UNIVERSITY OF TORONTO
FACULTY OF APPLIED SCIENCE AND ENGINEERING

FINAL EXAMINATION, APRIL 2010

CSC 190 H1S — ALGORITHMS, DATA STRUCTURES, AND LANGUAGES

Exam Type: C — NO calculator allowed
Examiner(s): François Pitt and Paul Gries

Student Number: |_|_|_|_|_|_|_|_|_|_|

Family Name(s): _____

Given Name(s): _____

Lecture Section: ☐ LEC 01 (with F. Pitt) ☐ LEC 02 (with P. Gries)

---

*Do **not** turn this page until you have received the signal to start.*
In the meantime, please read the instructions below *carefully.*

This final examination paper consists of 8 questions on 20 pages (including this one), printed on both sides of the paper. *When you receive the signal to start, please make sure that your copy of the final examination is complete and fill in the identification section above.*

Answer each question directly on the examination paper, in the space provided. If you need more space for one of your solutions, use one of the extra "blank" pages at the end of the examination and *indicate clearly the part of your work that should be marked.*

Write up your solutions carefully! In particular, use notation and terminology correctly and explain what you are trying to do - part marks *will* be given for showing that you know the general structure of an answer, even if your solution is incomplete.

When writing code, comments are not required except where indicated, although they may help us mark your answers. They may also get you part marks if you can't figure out how to write the code.

If you are unable to answer a question (or part), you will get 20% of the marks for that question (or part) if you write "I don't know" and nothing else—you will *not* get those marks if your answer is completely blank, or if it contains contradictory statements (such as "I don't know" followed or preceded by parts of a solution that have not been crossed off).

MARKING GUIDE

# 1:	_____/ 5
# 2:	_____/15
# 3:	_____/10
# 4:	_____/11
# 5:	_____/ 7
# 6:	_____/15
# 7:	_____/12
# 8:	_____/10
TOTAL:	_____/85

Good Luck!

OVER...

Question 1. [5 MARKS]

Give a *tight* big-O bound for the worst-case running time of each operation. Justify each answer.

Part (a) [1 MARK] Finding the second-smallest value in a *min-heap* with n elements. _____

Part (b) [1 MARK] Finding the second-smallest value in a *max-heap* with n elements. _____

Part (c) [1 MARK] Finding the second-smallest value in a *hash table* with n elements. _____

Part (d) [1 MARK] Using *QuickSort* (with the first element as pivot) to sort the values in a *min-heap* with n elements. _____

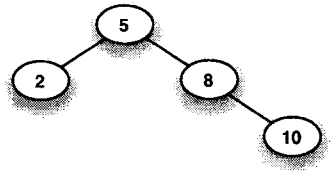
Part (e) [1 MARK] Using *MergeSort* to sort the values in a *min-heap* with n elements. _____

Question 2. [15 MARKS]

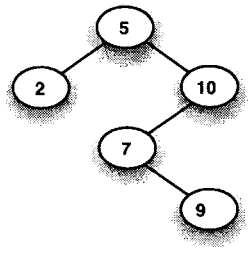
Part (a) [2 MARKS] This subquestion is about **BSTs**.

When you remove a value that has two children, replace it with the smallest value in the right subtree.

i. In the space to the right, draw this tree after 8 has been removed.



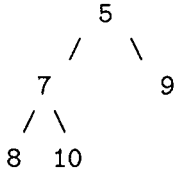
ii. In the space to the right, draw this tree after 5 has been removed.



