Name ____

Computer Science S-111
Practice Final Examination

This exam consists of three parts. Part I has 15 multiple-choice questions, worth 3 points each. Answer all of them by marking the letter corresponding to the best answer in the boxes below. Part II consists of 6 problems, of which you must complete 4. Part III consists of 3 problems that you must complete. Show your work clearly in answering Parts II and III so that partial credit may be awarded if your reasoning is partially correct.

You have three hours to complete the exam. The questions are worth a total of 155 points. In order to properly budget your time, plan on spending approximately **one minute per point**. Budgeting your time in this way should leave you extra time to spend on the harder problems and/or to check your work.

You may use **a single 3-inch x 5-inch sheet** with handwritten notes on both sides. **Put your name on it and turn it in with your exam**. Please put away all other materials, and turn off all electronic devices (including cell phones) and put them away. **Do all your work on the exam itself.** Good luck!

Problem	Max. Score	SCORE
I	45	
II-1	15	
II-2	15	
II-3	15	
II-4	15	
II-5	15	
II-6	15	
III-1	10	
III-2	15	
III-3	25	
TOTAL	**155**	

Answers to Part I

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

Part I. Multiple-choice (3 pts. each). Specify up to two answers.

Write your answers in the table at the bottom of the cover sheet. Your first choice for a particular question should be written directly below the question number in the row labeled "first choice." If you decide to provide a second choice for a particular question, write it directly below your first choice in the row labeled "second choice." You will get one point for any problem where your first choice is wrong but your second choice is correct.

1. What will be the value of b after the following section of code executes:

```
int a = 4, b = 0;
if (a > 5) {
    b = 4;
} else if (a < 10) {
    b = 3;
} else if (a < 5) {
    b = 2;
} else {
    b = 1;
}
```

   A. 1
   B. 2
   C. 3
   D. 4
   E. none of these

2. Consider the following recursive method:

```
public static int mystery(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return 2 * mystery(n - 1);
    }
}
```

   What value will be returned by the call `mystery(3)`?

   A. 1
   B. 2
   C. 4
   D. 8
   E. no value, because there will be infinite recursion

3. Given the `Vehicle` classes from lecture, which of the following assignment statements (if any) would ***not*** compile? (Assume that appropriate parameters replace the … in each line.)

   A. `TractorTrailer t = new Truck(…);`
   B. `Automobile a = new Taxi(…);`
   C. `Vehicle v = new MovingVan(…);`
   D. `Truck t = new MovingVan(…);`
   E. all of the above assignment statements would compile

4. Consider the following code fragment:

```
int i = 0;
while(i < 10) {
    System.out.print(i);
    i++;
}
```

   Which of the following fragments *does not* produce the same output?

   A.
```
int i = 0;
do {
    System.out.print(i);
    i++;
} while(i < 10);
```

   B.
```
int i = -1;
while (i < 10) {
    i++;
    System.out.print(i);
}
```

   C.
```
for (int i = 0; i <= 9; i++) {
    System.out.print(i);
}
```

   D.
```
for (int i = 0; i < 10; i++) {
    System.out.print(i);
}
```

   E. none of the above; they all produce the same output

5. With a poorly chosen hash function, it is possible to have a situation in which the search time in a hash table of n items goes to

   A. $O(n)$
   B. $O(n!)$
   C. $O(\log n)$
   D. $O(n^2)$
   E. $O(1)$

6. A binary tree is constructed of nodes that are instances of the following class:

```
public class Node {
    public int val;
    public Node left;
    public Node right;
}
```

Consider the following method:

```
public static Node mystery(Node root) {
    if (root.right == null) {
        return root;
    } else {
        return mystery(root.right);
    }
}
```

You consult three supposedly tech-savvy consultants, and you get the following three opinions about what the method does when passed a reference to the root node of a binary tree:

I. It returns the last node visited by an inorder traversal
II. It returns the last node visited by a postorder traversal
III. It returns the last node visited by a level-order traversal

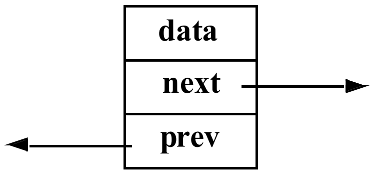
Which of these opinions is correct regardless of the contents of the tree?

A. I only
B. II only
C. III only
D. I and III
E. II and III

7. A police department wants to maintain a database of up to 1800 license-plate numbers of people who receive frequent tickets so that it can be determined very quickly whether or not a given license plate is in the database. Speed of response is very important; efficient use of memory is also important, but not as important as speed of response. Which of the following data structures would be most appropriate for this task?

