EECE.3220: Data Structures

Fall 2019

Exam 3 Solution

1. (10 points) ***Recursion***

For the short program shown below, list the output exactly as it will appear on the screen. Be sure to clearly indicate spaces between characters when necessary.

You may use the available space to show your work as well as the output; just be sure to clearly mark where you show the output so that I can easily recognize your final answer.

```
int f1(int v1, int v2) {
   cout << "v1 = " << v1 << ", v2 = " << v2 << '\n';
   if (v1 == 0) {
      cout << "Returning 1\n";
      return 1;
   }
   else if (v2 == 0) {
      cout << "Returning 2\n";
      return 2;
   }
   else if (v1 > v2) {
      cout << "v1 > v2\n";
      return 1 + f1(v1 / 2, v2 / 2);
   }
   else {
      cout << "v1 <= v2\n";
      return 2 + f1(v2 / 3, v1 / 3);
   }
}

int main() {
   cout << "Final return value = " << f1(7, 12) << '\n';
   return 0;
}
```

Solution: Each function call prints two lines of output—the values of v1 & v2, and something based on which condition is true. The final output is printed once all calls return.

There are four recursive function calls, shown in sequence on the next page, with the first four lines representing the initial calls, and the effects of the return statements shown after reaching the base case:

Solution to Question 1 (continued)

main() calls f1(7, 12)

f1(7, 12) → returns 2 + f1(12 / 3, 7 / 3) = 2 + f1(4, 2) → *call f1(4, 2)*

    f1(4, 2) → returns 1 + f1(4 / 2, 2 / 2) = 1 + f1(2, 1) → *call f1(2, 1)*

        f1(2, 1) → returns 1 + f1(2 / 2, 1 / 2) = 1 + f1(1, 0) → *call f1(1, 0)*

            f1(1, 0) returns 2 (since v2 == 0)

        f1(2, 1) → returns 1 + f1(1, 0) = 1 + 2 = 3

    f1(4, 2) → returns 1 + f1(2, 1) = 1 + 3 = 4

f1(7, 12) → returns 2 + f1(4, 2) = 2 + 4 = 6

So, the program output is:

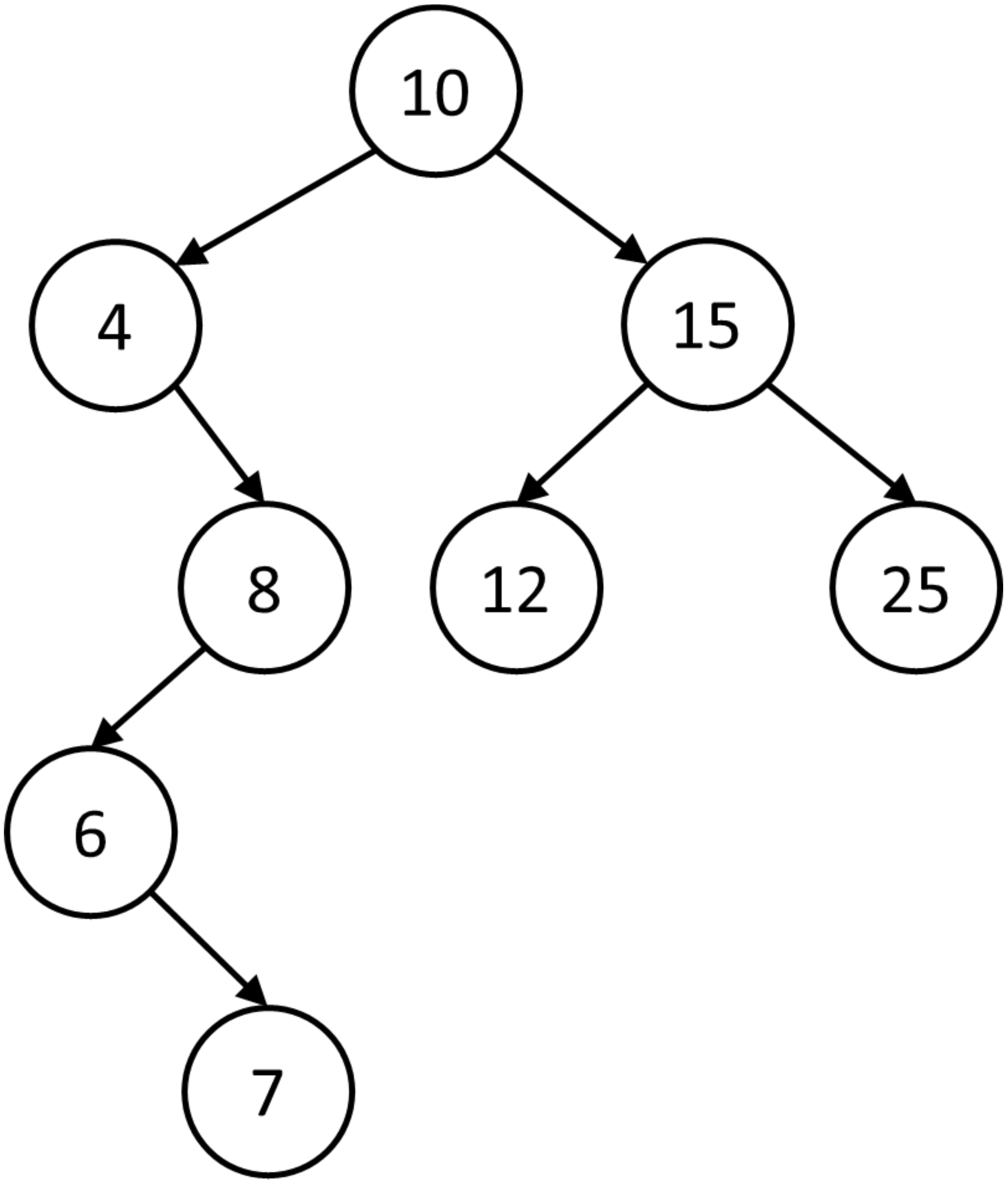
```
v1 = 7, v2 = 12
v1 <= v2
v1 = 4, v2 = 2
v1 > v2
v1 = 2, v2 = 1
v1 > v2
v1 = 1, v2 = 0
Returning 2
Final return value = 6
```

2. (44 points) ***Binary search trees***

a. (12 points) Show the binary search tree built by inserting the integers below in the order shown (beginning with 10). Assume the tree is empty to start.

Integers: 10, 4, 8, 15, 25, 12, 6, 7

Solution: Following the rules that (i) every new value is inserted as a leaf node, (ii) all values in the left subtree of a given node are less than that node's value, and (iii) all values in the right subtree of a given node are greater than that node's value, we get the tree shown below:



2 (continued)

b. (12 points) To print a binary search tree's contents, visit all the nodes and print their contents. The order in which the nodes are visited determines the order of the output values.

In this problem, the top-level function for the tree object (*the BST class defined on the extra sheet*) simply calls a function that takes a node pointer as an argument. That function does the work of visiting all the nodes and printing each of their contents.

The top-level print function is shown. You must write the `printtree()` function, which takes two arguments: an output stream reference, `os`, and a pointer to the root of a given subtree, `st`. `printtree()` prints the data in the node `st` points to and visits its left and right children to do the same.

I strongly suggest writing this function recursively, as an iterative solution is significantly more difficult to write without parent pointers in the tree.

For full credit, write `printtree()` so all tree data are printed in order from lowest to highest. For example, given the data in Question 2a, the function prints: `4 6 7 8 10 12 15 25`

```
void BST::print(ostream &os) {
    printtree(os, root);
    os << "\n";
}
```

Solution: The recursive solution is shown below. The key to printing the tree's contents in order is to remember the relationship between and a node and its subtrees is:

data in left subtree < data in node < data in right subtree

So, the data in the node and its subtrees should be printed in that order.

```
void BST::printtree(ostream &os, BNode *st) {
    if (st == NULL)
        return;
    else {
        printtree(os, st->left);
        os << st->data << " ";
        printtree(os, st->right);
    }

}
```

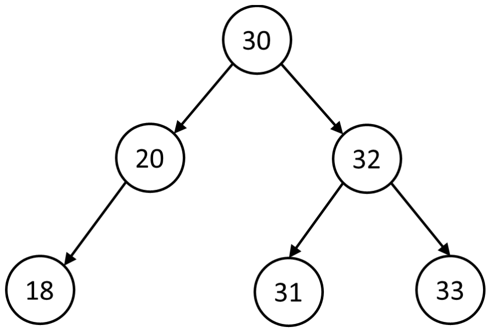
For those who are curious, the most common iterative solution I've found for a BST without parent pointers uses a stack to push node addresses on the way down each path in the tree, then popping those addresses to work your way back up the tree. As in the solution above, the left subtree of each node is visited, then the node itself, then the right subtree.

