University of Waterloo
Final Examination Sample Solutions
Spring 2015

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

Instructors:	Igor Ivkovic
Date of Exam	August 8th 2015
Time Period	12:30pm-3:00pm
Duration of Exam	150 minutes
Pages (including cover)	15 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: (9 marks)	Question 4: (15 marks)
Question 5: (10 marks)	Question 6: (10 marks)	Question 7: (6 marks)	Question 8: (10 marks)
Total: (80 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),
  $$S = n\left(\frac{a_1 + a_n}{2}\right) \tag{1}$$
- $$\sum_{i=k}^{n} 1 = (n - k + 1) \tag{2}$$
- $$\sum_{i=1}^{n} i = \left(\frac{n(n+1)}{2}\right) \tag{3}$$
- $$\sum_{i=1}^{n} i^2 = \left(\frac{n(n+1)(2n+1)}{6}\right) \tag{4}$$
- $$\sum_{i=0}^{n} r^i = \left(\frac{r^{n+1} - 1}{r - 1}\right) \tag{5}$$
- $$\log_b x = \frac{\log_c x}{\log_c b} \tag{6}$$
- $$\log_a a^x = x \tag{7}$$
- $$a^{\log_a x} = x \tag{8}$$

1 Algorithm Complexity (8 marks)

Consider the following function named *mergeSort* that takes as input an unordered array of integers A, and applies Merge Sort to the array to sort it in ascending order:

```
void mergeSort(int A[], int first, int last)}
    // If only one element is left, A is already sorted
    if (last - first <= 1) return;

    // Compute the middle point in the array
    int middle = (first + last) / 2;

    // Recursively divide the array into two subarrays
    mergeSort(A, first, middle);
    mergeSort(A, middle + 1, last);

    // And then merge the two subarrays
    merge(A, first, middle, last);
}
```

a. (3 marks)

Determine the recurrence relationship for the runtime efficiency $T(n)$ of *mergeSort*. The function *merge* requires n operations on each recurrence cycle, where n is the size of the problem for that cycle (e.g., for T(n), *merge* will take n operations; for T(n/2), *merge* will take $n/2$ operations; and so on).

Solution:

$T(1) = a$, $T(n) = 2T(n/2) + n + b$, where a is a constant that represents the number of primitive operations used in the base case, and b is a constant that represents the number of primitive operations needed to execute each cycle.

Grading scheme: 1 mark for the base case, 2 marks for the recursive case, for the total of 3 marks.

b. (5 marks)

Solve the recurrence relationship for this function 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. You do not have to provide a formal proof using the Big-O formal definition. Show all steps in deriving your solution.

Solution:

$$\begin{aligned}T(n) = 2T(n/2) + n + 2^0 b = 2^2 T(n/2^2) + n + 2\frac{n}{2} + 2^1 b + 2^0 b = \\ 2^3 T(n/2^3) + n + 2\frac{n}{2} + 2^2\frac{n}{2^2} + 2^2 b + 2^1 b + 2^0 b = \\ ... = 2^i T(n/2^i) + in + \frac{2^i - 1}{2-1} b\end{aligned} \tag{9}$$

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

From there,

$$\begin{aligned}T(n) = 2^i T(n/2^i) + in + \frac{2^i - 1}{2-1} b = \\ 2^{\log_2(n)} T(1) + \log_2(n) n + \frac{2^{\log_2(n)} - 1}{2-1} b = \\ an + \log_2(n) n + bn - b = O(n \log(n)).\end{aligned} \tag{10}$$

Grading scheme: 2 marks for the unfolding to n = 1 case, 1 mark for the computation of $c = \log_2(n)$, and 2 marks for the final derivation of O(n log(n)), for the total of 5 marks.

