University of Waterloo
Department of Electrical and Computer Engineering
ECE 250 Data Structures and Algorithms

Final Examination
2006-04-17

Instructor: Douglas Wilhelm Harder
Time: 2.5 hours
Aides: none
14 pages

Surname	Given name(s)
Student ID Number	Signature

You may only ask one question: "May I go to the washroom?"
If you are unsure of a question, write down your assumptions and continue.
If you have insufficient space to answer a question, use the back of the previous page.
The examination is out of 75 marks.

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
12.	13.	14.	15.	16.	17.	18.	19.	20.	21.	22.

1. **[4]** Assume that the function **`bool f()`** is a Boolean-valued function which randomly returns a value of **`true`** or **`false`** with a probability of 0.5. What are the best-case (e.g., **`f()`** always returns ...) and worst-case asymptotic run times of this code fragment? What is the average case asymptotic run time? Give a short justification for the average-case scenario. Otherwise, you need only to give the asymptotic run times (big-**O**) with respect to the variable **`n`**.

```
int m = 0;

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

2. **[2]** For each of the following two scenarios, suggest the appropriate data structure. You do not need to justify your choice.

a. You'd like to store information about all the circuit elements produced by one particular company. Each circuit element has its own unique six-digit identifier.

b. You require a priority queue, however, for each new element being placed into the queue, you are aware that it must have either:

i. higher priority than all elements currently in the queue, or
ii. lower priority than all elements currently in the queue.

Which data structure could be used to have optimal performance?

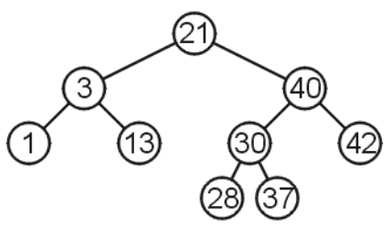
3. **[4]** The **`SingleList`** class which you implemented in Project 1 has three class members:

```
SingleNode * head;
SingleNode * tail;
int count;
```

The second and third class members are not necessary; we could implement a single list using only the **`SingleNode * head`** class member. Unfortunately, the run time of some of the member functions will change from **O**(1) to **O**(*n*). In the following table, place a check mark in the appropriate column indicating that the run time will change from O(1) to O(*n*) as a result of having removed that class member.

	Remove **`tail`**	Remove **`count`**
`int size() const;`		
`bool empty() const;`		
`Object front() const;`		
`Object back() const;`		
`SingleNode<Object> * head() const;`		
`SingleNode<Object> * tail() const;`		
`bool member( const Object & obj ) const;`		
`void push_front( const Object & obj );`		
`void push_back( const Object & obj );`		
`Object pop_front();`		
`bool remove( const Object & obj );`		

4. **[2]** Perform pre-order and post-order depth-first traversals of this tree:

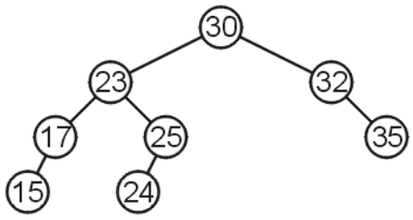


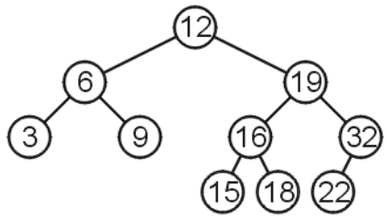
Place your answers in these tables:

Pre-order depth-first traversal

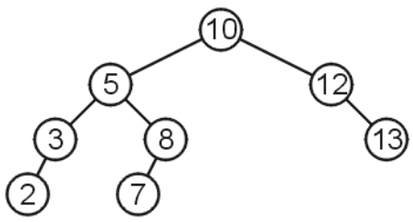
Post-order depth-first traversal

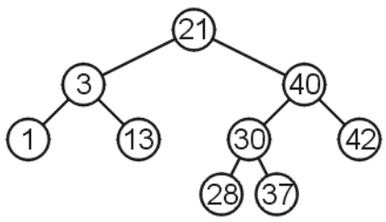
5. **[4]** For each of the following AVL trees, insert 13. Perform whatever rotations are necessary to maintain AVL balance.





For each of the following AVL trees, delete the node containing 13. Perform whatever rotations are necessary to maintain AVL balance. If no rotations are necessary, you can simply cross the node out.





6. **[4]** Implement the member function **`rotate_to_right`** which rotates around **`this`** which is a pointer to the unbalanced node. Update the heights of any nodes which may have had their heights changed as a result of the rotations by calling the **`void update_height()`** member function on those nodes (you do not have to implement **`void update_height()`**). You may optionally, though you are not required to, use the argument **`parent`** (this was used in some student's implementation, and thus, I will provide it if you wish to use it).

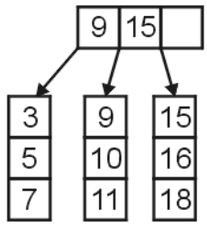
The structure of the **`AVLNode`** class is

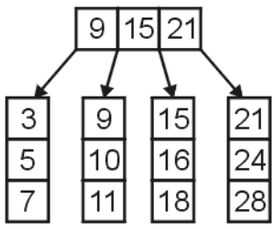
```
template <class Comparable>
class AVLNode {
      private:
            Comparable element;
            int height;
            AVLNode<Comparable> * left_tree;
            AVLNode<Comparable> * right_tree;
      public:
            // ...
};
```

Enter your answer here:

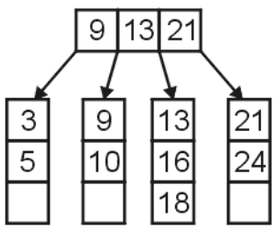
```
template<class Comparable>
void rotate_to_right( AVLNode<Comparable> * & parent ) {









