University of Waterloo
Final Examination
Spring 2017

Student Name:	_____
Student ID Number:	_____
Course Section:	**MTE 140**

Instructors:	Igor Ivkovic
Date of Exam	July 28th 2017
Time Period	7:30pm-10:00pm
Duration of Exam	150 minutes
Pages (including cover)	16 pages
Exam Type	Closed Book

NOTE1: No calculators or other electronic devices are allowed.

NOTE2: The exam questions are of random difficulty levels. If you get stuck on a question, come back to it later. Plan your time wisely to finish the exam on time.

NOTE3: The exam includes two pages of scrap paper at the back. You may separate these pages from the exam. Do not remove any other pages.

Question 1: (8 marks)	Question 2: (12 marks)	Question 3: (7 marks)	Question 4: (10 marks)
Question 5: (7 marks)	Question 6: (5 marks)	Question 7: (6 marks)	Question 8: (10 marks)
Total: (65 marks)			

Useful Formulas

- For $S = a_1 + (a_1 + d) + ... + (a_n - d) + a_n$
  (S is a series that goes from a_1 to a_n in d-size increments),

$$S = n\left(\frac{a_1 + a_n}{2}\right) \tag{1}$$

- $$\sum_{i=k}^{n} 1 = (n - k + 1) \tag{2}$$

- $$\sum_{i=1}^{n} i = \left(\frac{n(n+1)}{2}\right) \tag{3}$$

- $$\sum_{i=1}^{n} i^2 = \left(\frac{n(n+1)(2n+1)}{6}\right) \tag{4}$$

- $$\sum_{i=0}^{n} r^i = \left(\frac{r^{n+1} - 1}{r - 1}\right) \tag{5}$$

- $$\log_b x = \frac{\log_c x}{\log_c b} \tag{6}$$

- $$\log_a a^x = x \tag{7}$$

- $$a^{\log_a x} = x \tag{8}$$

1010100

1 Algorithm Complexity (8 marks)

(approx. 20 minutes)

a. (1 mark)

Provide a code fragment in pseudocode that would exhibit the runtime of $O(log^2(n) + log(log(n)))$. For instance, for O(n) runtime, you could write the following:

```
loop from 1 to n {  call swap(a, b) that performs constant number of steps }
```

Solution:

```
for (int i = 1; i <= n; i = i * 2) {
    for (int j = 1; j <= n; j = j * 2) {
        // constant number of steps
    }
}

for (int k = 2; k <= n; k = k * k) {
    // constant number of steps
}
```

Grading scheme: 0.5 marks for two loops for $O(log^2(n))$, and 0.5 marks for one loop for $O(log(log(n)))$.

b. (7 marks)

The Strassen's method for matrix multiplication can be represented using the following recurrence relationship: $T(1) = a$, $T(n) = 7\ T(\frac{n}{2}) + n^2$. Solve the recurrence relationship by unrolling the recurrence (use back substitution), and give the $T(n)$'s order of growth in terms of the Big-O notation as a function of n. Show all steps in deriving your solution.

Solution:

$$
\begin{aligned}
T(n) = 7T(\frac{n}{2}) + n^2 = 7^2T(\frac{n}{2^2}) + n^2 + 7\frac{n^2}{2^2} = \\
7^3T(\frac{n}{2^3}) + n^2 + 7\frac{n^2}{2^2} + 7^2\frac{n^2}{2^4} = \\
7^4T(\frac{n}{2^4}) + n^2 + 7\frac{n^2}{2^2} + 7^2\frac{n^2}{2^4} + 7^3\frac{n^2}{2^6} = \\
... = 7^iT(\frac{n}{2^i}) + n^2\sum_{j=0}^{i-1}(\frac{7}{2^2})^j = \\
= 7^iT(\frac{n}{2^i}) + n^2\frac{(\frac{7}{4})^i - 1}{\frac{7}{4} - 1} = \\
= 7^iT(\frac{n}{2^i}) + \frac{4}{3}n^2((\frac{7}{4})^i - 1)
\end{aligned}
\tag{9}
$$

When $\frac{n}{2^i} = 1$, let $i = c$. It follows that $\frac{n}{2^c} = 1$ and $n = 2^c$, so $c = \log_2(n)$.