One such solution can be found at: https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/

2 (continued)

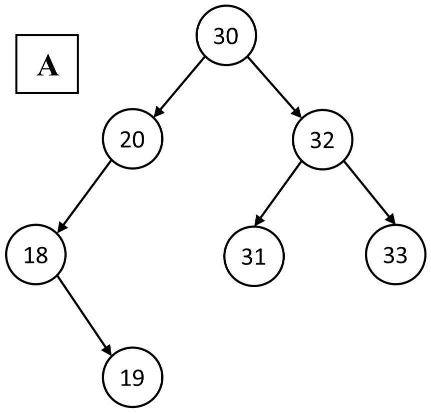
c. (8 points) For each part of this problem, you are given an AVL tree and an operation (add or remove a specific value) to perform. Show the modified tree after the specified operation, which will require at least one rotation to maintain the properties of an AVL tree.

i. Add 19 to the tree below

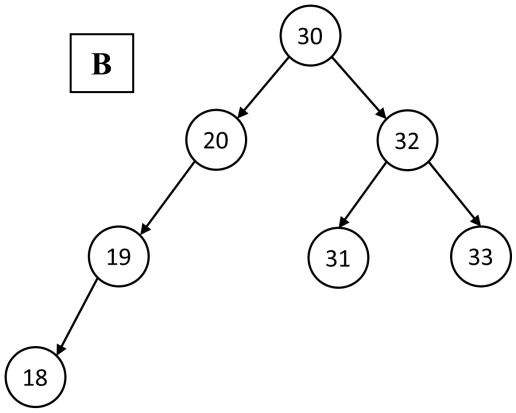


Solution: 19 will be added as the right child of 18, giving the node holding 20 a balance factor of +2 and requiring a rotation (Fig. A). This situation is a "left-right" case, which is fixed by flipping the relationship between 18 and 19 (Fig. B), then rotating the three nodes 18-19-20 such that 19 becomes the root of that subtree, with 18 and 20 as its children (Fig. C):

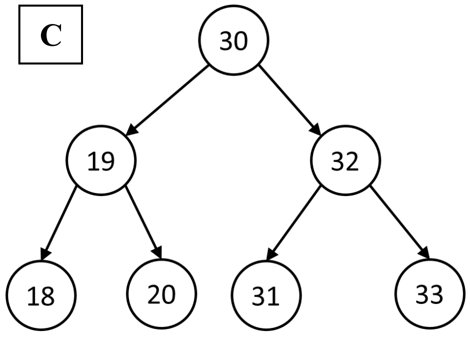
A



B

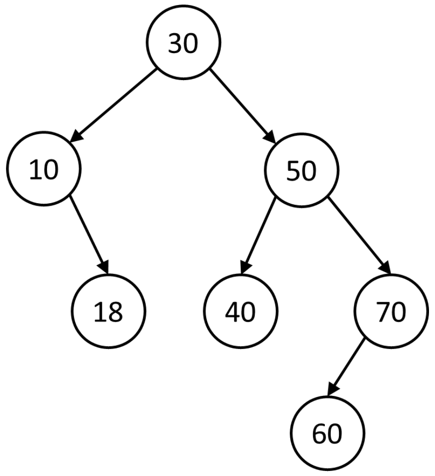


C



2c (continued)