}
```

7. **[4]** Insert 13 into the following two B-trees.





Remove 13 from this B-tree:



8. **[2]** In double hashing, it is necessary that the skip (the secondary hash function) must be relatively prime to size of the hash table. How can we ensure this if the hash table has 2^k bins where k is an integer and $k > 0$. (Short answer, no proof required.)

9. **[3]** Insert the numbers 3, 13, 23, and 33 into the following hash table which currently has only a single entry. Use quadratic hashing and use the least-significant digit to represent the initial bin.

0	1	2	3	4	5	6	7	8	9	10
							17			

10. **[2]** What is the printed output of when the following code is run?

```
int n = 1;

n = (n << 5);

cout << n << endl;

if ( !( n & 1 ) ) {
    n = n | 1;
}

cout << n << endl;
```

11. **[3]** Using the implementation of a min-heap as discussed in class, insert the following numbers into an initially empty min-heap:

7, 3, 2, 5, 8, 4

Indicate the state of min-heap after each insertion by filling in the appropriate row in this table.

	0	1	2	3	4	5	6
Insert 7							
Insert 3							
Insert 2							
Insert 5							
Insert 8							
Insert 4							

12. **[4]** Suppose that an implementation of a min-heap (as described in class) which stores integers has the following class members:

```
int * array;
int array_size;
int count;
```

The variables are all self-descriptive and you may assume that in the constructor, the memory for the array is appropriately allocated.

Implement the dequeue function which returns the minimum element and readjusts the elements in the array to maintain the complete-tree shape of the array.

```
int dequeue() {
     if ( count == 0 ) {
          throw underflow();
     }
```

```
}
```

13. **[3]** Perform one pass of the bubble sort algorithm on the following list of integers (visiting each entry once). Place your answer in the second row.

3	0	1	4	2	7	8	6	9	5

14. **[4]** The following code correctly compares the entries of two sorted arrays, **`array1`** and **`array2`**, of sizes **`n1`** and **`n2`**, respectively, and merges them together into a single sorted array **`merged_array`** which is of size **`n1 + n2`**