2 Algorithm Design (12 marks)

Implement a non-recursive $O(n^2)$ function named $transferStack$ that takes as input a collection of items stored in *DynamicStack p*, and transfers the elements from p into a new *DynamicStack q* while preserving the order from p in q. Once q includes all the data items from p and in the same order as in p, the function returns q and terminates.

Your function needs to use only one other *DynamicStack* to implement the algorithm; that is, the new *DynamicStack q*. You may also use a number of primitive temporary variables, but no other arrays, lists, stacks, or other data structures. You may use helper functions if needed, but those need to be non-recursive too.

In your code, you may only call the stack methods, such as *bool empty*(), *int size*(), *void push* (*StackItem value*), *StackItem pop*(), and *StackItem peek*(). However, your code should not make any assumptions about how any of these methods are implemented internally. 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., "*p.pop*();").

Clearly explain your design, and adequately document the steps in your code. Also, include error checking where appropriate.

```
DynamicStack transferStack(DynamicStack p) {
    DynamicStack q;

    // Implement your code here
```

Solution:

```
    // iterate through the stack p.size() times
    int p_size1 = p.size();
    for (int count = 0; count < p_size1; ++count) {              [2 marks]
        // pop the top element from p, and store it as temp
        StackItem temp = p.pop();                                [1 mark]
        // remove p.size() - count elements from p, and store them onto q
        int p_size2 = p.size();
        for (int i = 0; i < (p_size2 - count); ++i)              [3 marks]
            q.push(p.pop());
        // store temp onto p
        p.push(temp);                                            [1 mark]
        // move back all the elements from q to p
        while(!q.empty())                                        [2 marks]
            p.push(q.pop());
    }
    // move all the elements from p to q
    while(!p.empty())                                            [2 marks]
        q.push(p.pop());

    return q;                                                    [1 mark]
}
```

3 Tree Traversals (9 total marks)

Consider the following array representation of a complete binary tree:

	A	L	G	O	R	I	T	H	M	C	O	M	P	L	E	X	I	T	Y

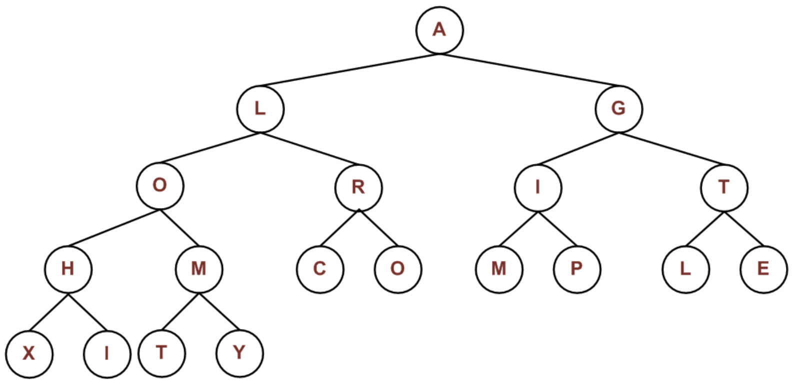
a. (0.5 marks) What is the parent of Y?
Answer: M (index 9)

b. (0.5 marks) What is the right child of R?
Answer: O (index 11)

c. (0.5 marks) What is the left child of G?
Answer: I (index 6)

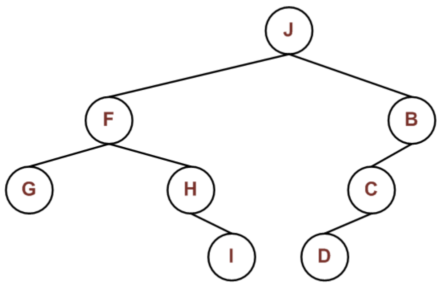
d. (1 marks) List all the leaf nodes.
Answer: COMPLEXITY (19 $<$ 2 x index)

e. (2.5 marks)
Write the order in which the nodes are visited for post-order traversal.
Solution:



Order: X I H T Y M O C O R L M P I L E T G A

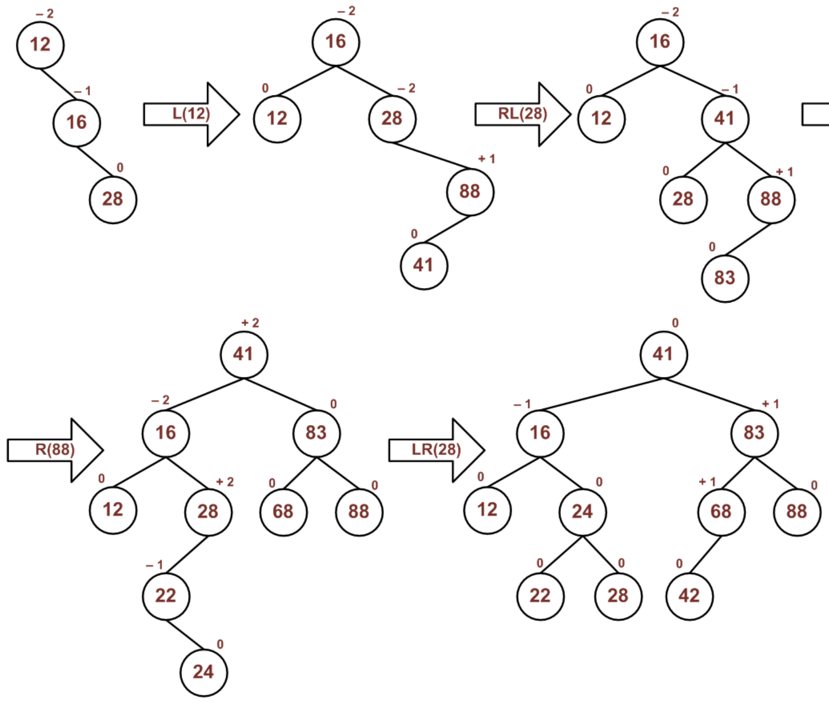
f. (4 marks)
A different binary tree was processed using post-order and in-order traversals. For post-order, the output derived is [G, I, H, F, D, C, B, J]. For in-order, the output derived is [G, F, H, I, J, D, C, B]. Draw a binary tree that complies with the traversals above. Show steps in deriving the tree. **Solution:**



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

4 AVL Trees (15 total marks)

a. (10 marks) Given an empty AVL tree, insert the following nodes into the tree:
12 16 28 88 41 83 68 22 24 42
Show the rotations used in deriving your solution, and write the *avlBalance* values for each node before each rotation. 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: 3 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. (5 marks) Write the function *performRLRotation* that performs the right-left AVL rotation at *pNode*. The function first applies the right rotation at *pNode*'s right child, and then it applies the left rotation at *pNode*. *pNodeAddress* is the address of a pointer to the *pNode*, such the parent node's left or right child, or the root node pointer. Update *pNodeAddress* value when appropriate.

To get you started, we have provided pointers *qNode* as $pNode \rightarrow right$ and *rNode* as $qNode \rightarrow left$. Use these pointers in your implementation. Adequately document your code.

```
void AVLTree::performRLRotation(BSTNode* pNode,
                                BSTNode** pNodeAddress) {
    BSTNode* qNode = pNode->right;
    BSTNode* rNode = qNode->left;

    // Implement your code here
```

Solution:

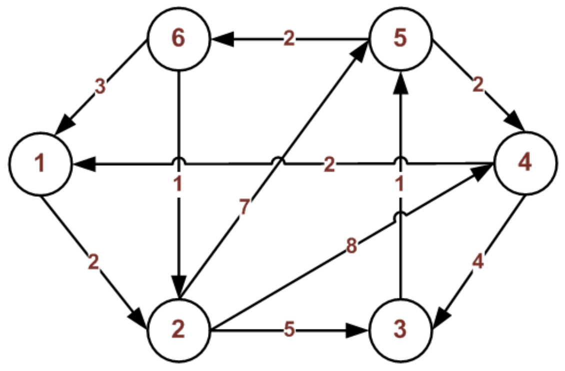
```
// adjust pNode and qNode [2 marks]
pNode->right = rNode->left;
qNode->left = rNode->right;

// adjust rNode [2 marks]
rNode->left = pNode;
rNode->right = qNode;

// adjust the parent node [1 mark]
*pNodeAddress = rNode;
```

