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

Final Examination
2007-04-10

Instructor: Douglas Wilhelm Harder
Time: 2.5 hours
Aides: none
18 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 144 marks. It will be marked out of 134.

A. **Algorithm Analsyis**

A.1 **[3]** Describe, in terms of limits, what it means for:

a. f(n) = **o**(g(n))
b. f(n) = Q(g(n))
c. f(n) = W(g(n))

A.2 **[2]** Using limits, find the asymptotic relationship between ln(n) and each of ln(n^2) and $\ln^2(n)$. Use the correct Landau symbol to show the relationship.

A.3 **[5]** Determine the run times of the following loops where **`bool f(int n)`** randomly returns true or false and runs in **O**(n) time. Use the most appropriate Landau symbol in each case.

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

b.
```
for ( int i = 0; i < n; ++i ) {
    if ( f(1) ) {
        for ( int j = i; j < n; ++j ) {
            ++sum;
        }
    }
}
```

c.

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = i; j < n; ++j ) {
        if ( f(1) ) {
            ++sum;
        }
    }
}
```

d.

```
for ( int i = 1; i < n; i *= 2 ) {
    ++sum;
}
```

A.4 **[3]** Suppose a teaching assistant has the choice of sorting a list of marked quizzes by last name, recording the marks, and then returning the examinations, or by simply going through the examinations in the order they were picked up, recording the marks, and then returning them. Assume that there are approximately $n = 100$ examinations and that students are approximately evenly distributed with respect to their last names across the $m = 26$ letters A through Z. Discuss the effort required in both cases.

B. **Lists, Stacks, Queues, Deques**

B.1 **[4]** Suppose we have a singly-linked list which has only a single member variable, a **`list_head`** pointer. What are the best possible run times of the following member functions (the behaviour of which is similar to that described in Project 1):

Member Functions	Run Time
`void push_front( const Object & );`	
`void push_back( const Object & );`	
`Object front() const;`	
`Object back() const;`	
`Object pop_front();`	
`Object pop_back();`	
`bool empty() const;`	
`int size() const;`	

B.2 **[7]** For this question, assume the given change was made to the data structure described in Question B.1 and indicate which functions could be written to run in **O**(1) time instead of **O**(*n*) time. Just give the function name, e.g., **`empty`**.

a. Assume we add the member variable **`int count`**. Which member functions can now be written to run in O(1) time?

b. Assume we add the member variable **`SingleNode<Object> * list_tail`**. Which member functions can now be written to run in O(1) time?

c. Suppose we change the singly-linked list to a doubly-linked list, that is, each node has both a previous and next pointer. Which member functions can now be written to run in O(1) time?

d. Suppose that in addition to changing the singly-linked list to a doubly-linked list, we also add the member variable **`DoubleNode<Object> * list_tail`**. Which member functions can now be written to run in O(1) time?

B.3 **[6]** Suppose we have a singly-linked list as described below which is storing integers but we would like to add the additional member function **`bool is_sorted() const`** which returns **`true`** if the elements in the list are sorted and **`false`** otherwise. It is easy to implement this new function so that it runs in O(*n*) time (*n* being the number of elements in the linked list) however, you can implement this function so that all member functions in this class runs in **O**(1) time? You will have to modify the other mutators.

```
class SingleNode {
    private:
        int element;
        SingleNode * next_node;
    public:
        SingleNode( int e, SingleNode * n ):element(e), next_node(n) {}
        int retrieve() const { return element; }
        SingleNode * next() const { return next_node; }
        friend class SingleList;
};
class SingleList {
    private:
        SingleNode * list head;
        SingleNode * list_tail;
        // insert other member variables here



    public:
        bool empty() const;  // do not implement
        bool is_sorted() const;
        int front() const;   // do not implement
        int back() const;    // do not implement

        void push_front();
        void push_back();
};
void SingleList::push_front( int n ) {









