ECE 250 *Algorithms and Data Structures*
MIDTERM EXAMINATION

Douglas Wilhelm Harder dwharder@uwaterloo.ca EIT 4018 x37023

2013-10-23T09:30:00P1H20M

Rooms: RCH-103 and RCH-302

Instructions:

_____ Read and initial each of these instructions for one bonus mark.
_____ There are 43 marks.
_____ No aides.
_____ Turn off all electronic media and store them under your desk.
_____ If there is insufficient room, use the back of the previous page.
_____ You may ask only one question during the examination:
"May I go to the washroom?"
_____ Asking **any** other question **will** result in a deduction of 5 marks from the exam grade.
_____ If you think a question is ambiguous, write down your assumptions and continue.
_____ **Do not leave during f hour or after there are only 15 minutes left.**
_____ Do not stand up until all exams have been picked up.
_____ If a question asks for an answer, you do not have to show your work to get full marks; however, if your answer is wrong and no rough work is presented to show your steps, no part marks will be awarded.

Attention:

The questions are in the order of the course material, not in order of difficulty.

THIS BLOCK MUST BE COMPLETED USING ALL CAPITAL LETTERS IN PEN

Surname/Last Name								
Legal Given/First Name(s)								
UW Student ID Number	2	0						
UW User ID								

I have read the above instructions:

Signature: ____

Asking any question other than that question noted above. -5

A.1 [2] List those relationships where there is a form of imposed order: that is, relationships are defined in terms of succession; for example, where for two elements, it may be possible to write $x \prec y$ or $x < y$.

A.2 [2] Using l'Hopital's rule and algebra, show that $n \ln(n) = o(n^{1.1})$.

A.3 [1] We can say that $f(n) < g(n)$ if $f(n) = \mathbf{o}(g(n))$. Does this define a linear order or a weak order? Justify your answer.

A.4 [2] What are the run times of the following code segments:

```
for ( int i = 0; i < n*m; ++i ) {
      for ( int j = 0; j < i; ++j ) {
            sum += i + j;
      }
}

for ( int i = 0; i < n; ++i ) {
      for ( int j = 0; j < n*m; ++j ) {
            sum += i + j;
      }
}
```

B.1 [4] Implement a `last( Type const & )` function for a singly linked list which returns a pointer to the last node in a linked list that contains an object equal to the argument (there may be multiple such nodes). Fill in the appropriate return type and implement the function. You may require the member functions `head()` and `tail()`, and `next()` and `retrieve()`. If there is no node containing the argument, return `nullptr`.

```
template <typename Type>
                                        Single_list<Type>::last( Type const & ) const {
```

B.2 [2] Why is it more complex to double the size of an array containing a queue as compared to doubling the size of an array containing a stack. You can use images to support your arguments.

C.1 [2] Define the depth of a node and the height of a tree in terms of path lengths.

C.2 [2] Write a member function `depth_print` that performs a pre-order depth-first traversal on a tree by printing out each node as it is visited together with the depth of the node followed by an underscore. For example, the output of the function run on the tree in Figure C.2 would be

```
A0_B1_E2_C1_D1_F2_G2_
```

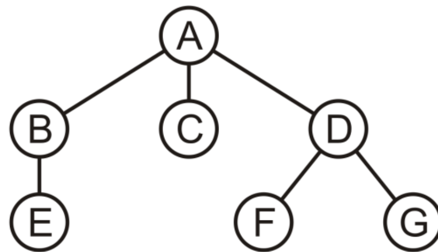

Figure C.2. A tree.

For your information, recall that a simple tree is a recursive structure: all children of a simple tree are themselves simple trees. You will have to pass a specific argument when you call the function, and this argument will have to be modified as the function recursively calls its children.

```
	template <typename Type>
	class Simple_tree {
		private:
			Type element;
			Simple_tree *parent_node;
			Single_list<Simple_tree *> children;
		public:
			void depth_print(          ) const;
	}

template <typename Type>
void Simple_tree<Type>::depth_print(                    ) const {
```

C.3 [**2**] A developer has already implemented an efficient implementation of the member function

```
template <typename Type>
bool Simple_tree<Type>::is_ancestor_of( Simple_tree<Type> *treeptr ) const;
```

which returns `true` if this node in the tree is an ancestor of the node pointed to by the argument `treeptr` by following the list of `parent_node` pointers of the argument (see Question C.2). You have to extend the interface by implementing a member function that returns `true` if this node in the tree is a descendant of the node pointed to by the argument `treeptr`. Implement this function efficiently, too. There are no restrictions on what you may do or what member functions you may call to implement this member function.

```
template <typename Type>
bool Simple_tree<Type>::is_descendant_of( Simple_tree<Type> *treeptr ) const {
```

D.1 [1] Describe the difference between a binary tree and a general tree where each node is restricted to having at most two children.