5 Graphs (10 total marks)

Consider the following weighted directed graph:



Determine the shortest paths and their associated distance costs starting from node 3 (not 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 the shortest path to each node (e.g., (3)-(i)-(j)) after each iteration.

Solution:

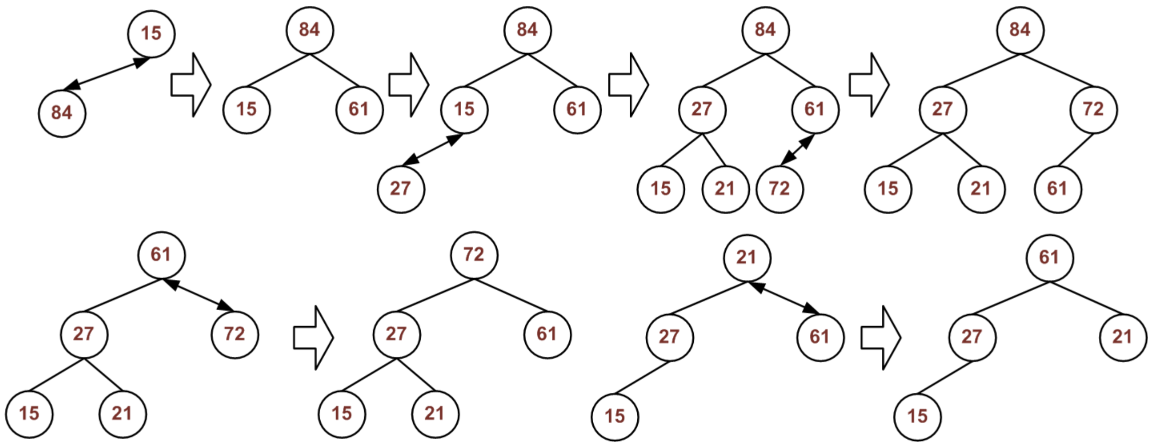
Iter	V	U	C	d_c	d_1	d_2	d_3	d_4	d_5	d_6
1	3	1,2,4,5,6	3	0	∞	∞	0	∞	1 (3)-(5)	∞
2	3,5	1,2,4,6	5	1	∞	∞	0	3 (3)-(5)-(4)	1 (3)-(5)	3 (3)-(5)-(6)
3	3,4,5	1,2,6	4	3	5 (3)-(5)-(4)-(1)	∞	0	3 (3)-(5)-(4)	1 (3)-(5)	3 (3)-(5)-(6)
4	3,4,5,6	1,2	6	3	5 (3)-(5)-(4)-(1)	4 (3)-(5)-(6)-(2)	0	3 (3)-(5)-(4)	1 (3)-(5)	3 (3)-(5)-(6)
5	2,3,4,5,6	1	2	4	5 (3)-(5)-(4)-(1)	4 (3)-(5)-(6)-(2)	0	3 (3)-(5)-(4)	1 (3)-(5)	3 (3)-(5)-(6)
6	1,2,3,4,5,6		1	5	5 (3)-(5)-(4)-(1)	4 (3)-(5)-(6)-(2)	0	3 (3)-(5)-(4)	1 (3)-(5)	3 (3)-(5)-(6)

6 Heaps (10 total marks)

a. (4 marks)

Insert the following nodes into an empty max-heap: **15 84 61 27 21 72**
After the insertions, run the *remove* function two times (i.e., remove two elements from the heap). Show all steps used in deriving your solution.