}
void SingleList::push_back( int n ) {









}
bool SingleList::is_sorted() {









}
```

C. **Trees**

C.1 **[3]** The following is a memory map of a binary search tree. The first row are the addresses, the second row are the contents of those addresses. Assume that each pointer and each value occupies one address. The member variable **`root`** which points to a binary-search-tree node is stored at address 08.

Each binary-search-tree node occupies three contiguous spaces, the first being the value, the second being the address of the left sub-node, and the third being the address of the right sub-node. Address 00 is used to represent empty sub-trees.

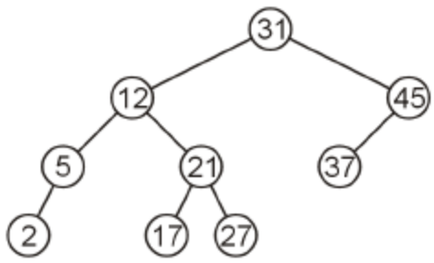
00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	21	00	00	33	0A	15		04		25	01	18		47	00

10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00					42	00	0E	27	00	1D			29	00	00

a. Draw a graphic representation of the tree.

b. Insert the node 30 and assume that the new operator was allocated the memory addresses 12, 13, and 14. Update the above memory map to insert this new node.

C.2 **[4]** List the elements of the tree



in the order in which they are visited for a:

a. breadth-first traversal:

b. pre-order depth-first traversal

C.3 **[6]** Implement the member function **`double_rotate_to_left`** which performs a double rotation on the given node to the left, as demonstrated in Figure C.3. The member function would be called on the node which is not balanced. You are not required to use the parameter to make this question work. You may implement helper functions on the back of the previous page.

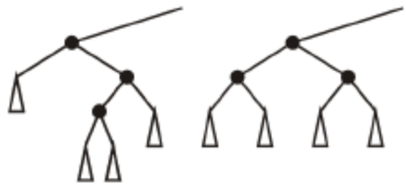


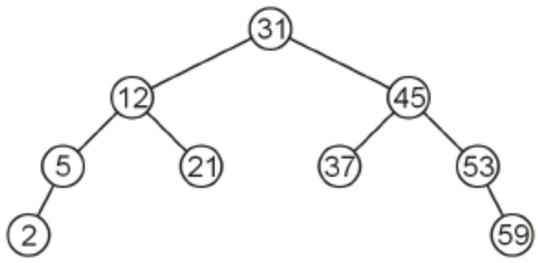
Figure C.3. The required double rotation.