D.2 [2] A full binary tree is one where each node is either a leaf node or a full node. What is the minimum number of nodes in a full binary tree of height h? You must give a formula, but you need not give a proof.

D.3 [3] Write a function that returns `true` if a tree is a full binary tree and `false` otherwise. An empty node is considered to be a full binary tree. You may use the `bool empty()` and `bool is_leaf()` member functions. You may be interested in using the member functions `retrieve()`, `left()`, and `right()`.

```
template <typename Type>
bool Binary_node<Type>::is_full() const {
```

E.1 [3] Give a proof by induction that a perfect binary tree of height h has $2^h - 1$ internal nodes. You must justify any statement you make.

E.2 [4] Explain why and how we must use the sum

$$\sum_{k=0}^{h} k2^k = h2^{h+1} - 2^{h+1} + 2$$

to find the average depth of a node in a perfect binary tree of height h, which contains $2^{h+1} - 1$ nodes, is asymptotically (in the limit as h becomes large) $h - 1$.

E.3 [3] Demonstrate how the complete binary tree in Figure E.3 can be stored in the provided array and describe how we can efficiently find the children and the parent of the node stored at index `k` within the array.

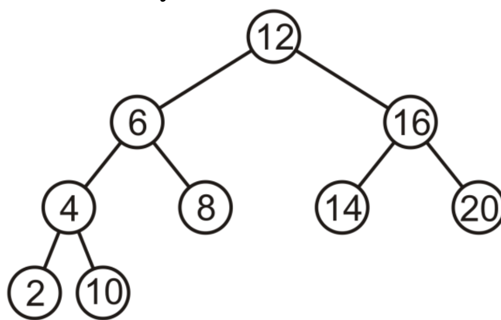

Figure E.3. A complete binary tree.

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

F.1 [2] In the order given, insert the numbers 72, 53, 85, 58, 42, 23, 33, 47, 34 into an initially empty binary search tree.

F.2 [6] Write a member function `next_largest( Type const & )` that returns a pointer to the node that contains the smallest element in the binary search tree that is greater than the argument. If the argument is greater than or equal to the largest entry, return `nullptr`. Figure F.2 may be used to determine how you should implement the algorithm.

```
template <typename Type>
Binary_search_node<Type> *Binary_search_tree<Type>::next_largest( Type const &obj ) {
	root_node->next_largest( obj );
}

template <typename Type>
Binary_search_node<Type> *Binary_search_node<Type>::next_largest( Type const &obj ) {
	if ( empty() ) {
		return nullptr;
	} else {



	}
}
```

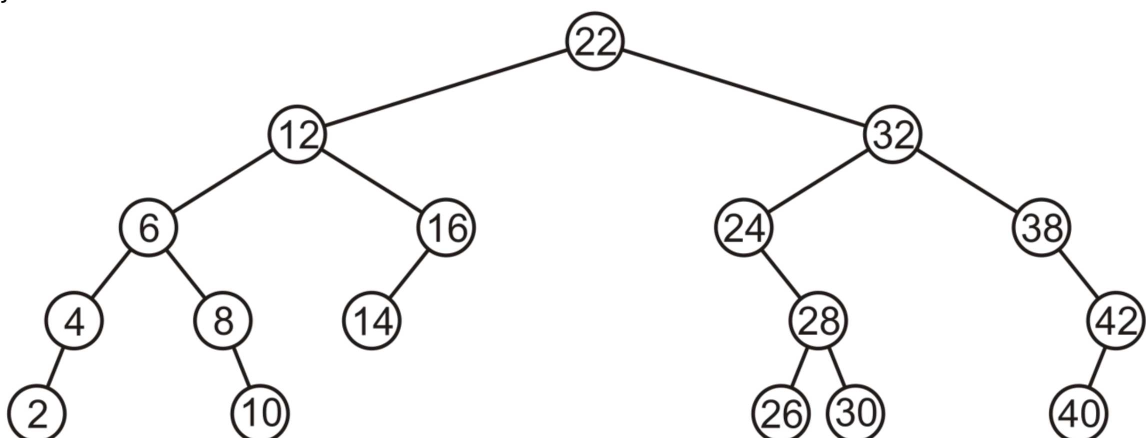

Figure F.2. A binary search tree.