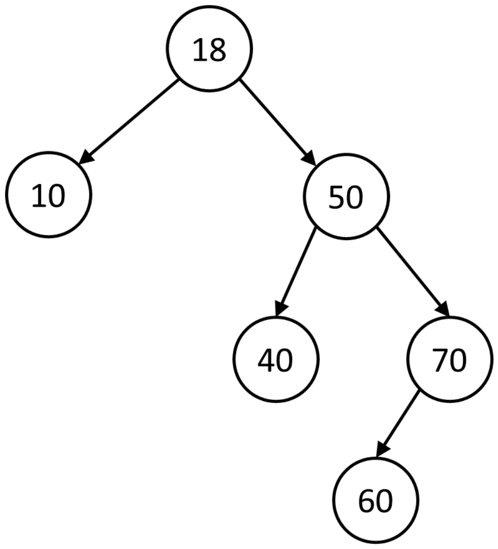
ii. Remove 30 from the tree below



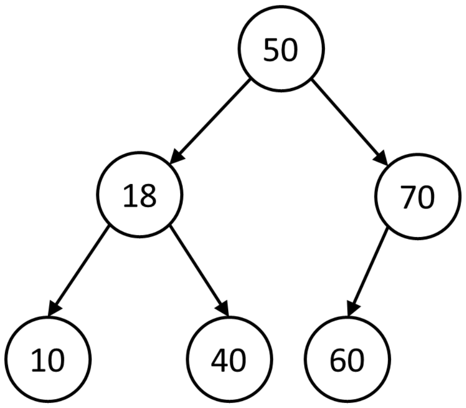
Solution: There are two possible solutions to this problem, depending on which value you use to replace 30 at the root of the tree after removing it.

If you choose the in-order predecessor, 18, that creates a situation in which the root of the tree has a balance factor of -2 (Fig. A). This situation can be resolved by rotating the entire tree to the left, bringing 50 to the root and making 40 the right child of 18 (Fig. B):

A



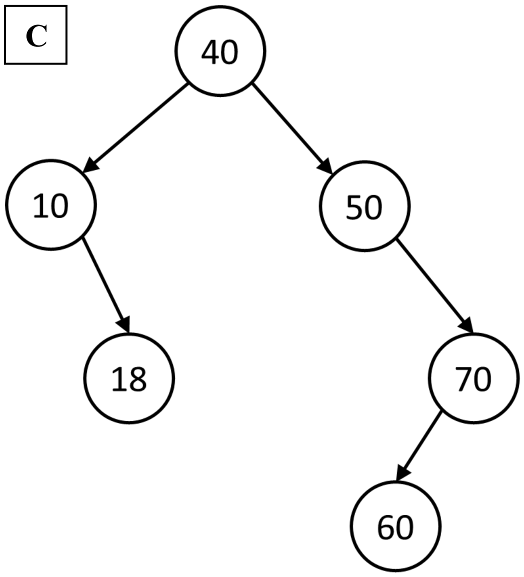
B



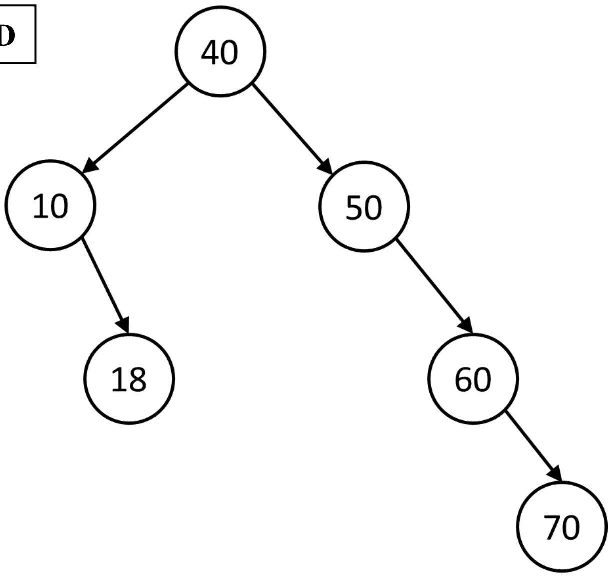
2c-ii (continued)

If you choose to move the in-order successor, 40, to the root of the tree after removing 30, that creates a situation in which the node with value 50 has a balance factor of -2 (Fig. C). That situation can be resolved by flipping the relationship between 60 and 70 (Fig. D), then rotating the three nodes 50-60-70 such that 60 becomes the root of that subtree, with 50 and 70 as its children (Fig. E):