From there,

$$
\begin{aligned}
T(n) = 7^i T(\frac{n}{2^i}) + \frac{4}{3}n^2((\frac{7}{4})^i - 1) = \\
7^{\log_2(n)} T(1) + \frac{4}{3}n^2((\frac{7}{4})^{\log_2(n)} - 1) = \\
7^{\frac{\log_7(n)}{\log_7(2)}} a + \frac{4}{3}n^2(\frac{7^{\log_2(n)}}{2^{2\log_2(n)}} - 1) = \\
n^{\log_2(7)} a + \frac{4}{3}n^2(\frac{7^{\frac{\log_7(n)}{\log_7(2)}}}{n^2} - 1) = \\
n^{\log_2(7)} a + \frac{4}{3}n^{\log_2(7)} - \frac{4}{3}n^2 = O(n^{\log_2(7)}).
\end{aligned}
\tag{10}
$$

Grading scheme: 3 marks for the unfolding to n = 1 case, 1 mark for the computation of $c = \log_2(n)$, and 3 marks for the final derivation of $O(n^{\log_2(7)})$, for the total of 7 marks. $O(n^3)$ is also acceptable as the final answer.

2 Algorithm Design (12 marks)

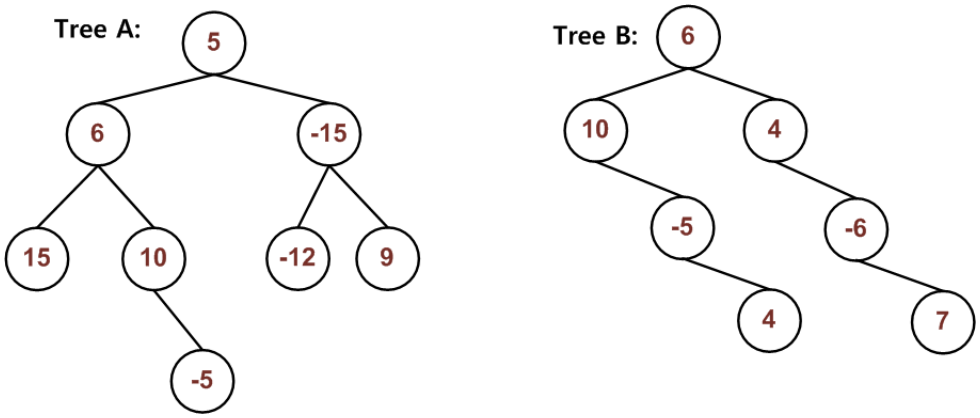
(approx. 35 minutes)

a. (9 marks)

Implement a recursive function named *int findMaxSumOfNodes* that finds a non-cyclical path with the maximum sum in a given binary tree including paths that do not start at the root node. The tree is a general binary tree that stores integers; it is not a heap, BST, or AVL tree. If the tree is empty, the function returns 0.

As input, the function is given: a $BinaryTreeNode * T$, where the function is first called on the root node, and a number of other parameters of your choosing. The function then recursively (1) iterates through the tree levels, and (2) keeps updating the maximum sum if applicable as new paths are discovered. The function returns the maximum sum back to the caller as an integer value; it does not have to output the actual maximum path.

For example, for Tree A below, your function should return 31 for the $(15, 6, 10)$ path. Similarly, for Tree B below, your function should return 21 for the $(10, 6, 4, -6, 7)$ path.



Each *BinaryTreeNode* instance includes as attributes: a *value* of type integer, and pointers to *left* and *right* child nodes. The function has friend access to private members of *BinaryTreeNode*. You may also utilize operations of any of the abstract data types (ADTs) discussed in class, such as List ADT, without implementing the details. You may instantiate multiple copies of the chosen ADT.