```
template <typename T>
void AVLNode<T>::double_rotate_to_left( AVLNode<T>::to_this * & ptr ) {
```

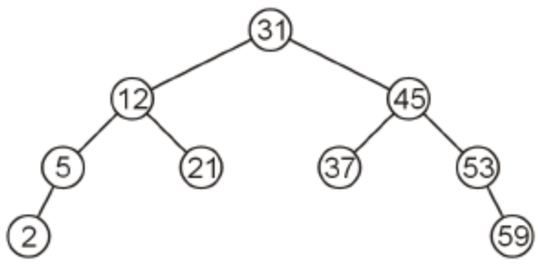
```
}
```

C.4 **[6]** For each of the given AVL trees, perform the given operation and perform whatever rotations are necessary to rebalance the tree. If no rotations are necessary, you can simply draw in or delete the appropriate node. If you are removing a full node, make the appropriate selection to the **right sub-tree**.

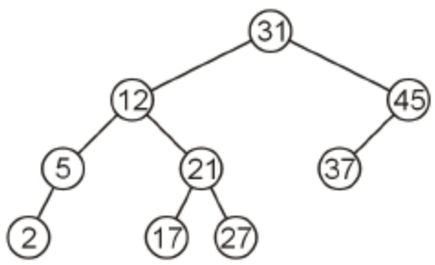
a. Insert 17.



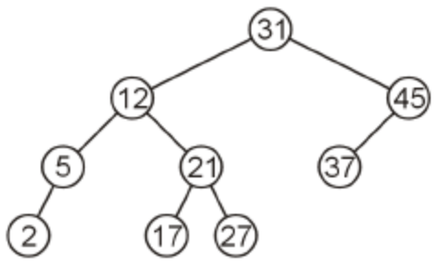
b. Insert 3.



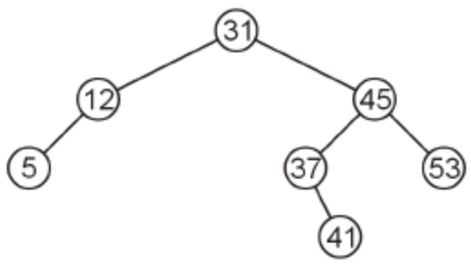
c. Remove 45.



d. Insert 47.

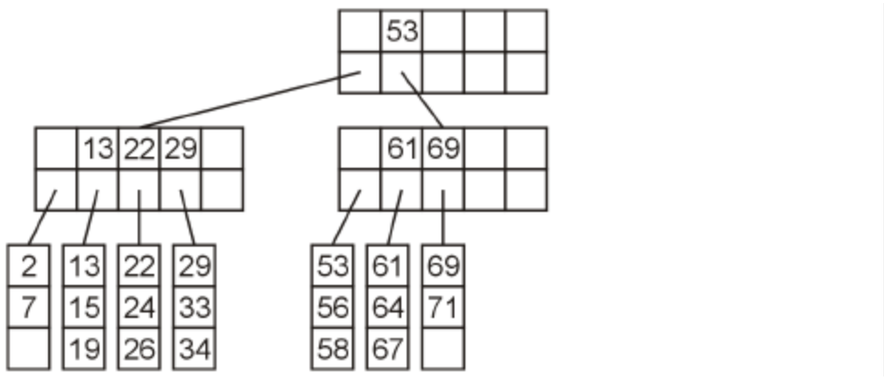


e. Remove 12.

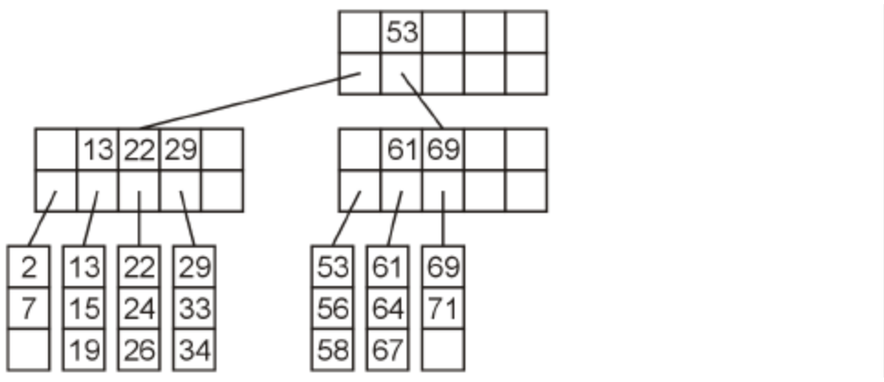


C.5 **[7]** For each of the given B-trees, perform the given operation and perform whatever changes are necessary maintain the B-tree structure. You may not use redistribution. You only need redraw those blocks which change.

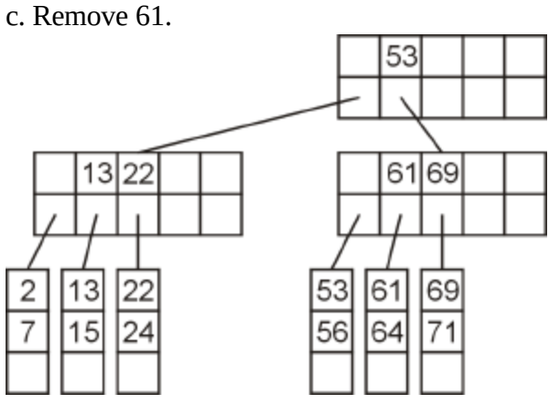
a. Insert 11.



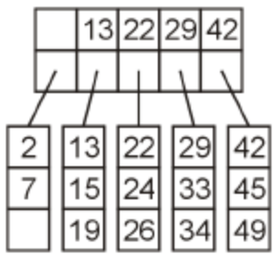
b. Insert 68.



c. Remove 61.



d. Remove 13.



C.6 **[2]** Correct the following statement: "If a block on the hard drive can hold a maximum of M next pointers, all internal blocks in a B-tree must point to at least M/2 nodes."

D. **Hash Tables**

D.1 **[2]** In the worst case, what is the minimum number of elements which can be stored in a hash table using open addressing for the following three techniques of probing (assume the size of the hash table is M).

Probing Technique	Worst Case
Linear Probing	
Quadratic Probing	
Double-hashing	

D.2 **[3]** Insert the following seven elements (in the given order) into a hash table of size 7 which uses quadratic probing. Use the least-significant digit to represent the initial bin. Stop after the first exception is thrown as a result of an invalid operation.

22 66 72 44 86 33 90

0	1	2	3	4	5	6

D.3 **[2]** We had two options for double hashing:

a. The hash table is of size M = 2^m and the skip size must be odd, or
b. The hash table is of size p where p is prime and the skip size is in 2, ..., $p - 1$.

Explain two benefits of using the first model. These benefits may relate to any of the operations which we may wish to perform on a hash table.

D.4 **[3]** What does the following algorithm do? The operator **`sizeof`** returns the number of bytes occupied by the variable *n*.