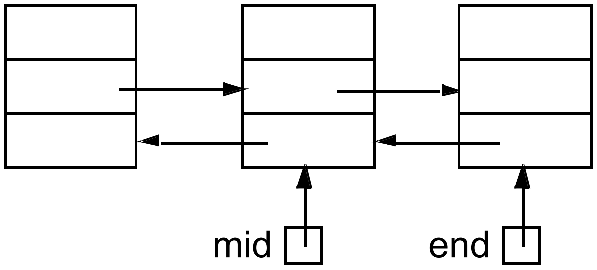
A. a sorted linked list
B. a sorted array with 1800 entries
C. a balanced search tree
D. a hash table using open addressing with 1800 entries
E. a hash table using open addressing with 3600 entries

8. Nodes for a doubly linked list are defined to have the following structure:



The `next` instance variable stores a reference to the next node in the list, and the `prev` instance variable refers to the previous node in the list.

Below is a list of three of these nodes, along with two reference variables, `mid` and `end`, that refer to specific nodes in the list.



Which of the following expressions does *not* refer to the first node in the list?

A. mid.prev
B. end.prev.prev
C. mid.next.next.prev
D. mid.next.prev.prev
E. end.prev.prev.next.prev

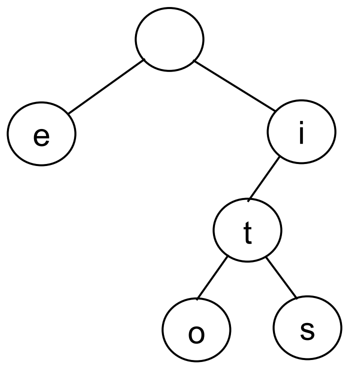
9. Which of the following implementation(s) of a data dictionary of n items could (at least in some cases) require O(n) operations to search for an item?

I. a binary search tree
II. a 2-3 tree
III. a hash table

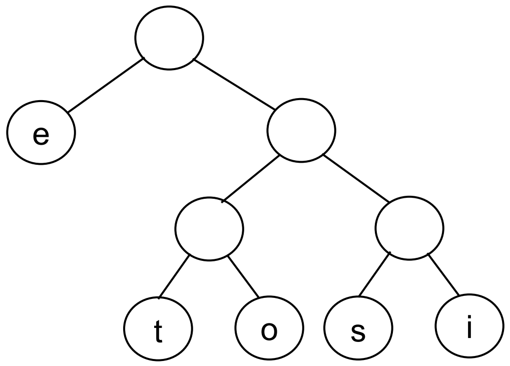
A. I only
B. II only
C. III only
D. I and III, but not II
E. I, II, and III

10. A Huffman tree is constructed for a text document containing 5 characters. The character ‘e’ appears most frequently, and the character ‘i’ has the next highest frequency. Which of the following could be the Huffman tree for this document?

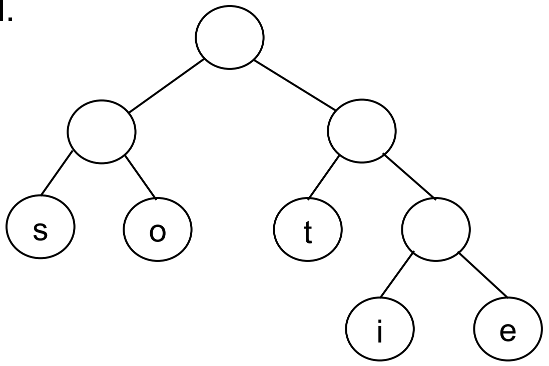
I.



II.



III.



A. I only
B. II only
C. III only
D. either II or III
E. none of these

11. The following array is to be sorted in ascending order:

```
25   47   14   33   15   74   52
```

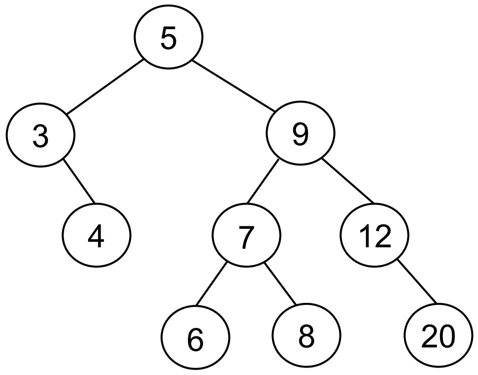
Which algorithm will cause the array to be ordered

```
52   47   25   33   15   14   74
```

at an intermediate stage in the sorting process?