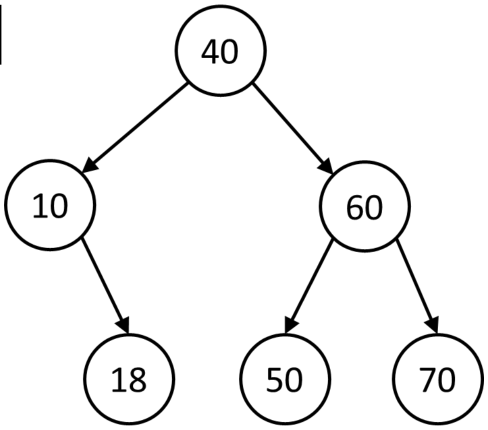
C



D



E



2 (continued)

d. (12 points) The tree below represents a red-black tree, but the colors of only two nodes are shown. The node storing the value 40 is black, while the node storing 70 is red. Use the rules for red-black trees we discussed in class to determine the correct color for each remaining node and write that color on the blank line in the node.

Solution: The red-black tree rules are:

i. The root node is black.
ii. All NULL pointers are black.
iii. If a node is red, both child nodes are black.
iv. Every path from a node to a NULL pointer contains the same number of black nodes.

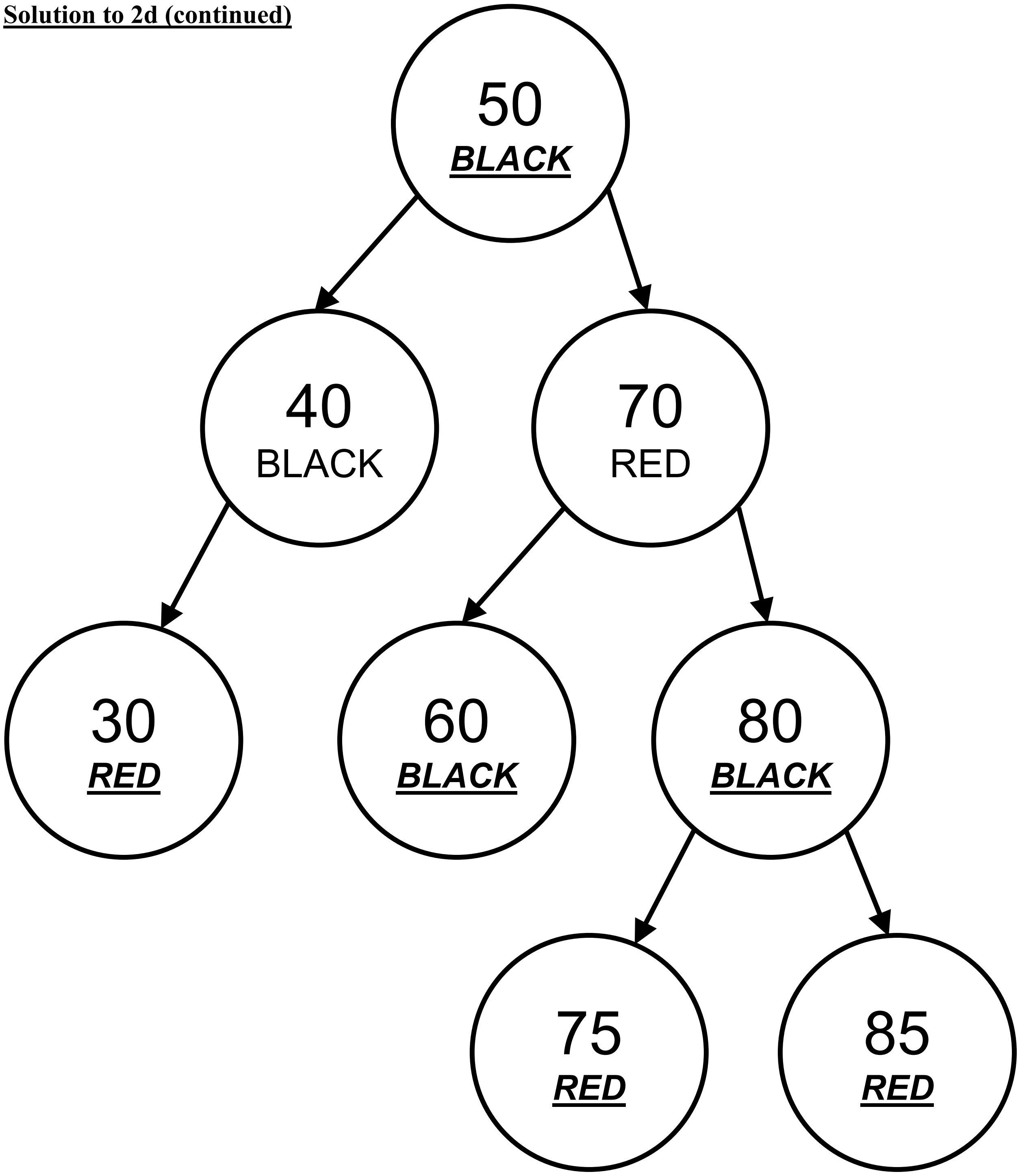
The tree doesn't show NULL pointers, but, since every node should have two children, any "missing" child represents a NULL pointer.