You may also write your function in pseudocode without focusing on syntactic details (e.g., you may write "$for\ i\ =\ 0\ to\ (n-1)$"), but you need to specify each step and each function call clearly and unambiguously (e.g., "$l.select(5)$;"). That is, you need to correctly specify data structure instantiations and copying, element access, loop iteration, function calls, and parameter passing; no marks will be awarded for just explanations.

```
int findMaxSumOfNodes(BinaryTreeNode* T,


                                                                    ) {
                                                  // Insert other parameters above
    // Implement your code below
    // Adequately document your code
    // You may use helper functions if needed
```

Solution:

```
int find_max_sum_of_nodes(BinaryTreeNode* T, int &temp_max_sum) {
// exit if T is NULL
if (!T) return 0;

// derive the maximum sum for the left subtree
int left_sum = find_max_sum_of_nodes(T->left, temp_max_sum);
// derive the maximum sum for the right subtree
int right_sum = find_max_sum_of_nodes(T->right, temp_max_sum);

// compare T->value, left_sum + T->value, and right_sum + T->value; store as max1
int max1 = max(T->value, std::max(left_sum, right_sum) + T->value);

// compare max1, left_sum + right_sum + T->value; store as max2
int max2 = std::max(max1, left_sum + right_sum + T->value);

// update temp_max_sum with the new maximum
temp_max_sum = max(temp_max_sum, max2);

// return max1
return max1;
}

int find_max_sum_of_nodes(BinaryTreeNode *T) {
// initialize maximum sum to the minimum INT value
int max_sum = INT_MIN;

// compute the maximum sum for a given tree
find_max_sum_of_nodes(T, max_sum);

// return the found maximum  sum
return max_sum;
}
```

b. (3 marks)

Once you have designed the algorithm, explain in natural language — not code — how would you test the algorithm. That is, list and briefly explain at least six test cases / equivalence classes that you would use to ensure correctness of your algorithm and enhance its robustness. You do not have to modify your code in part (a) to pass these test cases. **Solution:**

```
[Test for T being NULL/empty; 0.5 mark; mandatory]
1. T is NULL, max_sum is 0

[Test for T not empty with the maximum path starting from root; 0.5 mark]
2. T is not empty, max_sum is positive [only 1 path]
3. T is not empty, max_sum is negative [2 or more paths]

[Test for T not empty with the maximum path starting below root; 0.5 mark]
4. T is not empty, max_sum is negative [only 1 path]
5. T is not empty, max_sum is positive [2 or more paths]

[Test for T not empty with the maximum path being a single node; 0.5 mark]
6. T is not empty, max_sum is 0

[Test for T not empty with the maximum path being only subroot + left subtree; 0.5 mark]
7. T is not empty, max_sum is positive

[Test for T not empty with the maximum path being only subroot + right subtree; 0.5 mark]
8. T is not empty, max_sum is negative

Alternative:
[Test for T not empty with the maximum path being subroot + left + right; 0.5 mark]
9. T is not empty, max_sum is 0
```

Grading scheme: Need 6 categories from the above for full marks. Other equivalence classes may be accepted as appropriate.

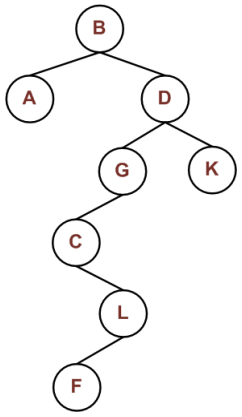
3 Tree Traversals (7 total marks)

(approx. 15 minutes)

a. (3 marks)

A binary tree was processed using post-order and in-order traversals. For post-order, the output derived is [A, F, L, C, G, K, D, B]. For in-order, the output derived is [A, B, C, F, L, G, D, K]. Draw a single binary tree that complies with the traversals above. Briefly explain how you have derived the tree. Hint: Use the character position in each traversal output to help determine its position in the needed binary tree.

