}
```

```
public class BinarySearchTree
{
    /* Basic node stored in binary search trees */
    private static class BinaryNode
    {
        /* Constructors */
        BinaryNode( int theElement )
        {
            this( theElement, null, null, null );
        }

        BinaryNode( int theElement, BinaryNode lt, BinaryNode rt,  BinaryNode pt)
        {
            element  = theElement;
            left     = lt;
            right    = rt;
            parent   = pt;
        }

        int element;       // The data in the node
        BinaryNode left;   // Left child
        BinaryNode right;  // Right child
        BinaryNode parent; // Parent
    }

    /** The tree root */
    private BinaryNode root;

    /**
     * Construct the tree.
     */
    public BinarySearchTree( )
    {
        root = null;
    }

    /**
     * Insert an integer x into the tree
     */
    public void insert( int x )
    {
        // This method is used to build the binary search tree.
        // We are not concerned with the implmentation of method insert().
    }

    /**
     * Return true if node p is an external (dummy) node; return false otherwise.
     */
    private boolean isExternal( BinaryNode p )
    {
        return ( ( p.left == null ) && ( p.right == null ) );
    }

    /**
     * Return the second smallest integer in the BST.
     */
    public int findSecondSmallest(  )
    {
        // WRITE YOUR CODE IN THE EXAM BOOKLET, NOT HERE.
    }
}
```