A. bubble sort
B. heap sort
C. insertion sort
D. quicksort (initial pivot = 33)
E. selection sort

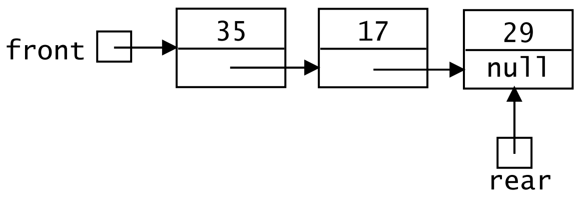
12. The binary search tree shown below was constructed by inserting a sequence of items into an empty tree.



Which of the following input sequences will *not* produce this binary search tree?

A.	5	3	4	9	12	7	8	6	20
B.	5	9	3	7	6	8	4	12	20
C.	5	9	7	8	6	12	20	3	4
D.	5	9	7	3	8	12	6	4	20
E.	5	9	3	6	7	8	4	12	20

13. The diagram below suggests how we could implement a double-ended linked list, in which we maintain a reference to both the first and last nodes in the linked list.



Which one of the following operations would be inefficient to carry out when there are a large number of elements in the linked list?

A. insertion at the end to which `front` refers
B. insertion at the end to which `rear` refers
C. deletion from the end to which `front` refers
D. deletion from the end to which `rear` refers
E. test for an empty linked list

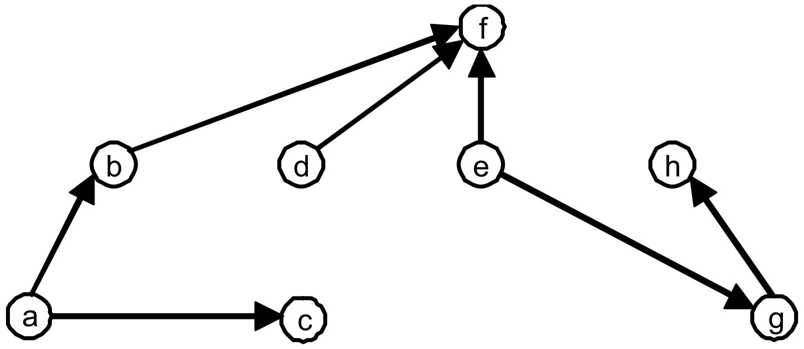
14. The following method operates on a heap containing n strings.

```
public static void mystery(Heap h) {
    for (int i = 0; i < 5; i++) {
        String s = (String)h.remove();
        System.out.println(s);
    }
}
```

What is the big-O time complexity of this method in terms of n?

A. *O*(n)
B. *O*(log n)
C. *O*(n log n)
D. *O*(n^2 log n)
E. *O*(n^2)

15. Given the graph shown below



Which of the following is *not* a topological ordering of the vertices?

A.	a	b	c	d	e	f	g	h
B.	a	c	b	f	d	e	g	h
C.	a	c	d	b	e	g	f	h
D.	a	b	d	e	g	f	h	c
E.	d	e	a	c	b	f	g	h

PART II: Answer <u>four</u> of the six questions in the space provided.

You are welcome to try all six problems, but make sure to <u>clearly indicate which problems you want us to grade by circling the numbers of the problems</u>.

II-1. Sorting *(15 points total)*

Parts a-f (2 points each part). The array below is to be sorted in ascending order.

```
17  53  71  62  36  46  41  23  12
```

a. After the initial partition step of the version of quicksort discussed in lecture, with 36 as the pivot, how would the array be ordered?

b. After the initial iteration of radix sort, how would the array be ordered?

c. After the initial iteration of bubble sort, how would the array be ordered?

d. After the initial iteration of selection sort, how would the array be ordered?

Here is another copy of the initial array:

```
17  53  71  62  36  46  41  23  12
```

e. After the heap-creation phase of heap sort, how would the array be ordered? (Your final answer should be an array, not a tree.)

f. After two iterations of insertion sort, how would the array be ordered?

g. *3 points*
Let's say that you need to sort a collection of data that is stored in a singly linked list. Would Shellsort be a good algorithm to adapt for this problem? Why or why not?