```
int i = 0, j = 0, k = 0;

for ( /* empty */; i < n1 && j < n2; ++k ) {
     if ( array1[i] <= array2[j] ) {
          merged_array[k] = array[i];
          ++i;
     } else {
          merged_array[k] = array[j];
          ++j;
     }
}
```

Note, however, that the merging process is not complete. Write the code required to complete the merging of the two arrays.

15. **[3]** Apply merge sort on the following list of unsorted data by showing each step (to the extent shown in class) of the process. If a list has an odd number of entries, you should divide the list so that the larger sub-list is on the left. If a sub-list has three or fewer entries, you can simply sort them in the next step.

5	3	0	2	4	6	1

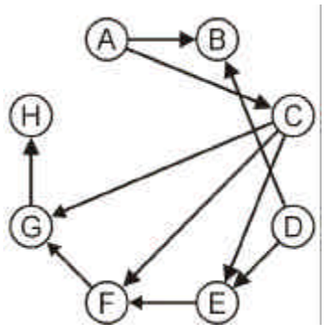
16. **[3]** Choosing the median-of-three to select the pivot (the three being the first, middle, and last entries) for quick sort has the following benefit:

> For a sorted array, the pivot will divide the array
> into two equal or almost-equal sub-arrays.

However, this also allows us to easily construct an sequence of integers for which quick sort must run in $\Theta(n^2)$ time. We can avoid this by choosing the median of three randomly chosen entries. This, however, has the drawback that a sorted list will not be sorted in an optimal amount of time.

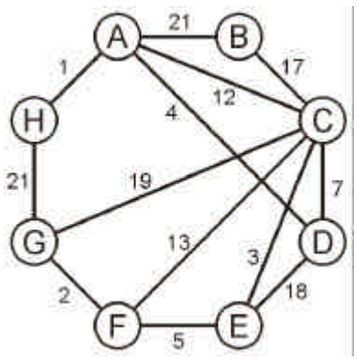
Suggest a hybrid of these two which at least partially solves both problems.

17. **[2]** Perform a topological sort of the following directed acyclic graph. There is more than one solution to this problem.

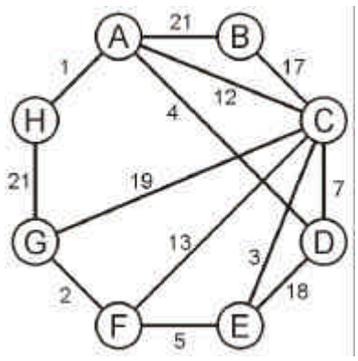


18. [6] Apply either of Prim's or Kruskul's algorithm (your choice) to find a minimum spanning tree of the following graph. Indicate which algorithm you are using. If you are using Prim's algorithm, start with the vertex A. Show each step with the appropriately numbered graphic. Draw the spanning tree at the bottom.

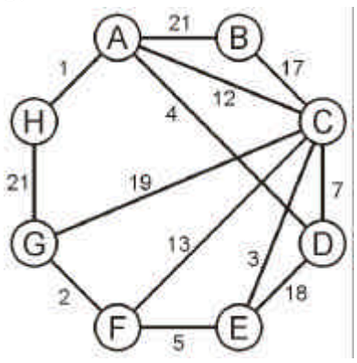
1.



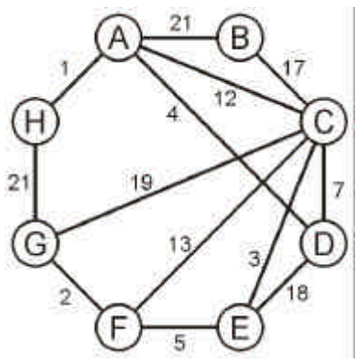
2.



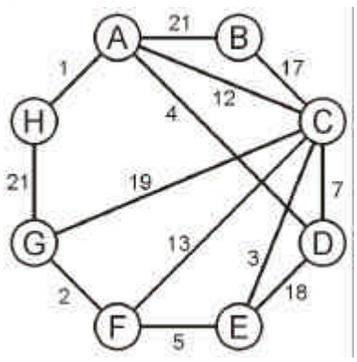
3.



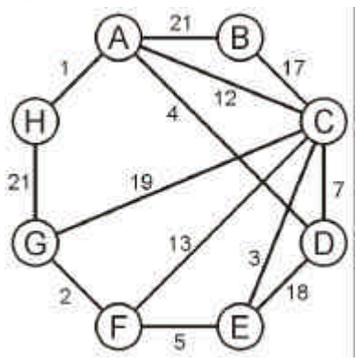
4.



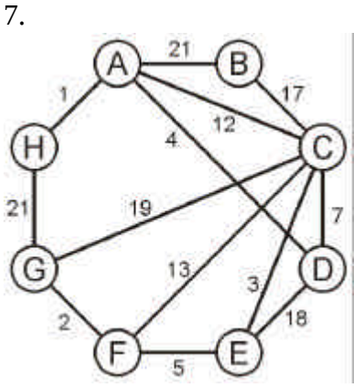
5.



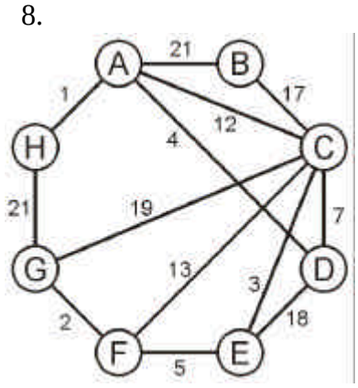
6.



7.



8.



19. **[4]** Explain why Prim's algorithm may be classified as *greedy*. You may use a diagram and a sample graph to justify your answer.

20. **[4]** For a divide-and-conquer algorithm which has a runtime $T(n) = aT(n/b) + \mathbf{O}(n^k)$ for $n > 1$, assume that $n = b^m$ and that $a > b^k$. Show how we can simplify

$$T(n) = a^m \sum_{\ell=0}^{m} \left(\frac{b^k}{a} \right)^{\ell}$$

to see that $T(n) = \mathbf{O}(n^{\log_b a})$. If you are not aware of the exact value of a particular sum, you may simply state the properties which you believe it has.