Here's how we determine the colors of every other node in the tree:

- Rule (i) tells us **50 must be in a black node**, since that node is the root.
- Rule (iii) tells us **both 60 and 80 must be in black nodes**, since their parent (70) is red.
- Rule (iv) helps us determine the color of all remaining nodes:
    - It's easiest to look at paths from the root to NULL pointers, particularly the shortest such path.
    - That shortest path goes through two nodes—the root (50) and its left child (40), which has a NULL pointer in place of its right child.
        - Therefore, all paths from the root to NULL pointers must go through two black nodes
    - Looking at the nodes we've already assigned colors, we can see there's two black nodes on every path from the root to a NULL pointer—the root (50) and exactly one of the nodes containing 40, 60, or 80.
        - Therefore, **all remaining values (30, 75, 85) must be in red nodes.**
        - Since each of these nodes is a leaf, both of its children are NULL pointers, which are black (by rule (ii)), so this result satisfies rule (iii) as well.

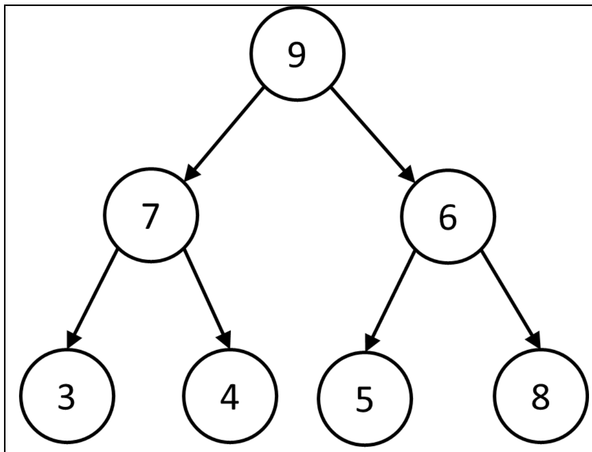
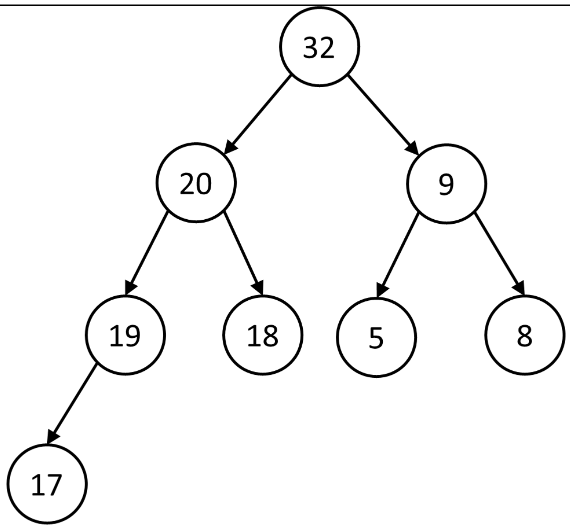
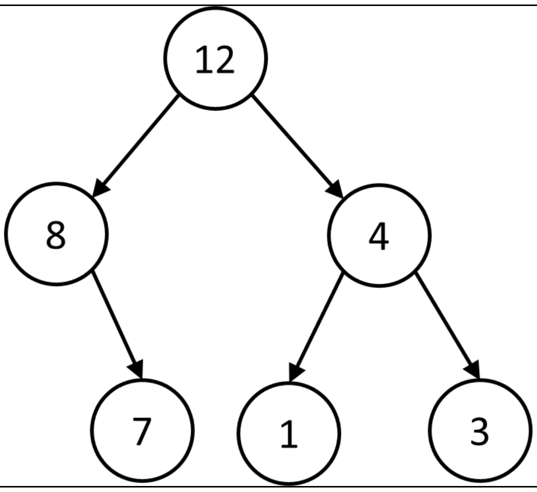
The final tree is shown on the next page.