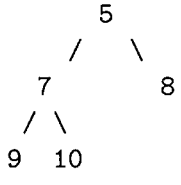
Part (b) [3 MARKS] This subquestion is about **min-heaps**.

i. Given an empty heap, draw the heap that results from inserting 1, 5, and 3, in that order:

ii. Draw the heap that results from inserting 2 into this heap:



iii. Draw the heap that results from removing the value at the root of this heap:



Part (c) [6 MARKS] This subquestion is about **AVL trees.**
Draw the AVL tree after each of the following operations, beginning with an empty tree. **In each answer, show the intermediate steps before and after each rotation.** In other words, an answer that involves no rotation would show only one tree, an answer that involves a single rotation would have two trees (showing the tree before and after that rotation), and an answer that involves two rotations would have three trees.

i. Insert 5:

ii. Insert 9:

iii. Insert 7:

iv. Insert 4:

v. Insert 2:

Part (d) [4 MARKS] This subquestion is about **Huffman trees.**
Given the following list of weights, draw the Huffman tree containing those weights. When you assemble two subtrees, place the one with the smaller weight on the left. If your answer is incorrect, part marks will be given if you show your work. Draw a box around your final Huffman tree so that we know what to mark.

Weights: `1 1 3 4 6 7`

Question 3. [10 MARKS]

Part (a) [6 MARKS] Complete the following function. *For full marks, you must not use any loops.*

```
struct Node {
    int val;
    Node *next;

    Node(int v, Node *nxt=0) : val(v), next(nxt) {}
};

// Move the first k nodes in the linked list pointed to by front to the end of the list,
// updating front and end appropriately.
// For example, if the list contained [a, b, c, d], and k was 2, then this function
// would move the a and b to the end, producing a linked list containing [c, d, a, b].
void moveToEnd(Node* &front, Node* &end, int k) {

}
```

Question 3. (CONTINUED)

Part (b) [1 MARK] What assumptions (if any) are you making about `k`?

Part (c) [1 MARK] What assumptions (if any) are you making about the number of nodes in the list?

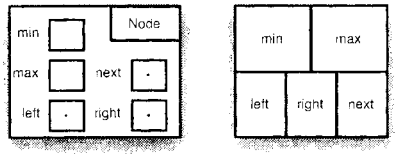
Part (d) [2 MARKS] What is the big-O running time of your `moveToEnd`?

Question 4. [11 MARKS]

Searching a sorted linked list for a particular value is slow. This question has you add a binary tree structure on top of a linked list by repeatedly adding rows of linked lists that also have left and right subtree pointers. The top row will only have one node, and that will be the root of the binary tree. We will call this a "PF tree".

Each node in this structure has a `next` pointer (for the next node in the same row), `left` and `right` pointers (for the subtrees), and `min` and `max` for the minimum and maximum values in the subtree rooted at that node—in a leaf, `min` and `max` will have the same value. Below is a definition for such a node. You are not allowed to change it. We also show two possible ways to draw such a `Node`; you are welcome to use either version in your answer to the first part.

```
struct Node {
    int min, max;
    Node *next;
    Node *left;
    Node *right;
};
```



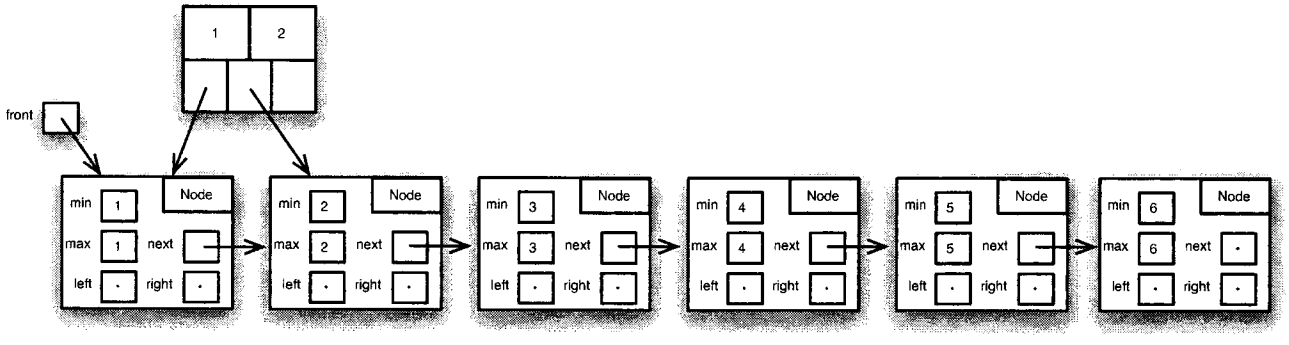
Part (a) [3 MARKS]