21. **[2]** Suppose we are adding two matrices which are stored using the sparse Harwell-Boeing representation. If you were to implement an algorithm for adding two such matrices, which algorithm which we have seen in class would this be closest to? (Short answer.)

22. **[6]** Using only **O**(1) additional memory, modify the **`SingleList`** class to allow the implementation of a **`Object pop_back()`** function which has the following run times:

a. If between two successive calls to **`Object pop_back()`** there is at least one call to **`void push_back( Object obj )`**, then the second call to **`Object pop_back()`** must run in **O**(1) time, otherwise
b. If between two successive calls to **`Object pop_back()`** there are no calls to **`void push_back( Object )`** then the run time of the second call to **`Object pop_back()`** need only be only **O**(*n*).

You must list the additional class members you require and you must implement the two member functions **`void push_back( Object obj )`** and **`Object pop_back()`**. You may use whatever other member functions of either class you wish to simplify your code. The interfaces to the two classes are given below. You may not modify the **`SingleNode`** class.

```
template <class Object>
class SingleNode {
   private:
      Object       element;
      SingleNode * next_node;

   public:
      SingleNode( const Object & e = Object(), SingleNode * n = 0 );
      Object retrieve() const;
      SingleNode *next() const;

      friend class SingleList<Object>;
};

template <class Object>
class SingleList {
        private:
                SingleNode<Object> * head;
                SingleNode<Object> * tail;
                int count;
                // add additional member(s) here **********************



        public:
                SingleList();
                ~SingleList();

                SingleList( const SingleList & );

                // Accessors

                int size() const;
                bool empty() const;

                Object front() const;
                Object back() const;

                SingleNode<Object> * head() const;
                SingleNode<Object> * tail() const;

                bool member( const Object & ) const;

                // Mutators

                void push_front( const Object & );
                void push_back( const Object & );

                Object pop_front();
                Object pop_back();          // * new *

                bool remove( const Object & );
};
```

```
template <class Object>
void push_back( Object obj ) {









}

template <class Object>
Object pop_back() {
        if ( empty() ) {
                throw underflow();
        }









}
```