II-2. Arrays *(15 points total)*

Parts a and b. Consider the following lines of Java code:

```
int[] a = {5, 4, 3, 2, 1};
int[] b = {5, 4, 3, 2, 1};
int[] c = a;
for (int i = 0; i < b.length; i++) {
    c[i] = b[i];
}
b[3] += b.length;
a[3]--;
System.out.println(a[3] + " " + b[3] + " " + c[3]);
```

a. *3 points*
Draw a single memory diagram that shows the final result of these lines. Include both the stack and the heap in your diagram. You may assume that these lines are part of the main method.

b. *2 points*
Indicate what will be printed by the final line of code shown above.

c. *5 points*
Write a method with the header

```
public static void shiftRight(int[] arr)
```

that takes a reference to an array of integers and shifts all of the array elements one position to the right, with the original last element wrapping around to become the new first element. For example, consider this array:

```
int[] values = {0, 2, 4, 6, 8, 10};
```

After calling `shiftRight(values)`, the contents of the `values` array should be `{10, 0, 2, 4, 6, 8}`. You should not use `System.arraycopy` in your solution.
Put your method at the top of the next page.

d. *5 points*
Write a method with the header

```
    public static int indexOf(int[] arr1, int[] arr2)
```

that takes two arrays of integers and that returns the index of the first occurrence of the first list in the second list, or -1 if the first list does not appear in the second list. For example, suppose that you have these arrays:

```
        list1: {1, 3, 6}
        list2: {1, 3, 5, 8, 12, 1, 3, 17, 1, 3, 6, 9, 1, 3, 6}
```

then the call `indexOf(list1, list2)` should return 8 because the sequence of values stored in `list1` appears in `list2` starting at index 8. Notice that `list1` actually appears twice in `list2`, starting at position 8 and starting at position 12. Your method is to return the first such position. You may assume that both arrays have at least one element.

II-3. Trees *(15 points total)*

a. *4 points*

Suppose the keys on the middle row of a standard keyboard (ASDFGHJKL) are inserted in succession into an initially empty binary search tree. Draw the tree after this sequence of insertions has been made.

b. *4 points*

Given your answer to part a, what would the tree look like after performing the following sequence of operations:

insert B, insert E, remove A, remove D

b. *7 points*
Suppose the keys on the middle row of a standard keyboard (ASDFGHJKL) are inserted in succession into an initially empty 2-3 tree. Draw diagrams to illustrate the growth of the tree, showing the tree just before and just after all splits.

II-4. Hashing *(15 points total; 5 points each part)*
You are given an empty hash table of size 7 that uses open addressing. The following sequence of keys is to be inserted:

15 17 8 23 3 5

Insert these keys using each of the following approaches. If overflow occurs, say so, and indicate the element that causes the overflow.

a. h(x) = x % 7; linear probing

0	
1	
2	
3	
4	
5	
6	

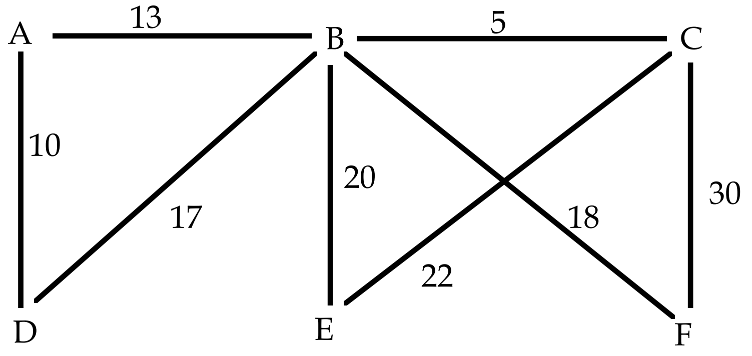
b. h(x) = x % 7; quadratic probing

0	
1	
2	
3	
4	
5	
6	

c. h(x) = x % 7; double hashing with h2(x) = x / 7 + 1 (using integer division)

0	
1	
2	
3	
4	
5	
6	

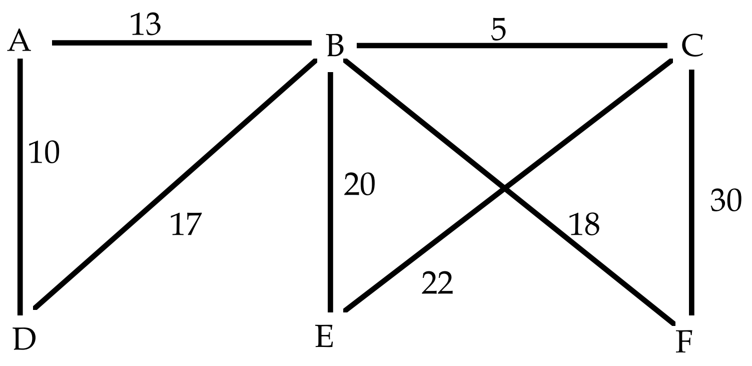
II-5. Graph Algorithms *(15 points total)*



a. *3 points*
Perform a depth-first traversal of the graph shown above, starting with vertex C. Select the smallest edge first when appropriate. In the space below, list the vertices in the order in which they are visited.