At the bottom of the next page, we provide a linked list and the first node in the row above it. (Note that in the bottom row we have used the more detailed version of a `Node` and in the row above we have use the "shorthand" version.) Finish the PF tree: complete the second row, and then draw all the other rows. Just like in a binary tree, every node you draw will have two children, so there will be three nodes in the row above the bottom row. The topmost row will have only a single node: the root of the whole tree.

You will need to decide what to do if there are an odd number of nodes in a row.

Question 4. (CONTINUED)

Part (a) (CONTINUED)



Part (b) [8 MARKS]

Complete the following function.

```
// Given a linked list pointed to by front, build a PF tree with this linked list as the
// bottom row, and return the root of the resulting tree.
Node *buildPF(Node *front) {
    // If there are 0 or 1 nodes in the linked list, we are done.
    if (! front || ! front->next)
        return front;
```

```
}
```

Question 5. [7 MARKS]

The Dutch national flag is divided into three horizontal colours, as exhibited to the right. Computer scientists worldwide are forced to think about this flag at some point in their lives. Today, it's your turn.

Red
White
Blue

BWAHAHAHAHAHA!

Oh, wait, sorry, this is a final exam! That may have been inappropriate... ;)

Part (a) [6 MARKS]

Computer scientists use numbers instead of colours. Complete the following function. *You must rearrange the values in place*: you are not allowed to count the number of each and then write 0's, 1's and 2's into the array! The faster your algorithm, the more marks you will get. Below is a helper function you can use in your solution.

```
// Rearrange the 0's, 1's, and 2's in list[0..size-1] so that the 0's come first, then
// the 1's, and then the 2's.
void dutch(int *list, int size) {




}

// Swap list[i] and list[j]-- you may call this function in your solution.
void swap(int *list, int i, int j) { int t = list[i]; list[i] = list[j]; list[j] = t; }
```

Part (b) [1 MARK]

What is the big-O time complexity of your solution? Explain how you obtained your answer.

Question 6. [15 MARKS]

Suppose you want to write your own application to keep track of birth dates of friends and family. You plan to use a hash table to store the records. Each record consists of the *month/day/year* of birth (all positive integers, with months in the range 1–12), along with the *name* of the person whose birth date this is (as a string).

Part (a) [2 MARKS]

Fill in the the hash value for each date (*month/day/year*) in the table below. Use the hash function $h(month, day, year) = (31 \cdot month + day) \bmod 10$.

Date	Hash value
03/14/1985	
02/07/1977	
06/19/1999	
10/11/1912	
08/14/1985	
06/25/2001	

Hash Table for Part (b)

0:	
1:	
2:	
3:	
4:	
5:	
6:	
7:	
8:	
9:	

Part (b) [2 MARKS]

Write each date in the hash table on the right, at the appropriate index (based on the hash values you computed above). Use quadratic probing to resolve collisions.

Part (c) [11 MARKS] Complete the code for class `BirthDates` on the next two pages. Use the following class to represent dates.

```
struct Date {
    unsigned char day;
    unsigned char month;
    unsigned short year;

    Date(unsigned char d = 0, unsigned char m = 0, unsigned short y = 0)
      : day(d), month(m), year(y) {}

    // Return the basic hash value (without the modulo) for this date.
    unsigned short hash() const { return day + 31 * month; }

    // Return whether this date is the same as the other date.
    bool operator==(const Date &other) const {
        return day == other.day && month == other.month && year == other.year;
    }
    // Return whether this date is different from the other date.
    bool operator!=(const Date &other) const { return ! (*this == other); }
};
```

Question 6. (CONTINUED)