Solution:



Use post-order to derive the root nodes from back to front, and use in-order to divide nodes intro subtrees.

Grading scheme: 2 marks for the correct tree, 1 mark for the explanation.

b. (4 marks)

Design a function *bool* $isBinarySearchTree(BinaryTreeNode * T)$ that takes as input the root node of a tree. The function outputs *true* if the given tree is a valid binary search tree (BST), and $false$ otherwise. In a valid BST instance, the BST property holds at every node; also, empty tree is a BST.

Clearly explain your design. You may write the steps for this function in structured English (i.e., high-level pseudocode), where you describe each logical step precisely and unambiguously (e.g., "for each node in the set, apply function X"). If you create helper functions (e.g., "function X"), specify the steps of each function. You are not required to write C/C++ code.

```
bool isBinarySearchTree(BinaryTreeNode* T) {
    // specify algorithm steps in structured English
```

Solution:

```
int isBinarySearchTree(BinaryTreeNode* root)  {
   if (!root) // 1 mark
     return true;

    if (root->left && maxValue(root->left) > root->value) // 1 mark
```

```
        return false;

    if (root->right && minValue(root->right) < root->value) // 1 mark
        return false;

    if (!isBinarySearchTree(root->left) || !isBinarySearchTree(root->right)) // 1 mark
        return false;

    return true;
}
```

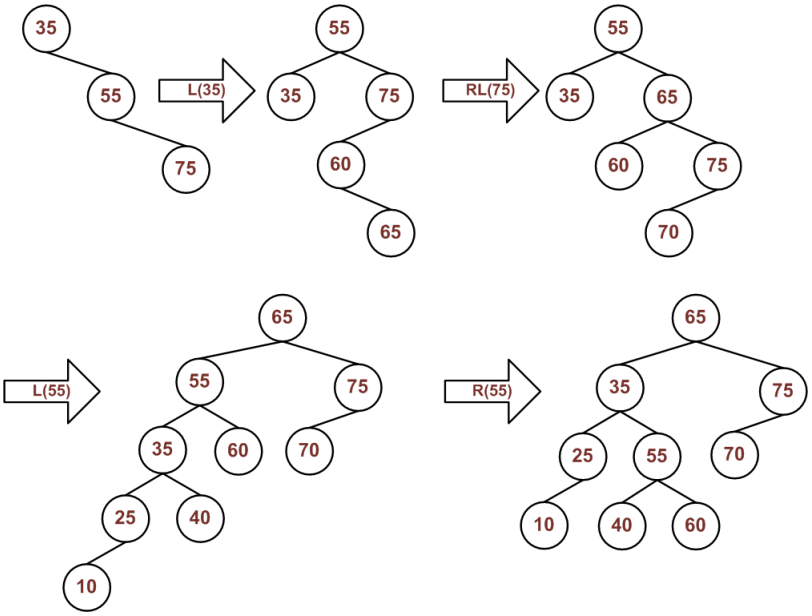
4 AVL Trees (10 total marks)

a. (8 marks)

(approx. 15 minutes)

Given an empty AVL tree, insert the following nodes into the tree:
35 55 75 60 65 70 25 40 10

Show the rotations used in deriving your solution, and write the *avlBalance* values for each node before each rotation. Ensure that the tree remain an AVL tree after each insertion is complete. Check the balancing in a bottom-up manner, by finding the first node starting from the bottom for which the $|avlBalance| > 1$. **Solution:**