b. *3 points*
Perform a breadth-first traversal of the graph shown above, starting with vertex C. Select the smallest edge first when appropriate. In the space below, list the vertices in the order in which they are visited.

c. *4 points*
Suppose that Prim's algorithm has been executed on the graph above, starting from node F, up to the point at which there are four edges selected for inclusion in the minimal spanning tree. List these four edges in the order that they are selected for inclusion, using notation similar to (A, B) to specify an edge.



d. *5 points*
Suppose you are using Dijkstra's algorithm to find the shortest path from vertex D to vertex E in the graph above. List, in the order in which they become known, all vertices to which a shortest path is determined in the process of solving this problem, and the length of the shortest path to each of these vertices.

II-6. Recursive Backtracking (*15 points)*

The Knapsack Problem is another classic problem that can be solved using recursive backtracking. You are given a set of items of different weights, and you need to pack some subset of these items into a knapsack so that the knapsack has a specified total weight known as the *target*. A given solution to the problem is called a *packing*.

For example, let's say that we are given the following array of item weights:

```
int[] weights = { 15, 8, 5, 4 };
```

Here's an illustration of how recursive backtracking could be used to find a packing that gives the knapsack a total weight of 17 pounds:

i	target	packing	comments
0	17	15	15 < 17, so we try it as element 0 in the packing and make a recursive call with target = 17 – 15 = 2
1	2	15, 8	8 > 2, so try the next weight
1	2	15, 5	5 > 2, so try the next weight
1	2	15, 4	4 > 2, and there are no more weights, so backtrack
0	17	8	8 < 17, so we try it as element 0 in the packing and make a recursive call with target = 17 – 8 = 9
1	9	8, 5	5 < 9, so we try it as element 1 in the packing and make a recursive call with target = 9 – 5 = 4
2	4	8, 5, 4	4 == 4, so we try it as element 2 in the packing and make a recursive call with target = 4 – 4 = 0
3	0	8, 5, 4	target == 0, so {8, 5, 4} is a solution!
…			

Notes:
- In a given invocation of the method, we only need to consider elements of the weights array that have an index that is greater than the index of the element added to the packing by the previous invocation of the method.
- We can only add a weight to the packing if it is <= to the target weight.
- We change the target weight when we make a recursive call to reflect the item that was just added to the packing. This allows us to focus on a subproblem that involves only the remaining weights in the array, ignoring the weights that have already been added to the packing.
- We reach a solution when the target equals 0.

You are given a partially completed Java class called Knapsack:

```
public class Knapsack {
    public int[] weights;   // weights of the available items
    public int[] packing;   // current combination of weights
    …
}
```

Below is a partially implemented method of the `Knapsack` class that we will invoke to find all possible packings for a given target weight. The parameter `target` represents the current target weight, and the parameter `i` represents the element of the `packing` array that you are attempting to find a value for (see the example above); it would have the value 0 in the first invocation of this method.

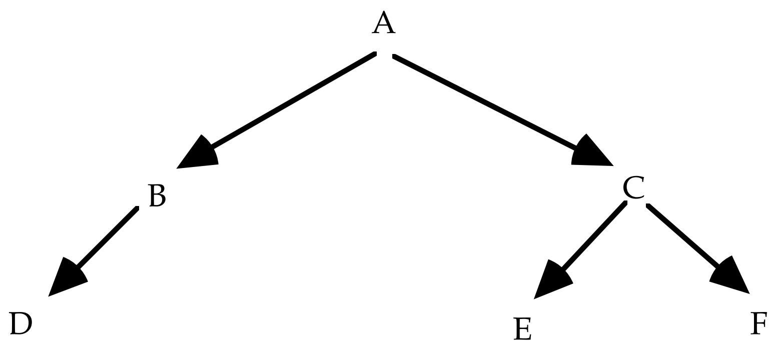
Complete this method so that it uses recursive backtracking to find all possible packings for the specified target. You may find it helpful to add one or more additional parameters. If you do so, please indicate the values that they would be given in the first invocation of the method.