Solution:



b. (6 marks)

A max-heap ADT is also a complete binary tree that consists of binary tree nodes. Consider a linked data representation of $BinaryTreeNodes$, where each $BinaryTreeNode$ includes pointers to $leftChild$ and $rightChild$.

Design the function *bool* $isTreeComplete(BinaryTreeNode * \ T)$ that takes as input the root node of a tree. The function outputs *true* if the given tree is complete, and $false$ otherwise.

Clearly explain your design. 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 isTreeComplete(BinaryTreeNode* T) {
    // specify algorithm steps in structured English
```

Solution:
(Step1) Perform level-based traversal of the tree (e.g., using a queue), and store the tree into an array of pointers.
(Step1.1) For each level up to the tree height, store each node pointer, including NULLs, as one of the elements of the array.
(Step2) Iterate through the array, and check for NULL nodes.
(Step2.1) If there exists a NULL node followed by a non-NULL node, return false and terminate.
(Step2.2) If there exists no NULL node followed by a non-NULL node, and the end of the array is reached, return true and terminate.

7 Sorting (6 total marks)

Consider the following unordered sequence of letters:
N, C, B, Z, E, Q, D, F

Apply the Quick Sort algorithm that was discussed in class to sort the letters in descending alphabetical order (e.g., **D** should be listed before **C**). As the pivot, always select the last element from each collection (e.g., select **F** as the first pivot). You must move the pivot value to the beginning of each collection, apply the pivoting, and then place the pivot into the appropriate location.

For each collection, clearly mark the pivot, indicate where the swaps are performed, and show *low* and *high* pointers before each swap. **Solution:**

Solution1: Pivot was Moved (Swapped with First Element)

```
Iteration 1:
N         C   B   Z   E   Q   D   F


F         C   B   Z   E   Q   D   N
^ Pivot   ^ High                  ^ Low


F         N   B   Z   E   Q   D   C
^ Pivot   ^ High                  ^ Low


F         N   B   Z   E   Q   D   C
^ Pivot       ^ High      ^ Low


F         N   Q   Z   E   B   D   C
^ Pivot       ^ High      ^ Low


F         N   Q   Z   E   B   D   C
^ Pivot           ^ L ^ H


Z         N   Q   F           E   B   D   C
                  ^ Sorted
Iteration 2:
      Q         N   Z         C       B   D   E
      ^ Pivot   ^ H ^ L       ^ Pivot ^ H     ^ L
      Q         Z   N         C       E   D   B
      ^ Pivot   ^ H ^ L       ^ Pivot ^ H     ^ L
      Q         Z   N         C       E   D   B
      ^ Pivot   ^ L ^ H       ^ Pivot     ^ L ^ H
Z         Q        N          D       E   C        B
^ Sorted  ^ Sorted ^ Sorted               ^ Sorted ^ Sorted


Iteration 3:
                           E          D
                           ^ Pivot    ^ H/L
                           E          D
                           ^ Pivot/L  ^ H
                           E          D
                           ^ Sorted   ^ Sorted
```

```
Solution:
      Z   Q   N   F   E   D   C   B
```

Solution2: Pivot is Fixed (out of 4 marks)

```
Iteration 1:
      N   C   B   Z   E   Q   D       F
      ^ High                ^ Low   ^ Pivot
      N   C   B   Z   E   Q   D       F
          ^ High          ^ Low       ^ Pivot
      N   Q   B   Z   E   C   D       F
          ^ High          ^ Low       ^ Pivot
      N   Q   B   Z   E   C   D       F
              ^ H ^ L                 ^ Pivot
      N   Q   Z   B   E   C   D       F
              ^ H ^ L                 ^ Pivot
      N   Q   Z   B   E   C   D       F
              ^ L ^ H                 ^ Pivot
      N   Q   Z   F   E   C   D   B
                  ^ Sorted
Iteration 2:
      N   Q   Z         E   C   D   B
      ^ H ^ L ^ Pivot   ^ H     ^ L ^ Pivot
      N   Q   Z         E   C   D   B
  ^ L ^ H     ^ Pivot           ^ L ^ Pivot/H
      Z   Q   N         E   C   D   B
      ^ Sorted                      ^ Sorted