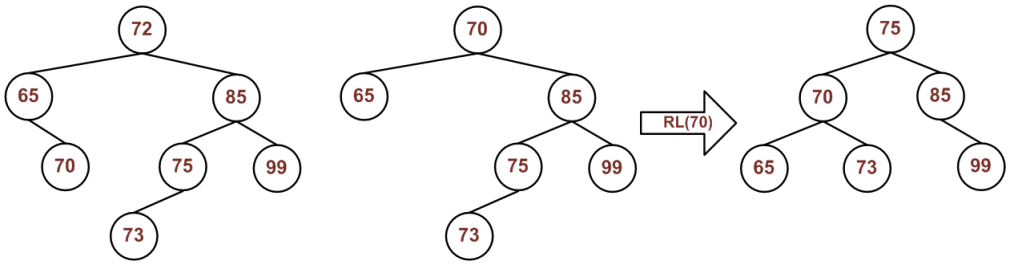


Grading scheme: 2 marks per each correct rotation. Subtract 0.5 marks per rotation if the *avlBalance* values are missing or incorrect. Subtract 3 marks if all *avlBalance* values are missing or incorrect.

b. (2 marks)

(approx. 5 minutes)

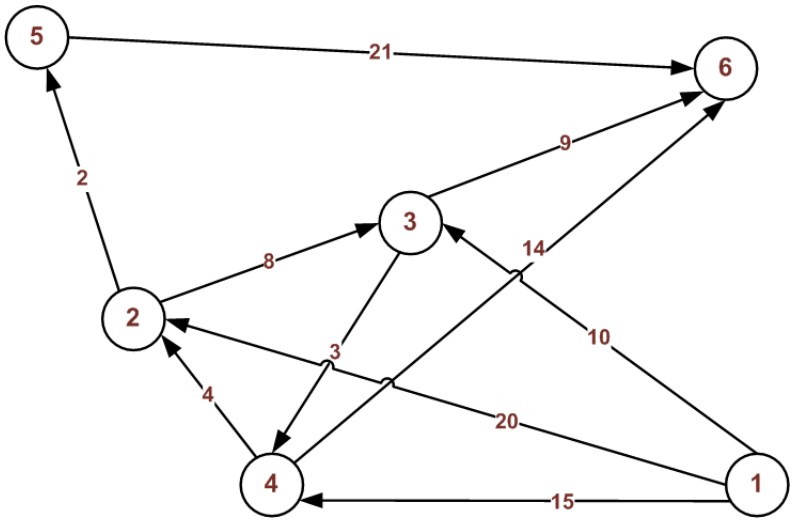
Given the following AVL tree, remove node 72 from it; use its predecessor in the removal operation. Show the rotations used in deriving your solution. Ensure that the tree remains an AVL tree after the removal is complete. **Solution:**



Grading scheme: 1 mark for each correct step.

5 Graphs (7 total marks)

(approx. 10 minutes)
Consider the following weighted directed graph:



Determine the shortest paths and their associated distance costs starting from node 1 to all the other nodes using Dijkstra's shortest-path algorithm. Show steps used in deriving your solution. When keeping track of information for each step, use the tabular format as shown in class (i.e., *iteration*, V, U, C, d_c, d_i's). Also show all the final shortest paths to each node (e.g., (1)-(i)-(j)); if there are multiple paths that produce the lowest cost access to any one node, include them all.
Solution:

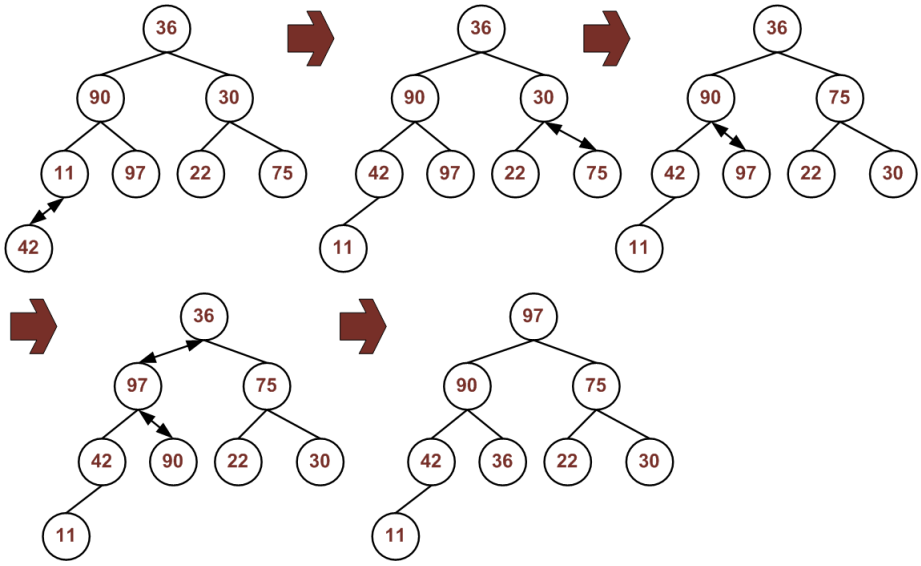
Iter	V	U	C	d_c	d_1	d_2	d_3	d_4	d_5	d_6
1	1	2,3,4,5,6	1	0	0	20 (1)-(2)	10 (1)-(3)	15 (1)-(4)	∞	∞
2	1,3	2,4,5,6	3	10	0	20 (1)-(2)	10 (1)-(3)	13 (1)-(3)-(4)	∞	19 (1)-(3)-(6)
3	1,3,4	2,5,6	4	13	0	17 (1)-(3)-(4)-(2)	10 (1)-(3)	13 (1)-(3)-(4)	∞	19 (1)-(3)-(6)
4	1,2,3,4	5,6	2	17	0	17 (1)-(3)-(4)-(2)	10 (1)-(3)	13 (1)-(3)-(4)	19 (1)-(3)-(4)-(2)-(5)	19 (1)-(3)-(6)
5	1,2,3,4,5	6	5	19	0	17 (1)-(3)-(4)-(2)	10 (1)-(3)	13 (1)-(3)-(4)	19 (1)-(3)-(4)-(2)-(5)	19 (1)-(3)-(6)
6	1,2,3,4,5,6		6	19	0	17 (1)-(3)-(4)-(2)	10 (1)-(3)	13 (1)-(3)-(4)	19 (1)-(3)-(4)-(2)-(5)	19 (1)-(3)-(6)

6 Heaps (5 total marks)

(approx. 10 minutes)

Given the following array: $[36, 90, 30, 11, 97, 22, 75, 42]$. Transform the array into a max-heap. That is, set it into the right format, and then apply the heapify function; do not insert one element at a time. Show all steps used in deriving your solution.

Solution:



Grading scheme: 1 mark for each correct swap.

7 Sorting (6 total marks)

(approx. 15 minutes)
Consider the following random and unordered sequence of integers:
16, 35, 25, 79, 12, 6, 12, 56

Apply the Quick Sort algorithm that was discussed in class to sort the letters in descending (non-ascending) order. As the pivot, always select the second element from each collection (e.g., select 35 as the first pivot), and then swap the pivot with the last element; do not just insert the pivot at the end.

For each collection, clearly mark the pivot, indicate where the swaps are performed, and show *low* and *high* pointers before each swap; you need to perform an in-place sort without using additional data structures. **Solution:**

```
       16      35      25      79      12      6       12      56
                ^ Pivot
Iteration 1:
        16      56      25      79      12      6       12      35
        ^ High                                          ^ Low   ^ Pivot
        16      56      25      79      12      6       12      35
        ^ High                  ^ Low                           ^ Pivot
        79      56      25      16      12      6       12      35
        ^ High                  ^ Low                           ^ Pivot
        79      56      25      16      12      6       12      35
                ^ Low   ^ High                                  ^ Pivot
        79      56      35      16      12      6       12      25
                        ^ Sorted

Iteration 2:
        79      56
                ^ Pivot
        79      56
        ^ Low   ^ Pivot / High
        79       56
        ^ Sorted ^ Sorted

        16      12      6       12      25
                ^ Pivot
        16      25      6       12      12
                        ^ High  ^ Low   ^ Pivot
        16      25      12       6      12
                        ^ High  ^ Low   ^ Pivot
        16      25      12       6      12
                        ^ Low    ^ High  ^ Pivot
        16      25      12       12       6
                                ^ Sorted  ^ Sorted
```

```
Iteration 3:
        16      25      12
                ^ Pivot
        16      12      25
        ^ High  ^ Low   ^ Pivot
        16      12      25
  ^ Low ^ High          ^ Pivot

        25      12      16
        ^ Sorted

Iteration 4:
                12            16
                ^ High/Low    ^ Pivot
                12      16
          ^ Low ^ High  ^ Pivot
                16        12
                ^ Sorted  ^ Sorted

Solution:
79    56    35    25    16    12    12    6
```