Don't forget that you have access to both the `weights` array, which you can assume has been initialized to the hold the weights of the available items, and to the `packing` array, which you can assume has the same length as the `weights` array.
We strongly encourage you to follow the recursive-backtracking template covered in lecture, although instead of making method calls for things like `applyValue()`, you should put all necessary functionality within the body of the method below.

```
void findPacking(int target, int i,                      ) {
    if (target == 0) {
        // We've found a solution!
        System.out.print("packing = ");
        for (int j = 0; j < packing.length && packing[j] != 0; j++) {
            System.out.print(packing[j] + " ");
        }
        System.out.println();
        return;    // find any additional solutions
    }










}
```

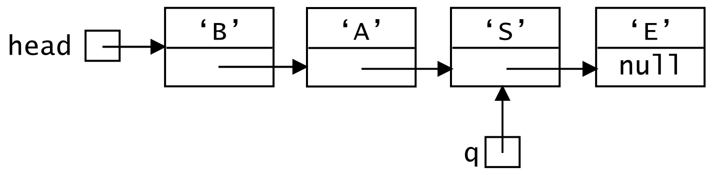
PART III (50 pts total): Complete all of the following problems.

III-1 Using Data Structures ***(10 points total)***
Suppose you want to <u>print a binary tree</u> like the one shown below in **reverse level order**, so that the labels of the nodes of the tree in the diagram would appear in reverse alphabetical order. You may use a **stack**, a **queue**, or both as auxiliary data structures. **Describe an algorithm** that will accomplish the task.



Pseudocode for your reverse-level-order algorithm, in clear, concise English:

III-2. Linked Lists (*15 points total)*
The diagram below shows a linked list of characters, along with two variables that each store a reference to one of the nodes in the list.



The nodes in the linked list could be implemented in Java using the following class:

```
public class CharNode {
    public char val;
    public CharNode next;
}
```

a. *1 point*
The expression `head.next.next` specifies an instance variable in one of the nodes in the diagram. Circle the box in the diagram that represents that instance variable.

b. *1 point*
What is the value of the expression `head.next.next`?

c. *3 points*
Modify the diagram to reflect the results of executing the following lines of code:

```
q = q.next;
head.next = q;
```

In addition, briefly describe below how these lines of code change the linked list.

d. *7 points*
Write a static method with the header

```
public static Node append(Node head, char ch)
```

that appends a node containing the character `ch` to the end of the singly linked list whose first node is referred to by the parameter `head`. A `head` value of `null` represents an empty list. The nodes are instances of the `Node` class from the previous page.

Your method should use iteration, and it should work correctly regardless of the length of the initial list. After appending the new node, it should return a reference to the first node.

e. *3 points*
If we wanted to improve the efficiency of the `append()` method, what additional information would we need to maintain about the linked list? What is the worst-case time efficiency of the `append()` method before this improvement, and what would its efficiency be after? Use big-O notation, and explain briefly how you came up with your answers.

III-3. Binary Trees (*25 points total)*

This problem deals with binary trees that are constructed of nodes that are instances of the following class:

```
public class Node {
    public int key;
    public Object data;
    public Node left;
    public Node right;
}
```

a. *10 points*
The following Java method uses recursion to search for a key in the binary search tree whose root node is referred to by the parameter `root`. If it finds the key, it returns a reference to the corresponding data item. If it doesn't find it, it returns `null`.

```
public static Object search(Node root, int key) {
    if (root == null) {
        return null;
    } else if (key == root.key) {
        return root.data;
    } else if (key < root.key) {
        return searchTree(root.left, key);
    } else {
        return searchTree(root.right, key);
    }
}
```

In the space below, rewrite the `search()` method so that it uses iteration instead of recursion:

b. *10 points*
Write a Java method named `mirror()` that takes a reference to the root node of a binary tree and creates a new tree (with its own nodes) that is the mirror image of the original tree. For example: if `root` is a reference to the root of the tree on the left below, then the return value of `mirror(root)` would be a reference to the root of the tree on the right below.
Hint: This method is much easier to write if you use recursion.

```
    1               1
   / \             / \
  2   3           3   2
 / \                 / \
4   5               5   4
   / \             / \
  6   7           7   6
```

```
public static Node mirror(Node root) {



}
```

c. *5 points*
What is the running time of your implementation of the `mirror()` method? Use big-O notation, and explain your answer briefly.