Iteration 3:
      Q      N          E   C   D
      ^ H/L  ^ Pivot    ^ H ^ L ^ Pivot
      Q      N          E   C   D
      ^ Low  ^ Pivot/H  ^ L ^ H ^ Pivot
      Q        N          E        D        C
      ^ Sorted ^ Sorted   ^ Sorted ^ Sorted ^ Sorted

Solution:
      Z   Q   N   F   E   D   C   B
```

Grading scheme: Subtract 0.5 marks for each incorrect step or part of iteration. Moving in front of the first element instead of swapping is also acceptable. Subtract 2 marks if the pivot was not moved as specified.

8 Data Structure Design (10 total marks)

We are designing a mobile application (app) for a service similar to Netflix that offers a subscription-based access to movies and television series. Our app allows users to create a list of their favourite viewing items that is offered by the service. For each favourite item, the app stores the item name as a string, item rating as an integer that represents the number of stars (from one star to five stars in one-star increments), item genre as a string (e.g., “Comedy” or “TV Show”), and user-defined notes also as a string.

To organize the items into a collection, we are going to make use of binary search trees (BST), and implement the related functionality based on BST functions. To that end, we have provided the initial data structure definition for *FavItem* and *MyFavourites* as follows:

```
class FavItem {
    string title, genre, userNote;
    int rating;
    FavItem* left, right;

    friend class MyFavourites;

public:
    FavItem();
    FavItem(string newTitle, string newGenre, string newUserNote, int newRating);
    ~FavItem();

    bool operator< (const FavItem& rhs) const;
    bool operator== (const FavItem& rhs) const;

    // getters and setters omitted for brevity
};

class MyFavourites {
    FavItem* root;
    int size;
public:
    int getSize() const;

    bool insertItem(FavItem* item);
    bool existsItem(FavItem* item);
    bool removeItem(FavItem* item);
};
```

a. (3 marks)

Implement the *operator* < (is-less-than) function for *FavItem* as declared above. The function compares the current *FavItem* object with the right-hand side *FavItem* object, and it allows the usage as $A < B$ where A and B are of *FavItem* type.

Use *title* as the key for the comparison. Return *true* if the key of the current *FavItem* is less than the key of the right-hand side *FavItem*; otherwise, return *false*. You can assume that the *operator* < is already available for strings.

Solution:

```
bool FavItem::operator< (const FavItem& rhs) const { //[1 mark]
    return title < rhs.title; //[2 marks]
}
```

b. (7 marks)

Implement *bool MyFavourites :: insertItem(FavItem * item)* using the BST functionality. That is, insert the new favourite item into the appropriate location inside the BST, and ensure that the BST property holds after each insertion. To compare two *FavItem* objects, use the *operator* < as declared above. You can assume that this operator has been implemented correctly, irrespective of your answer to part (a). For duplicate keys, insert the value into the right subtree.

You may only use *iostream* and *string* libraries in your implementation, and no other external libraries. Include appropriate error checking in your code, and adequately document your code.

```
bool MyFavourites::insertItem(FavItem* item) {
    // Implement your function here
```

Solution:

```
    if (!item) return false; //[1 mark]

    FavItem** curitem = &root; //[1 mark]

    while(*curitem) {  //[1 mark]
        if(*item < **curitem)
            curitem = &(*curitem)->left; //[1 mark]
        else
            curitem = &(*curitem)->right; //[1 mark]
    }
    *curitem = item; //[1 mark]
    size++; //[1 mark]

    return true;
}
```