Part (c) (CONTINUED)

```
// FOLLOW THE INSTRUCTIONS WRITTEN IN CAPITALS THROUGHOUT THE CODE.
// A collection of birth dates stored in a hash table.
class BirthDates {
private:
    unsigned num;       // Number of elements currently in the hash table.
    unsigned cap;       // Total number of slots in the hash table.
    Date *table;        // The array for the hash table (see below for initial values).
    static Date EMPTY; // Special value used to represent empty slots (defined below).

    // Return the location of date in the hash table, using quadratic probing and
    // starting at location index.  Return index if date is not in the table.  Note that
    // this can be used to find an empty table slot by calling it with date = EMPTY.
    // IMPLEMENT THIS FUNCTION.
    unsigned probe(const Date &date, unsigned index) const {









    }

public:
    // Initialize the collection to be empty.
    BirthDates(int c = 3) : num(0), cap(c), table(new Date[c]) {
        for (unsigned i = 0; i < cap; ++i)  table[i] = EMPTY;
    }
    // Free the memory allocated for this collection.
    ~BirthDates() { delete[] table; }
    // Return the number of dates stored in this collection.
    unsigned size() const { return num; }

    // Return whether or not date belongs to this collection.  IMPLEMENT THIS FUNCTION.
    bool contains(const Date &date) const {









    }
```

Question 6. (CONTINUED)

Part (c) (CONTINUED)

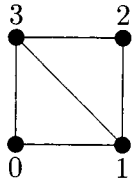
```
    // Remove date from this collection; do nothing if date is not in the collection.
    // IMPLEMENT THIS FUNCTION.
    void remove(const Date &date) {









    }

    // Insert date into this collection; do nothing if date is already present.
    // Double the size of the hash table and rehash every value if the load factor
    // exceeds 1/2.  Return true if insertion was successful; return false otherwise
    // (if no available slot was found).  IMPLEMENT THIS FUNCTION.
    bool insert(const Date &date) {









    }
};  // class BirthDates

Date BirthDates::EMPTY;   // Definition for the static member of class BirthDates.
```

Question 7. [12 MARKS]

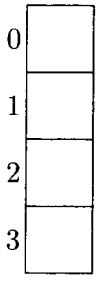
Part (a) [4 MARKS]

The picture below shows you a graph G together with an adjacency matrix and the "backbone" of an adjacency list for G. Fill in the correct values for the adjacency matrix and draw the correct adjacency list for G.

$G =$



	0	1	2	3
0				
1				
2				
3				



Part (b) [8 MARKS]

In the space below and on the next page, write a simple `Graph` class (to save you time and space, don't separate declarations from definitions: write the method implementations directly inside the class) implemented using *either* an adjacency matrix *or* an adjacency list. State *clearly and explicitly* which implementation you are attempting.

Your class should provide the public interface used below, along with any required private members (data or helper functions). Make sure to also include a destructor.

```
Graph g(4);            // constructor: initialize a graph with 4 vertices but no edge
g.insert(0,1);         // add an edge between vertices 0 and 1
g.insert(1,2);         // add an edge between vertices 1 and 2
g.insert(1,3);         // add an edge between vertices 1 and 3
if (g.hasEdge(1,2))    // return whether or not an edge exists
    g.remove(1,2);     // remove the edge between vertices 1 and 2
cout << g.numEdges(); // print the number of edges of g
```

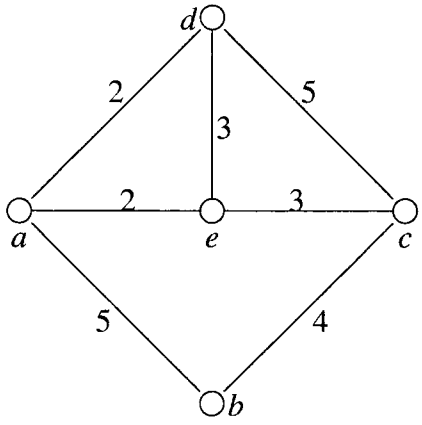

CONT'D...