Grading scheme: 2 marks each for Iteration 1 and Iteration 2; 1 mark each for Iteration 3 and Iteration 4; for the total of 6 marks. If the correct method is not used (e.g., not using high/low pointers, using additional data structures), award up to 1 mark for reasonable effort.

8 Data Structure Design (10 total marks)

(approx. 25 minutes)
We are redesigning art auction house software from the midterm exam. This time, we are tasked with improving the performance of our data structure used to store various types of artwork.

a. (2 marks) We will make use of the binary search tree (BST) implementation from A3 as the basis for our redesign. To that end, in the code below provide declarations for missing attributes that are needed to make *Artwork* and *ArtManager* function.

Solution:

```
class Artwork {
    unsigned int yearMade;
    string artistName, artTitle, artType;
    // TODO: insert other required member attributes here
    Artwork* left; // 0.5 mark
    Artwork* right; // 0.5 mark


    friend class ArtManager;
public:
    Artwork() { ... }

    // other methods omitted for brevity
};


class ArtManager {
    // TODO: insert other required member attributes here
    Artwork* root_; // 0.5 mark
    unsigned int size_; // 0.5 mark



public:
    ArtManager();
    ~ArtManager();

    bool insertArtwork(Artwork* record);
    bool removeArtwork(string artistName, string title);
    // other methods omitted for brevity
};
```

b. (8 marks) Write the method $ArtManager :: removeArtwork(string\ artistName,\ string\ title)$ to match the new BST implementation. You are only required to handle cases where the node that is being removed is a leaf node or a node with one child. That is, do not implement code that handles the case where the node that is being removed has two children.

The method removes the artwork with the given artist name and artwork title. If the artwork is not found or if the target node has two children, the method returns $false$. Otherwise, if the record is found and the removal succeeds, the method returns $true$.

You may not call other class methods without implementing them first. Also, you may only use *iostream* and *string* libraries in your implementation, and no other external libraries. However, you may write helper functions of your own. Include appropriate error checking in your code, and adequately document your code. Marks will be deducted for implementations that are difficult to read, or that are not adequately documented.

Solution:

```
bool ArtManager::removeArtwork(string artistName, string title) {
    Artwork** curnode = &root_; // 2 marks for finding the right node
    while( (*curnode) != NULL ) {
        if( artistName == (*curnode)->artistName && title == (*curnode)->title )
            break;
        else if( artistName < (*curnode)->artistName )
            curnode = &(*curnode)->left;
        else
            curnode = &(*curnode)->right;
    }

    if( (*curnode) == NULL ) // 0.5 marks for returning false if not found
        return false; // the given value was not found

    // the given value was found, so remove the node
    Artwork* left = (*curnode)->left;
    Artwork* right = (*curnode)->right;

    if( left != NULL && right != NULL ) { // 1 mark
        return false; // node has two children
    } else if (left != NULL) { // 2 marks
        delete *curnode;
        *curnode = left;
    } else { // 2 marks
        delete *curnode;
        *curnode = right;
    }
    --size_;
    return true; // 0.5 marks
}
```

Grading scheme: See the grading scheme above.