Solution to 2d (continued)



3. (22 points) ***Heaps***

a. (10 points) For each of the binary trees below, indicate (i) whether it is a max-heap (by circling "Yes" or "No"), and (ii) if not, which heap property it violates.

	(i) Is this tree a max-heap? Yes [✓ ***No***] (ii) If no, which heap property does it violate? *Each node should hold a greater value than its children. Rightmost leaf node (8) has a greater value than its parent (6)*
	(i) Is this tree a max-heap? [✓ ***Yes***] No (ii) If no, which heap property does it violate?
	(i) Is this tree a max-heap? Yes [✓ ***No***] (ii) If no, which heap property does it violate? *Lowest level should be mostly complete. Only missing leaf node(s) should be rightmost node(s) on that level, so 8 should have a left child.*

3 (continued)

b. (12 points) For some heap operations, the number of non-leaf nodes in the heap is a useful piece of information. Use the blank space below to write a function, `countNonLeaves()`, that returns the number of non-leaf nodes in a `Heap` object. Note that:

- The number of non-leaf nodes is a function of the heap size.
- The number of nodes in each complete level is a power of 2.
    - The top level (root) has 1 node, the 2nd level has 2, the 3rd level has 4, and so on.
    - So, a heap with 2 levels has 1 non-leaf node (since the bottom level is composed solely of leaf nodes); a heap with 3 levels has 1 + 2 = 3 non-leaf nodes, and so on.

This function uses the `Heap` class definition shown on the extra sheet provided with the exam. You should assume that this `Heap` class is implemented similarly to the heaps discussed in lecture. Specifically:

- The root of the heap is stored in the first array location (`heaparr[0]`).
- No array locations are left empty for swap space.

Solution:

```
unsigned Heap::countNonLeaves() {
  unsigned total = 0;         // Running total of non-leaf nodes
  unsigned nNodes = 1;        // Max # of nodes at each level
                              //  Doubles at each level

  /*
     If total + nNodes >= size, current level is the last one,
       so it only contains leaf nodes
     Otherwise, there must be at least one more level, so add
       number of nodes at current level (nNodes) to total,
       then double that value to represent max number of nodes
       at the next level
  */
  while (total + nNodes < size) {
    total = total + nNodes;
    nNodes = nNodes * 2;
  }

  return total;
}
```

4. (24 points, 4 points each) <u>***Multiple choice (heapsort, priority queues, hash tables, C++ containers)***</u>

a. Which of the following statements about heapsort are true? <u>**Circle ALL correct answers—there may be more than one.**</u>

   i. <u>***Heapsort has a worst-case execution time of O(n $\log_2$ (n)).***</u>

   ii. The heapsort algorithm can only be used on arrays whose contents represent a heap before the sorting begins.

   iii. <u>***After running heapsort on an array to sort the array from lowest to highest value, the array's contents represent a min-heap.***</u>

   iv. <u>***The heapify algorithm, which is part of heapsort, has a worst-case execution time of O(n $\log_2$(n))***</u>

b. Which of the following statements about priority queues are true? <u>**Circle ALL correct answers—there may be more than one.**</u>

   i. A priority queue <u>must</u> be implemented using a heap.

   ii. <u>***If a priority queue is implemented using a heap, all priority queue operations can be written in terms of heap operations.***</u>

   iii. The contents of a priority queue <u>must</u> be ordered from highest priority to lowest priority.