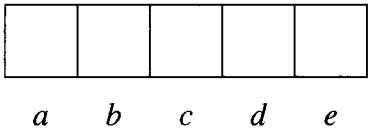
Question 7. (CONTINUED)
Part (b) (CONTINUED)

Question 8. [10 MARKS]

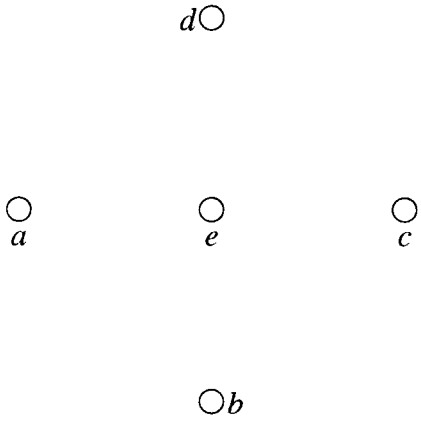
Part (a) [4 MARKS]

Trace Prim's algorithm on the following graph, starting from vertex e. When Prim's algorithm has to make a choice and there is more than one equivalent possibility, just break the tie any way you like. Use the array to keep track of the information you need during the execution of Prim's algorithm, but don't bother to redraw the entire array at each step: simply cross out and overwrite the old values in-place. Draw the final spanning tree returned by the algorithm using the vertices below the original graph.





Final Tree:



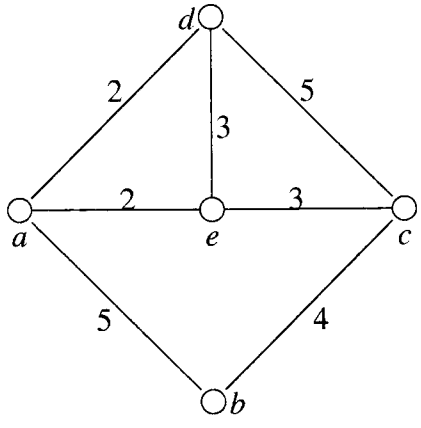
Question 8. (CONTINUED)

Part (b) [6 MARKS]

The following algorithm also finds a minimum-cost spanning tree in a weighted graph.

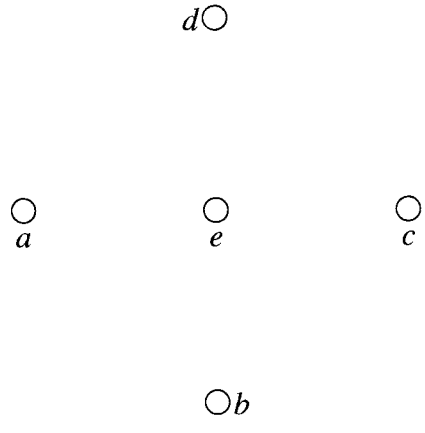
KRUSKAL'S ALGORITHM FOR INPUT $G = (V, E)$:
    initialize an empty set of edges T
    while T is not a spanning tree:
        select and remove from E an edge e with minimum weight
        if the endpoints of e are *not* already connected by edges in T:
            add e to T
    return T

Trace Kruskal's algorithm on the following graph. As the algorithm executes, use the table to record the edges that are added to T and the edges that are discarded (removed from E but not added to T), in the order that the algorithm considers them. For example, to include the edge from a to d, you would write (a, d) in the table. When Kruskal's algorithm has to make a choice and there is more than one equivalent possibility, just break the tie any way you like. Draw the final spanning tree returned by the algorithm using the vertices below the original graph.



edges added to T	discarded edges

Final Tree:



[Use the space on this page for rough work. Indicate clearly any work you want us to mark.]

CONT'D...

[Use the space on this page for rough work. Indicate clearly any work you want us to mark.]

[Use the space on this page for rough work. Indicate clearly any work you want us to mark.]