   iv. <u>***Storing data in a priority queue allows you to access the highest-priority item in the queue in a constant amount of time (O(1)).***</u>

4 (continued)

c. Say you have a hash table containing 10 buckets, and you want to store a total of 20 items in the table. What is the minimum and maximum possible number of collisions (instances in which a new item maps to the same bucket as another item) in this scenario? **<u>This question has exactly one correct answer.</u>**

i. Minimum = 0 collisions; maximum = 10 collisions

ii. Minimum = 0 collisions; maximum = 19 collisions

iii. Minimum = 0 collsions; maximum = 20 collisions

iv. ***<u>Minimum = 10 collisions; maximum = 19 collisions</u>***

v. Minimum = 10 collisions; maximum = 20 collisions

I was lenient with the grading on this question and accepted either choice (iv) or (v) because the problem doesn't specify how collisions are resolved. I intended to write the problem as using a hash table <u>with chaining</u>, which would make (iv) the only correct answer. If you use probing or double hashing to resolve collisions, the maximum number can be greater than 20, since every time you try to choose a new bucket and find it to be full is a collision.

d. Which of the following statements about the C++ sequence containers and sequence container adapters are true? **<u>Circle ALL correct answers—there may be more than one.</u>**

i. All sequence containers must be implemented as linked data structures.

ii. ***<u>The key difference between the `array` and `vector` types is that a `vector` can be resized, while an `array` can not.</u>***

iii. ***<u>The basic `list` can be traversed in either direction, while the `forward_list` can only be traversed from first to last element.</u>***

iv. A sequence container adapter always stores its data in a `vector`, then includes rules about how the data in the vector is accessed to create a stack, queue, or priority queue.

4 (continued)

e. Which of the following statements about the C++ associative containers are true? **Circle ALL correct answers—there may be more than one.**

i. Say you have the same data set stored in two different associative containers—one an ordered container (for example, set or map), and the other one an unordered version of the same type of container. Searching for an item in the ordered container will always be faster than searching for the same item in the unordered container.

ii. ***Ordered associative containers are typically implemented using binary search trees to ensure worst-case search times of O(log n).***

iii. In a map, the data being stored represents the key value used to order that data. In a set, the data being stored is associated with a separate key used to order the data.

iv. ***Unordered associative containers are often implemented using hash tables, where the key is fed into a hash function to determine the bucket used to store the data.***

f. Circle one (or more) of the choices below that you feel best "answers" this "question."

i. "Thanks for the free points."

ii. "This is the best final exam I've taken today."

iii. "At least we're getting this out of the way early in the day."

iv. "I would have preferred to take this on Tuesday."

v. None of the above.

All of the above are "correct" … but (iv) is more correct than the rest.

5. (10 points) ***EXTRA CREDIT***

You are given the postfix expression shown below:

```
5 3 + 6 8 - * 1 3 + /
```

Complete the table below to show how a stack can be used to evaluate this expression, as your Program 3 solution would do. I've filled in all tokens; you show what operation is performed for each token (it's pushed on the stack or it causes data to be popped and a value to be computed) and the stack state once that operation is done. I've completed the first row.

Note that a postfix expression of the form `v1 v2 op` is evaluated as `v1 op v2`. So, for example, `8 2 -` represents the operation `8 - 2`.

Token	**Operation** (if data popped & result computed, list values removed and result of operation)	**Stack state** (show state as list of values with top of stack listed first)
5	5 pushed on stack	5
3	**3 pushed on stack**	**3 5**
+	**3 & 5 popped off stack, 5 + 3 computed; result (8) pushed on stack**	**8**
6	**6 pushed on stack**	**6 8**
8	**8 pushed on stack**	**8 6 8**
–	**8 & 6 popped off stack, 6 – 8 computed; result (-2) pushed on stack**	**-2 8**
*	**-2 & 8 popped off stack, 8 * -2 computed; result (-16) pushed on stack**	**-16**
1	**1 pushed on stack**	**1 -16**
3	**3 pushed on stack**	**3 1 -16**
+	**3 & 1 popped off stack; 1 + 3 computed; result (4) pushed on stack**	**4 -16**
/	**4 & -16 popped off stack; -16 / 4 computed; result (-4) pushed on stack**	**-4**