```
int a = 0;
int b = 1;

for ( int i = 0; i < 8*sizeof( n ); ++i ) {
    if ( n | b ) {
        ++a;
    }

    b <<= 1;
}
```

E. **Heaps**

E.1 **[6]** Using the implementation of a min-heap as discussed in class, insert the following numbers (in the order given) into an initially empty min-heap.

6 4 2 5 3 1

Indicate the final state of the min heap in the following array:

0	1	2	3	4	5	6

E.2 **[6]** In class, we saw a min-heap with two sub trees. A *d*-min-heap is a min-heap where each node has *d* sub trees, each of which is also a *d*-min-heap. In this case, it is easier to start at the array entry **0** and let the children of *i* be *di* + 1, *di* + 2, ..., *di* + *d*. Unfortunately, the code below to dequeue the minimum element has some bugs. Correct the code.

```
class DHeap {
    private:
        int * array;
        int count;
        int d;

    public:
        void dequeue_min( int );
        // ...
};

void DHeap::dequeue_min( int n ) {
    int value = array[0];

    int i = 0;

    while ( i < count ) {
        int min = array[i + 1];
        int posn = 1;

        for ( int j = 2; j <= d; ++j ) {
            if ( array[i + j] < min ) {
                max = array[i + j];
                posn = j;
            }
        }

        if ( array[count - 1] < min ) {
            array[i] = array[count – 1];
            return value;
        } else {
            array[i] = array[i + j];
            i = i + j;
        }
    }

    return value;
}
```

F. **Sorting**

F.1 **[3]** Perform one iteration (one pass through the array) of bubble sort on the following unsorted array. Write your answer in the next row.

7	5	3	8	2	9	10	1	6	4

F.2 **[4]** Convert the unsorted array into a max-heap using the heapify algorithm shown in class.

7	5	3	8	2	9	10	1	6	4

F.3 **[3]** Perform one iteration of quick sort (after the values are appropriately moved but before quick sort is recursively called on both halves). Use integer division to find the mid point. The first row represents the array entries. Write your answer in the next row.

0	1	2	3	4	5	6	7	8	9	10	11
43	21	93	87	23	57	19	35	77	54	96	12

F.4 **[4]** Apply radix sort to the following 3-bit binary numbers:

010 110 111 001 011 101 100 110

Place the intermediate lists (after dequeuing) into the following three tables. The last (sorted) list is given.

001	010	011	100	101	110	110	111

G. **Graphs**

G.1 **[4]** Perform a topological sort on the following graph with the following condition: if there is more than one eligible node which could appear next in the topological sort, choose that one which has the smallest value.

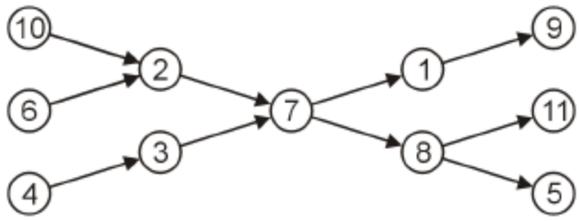
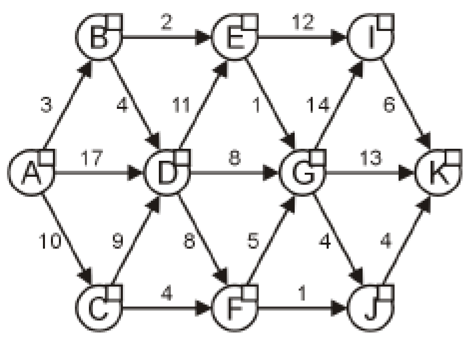


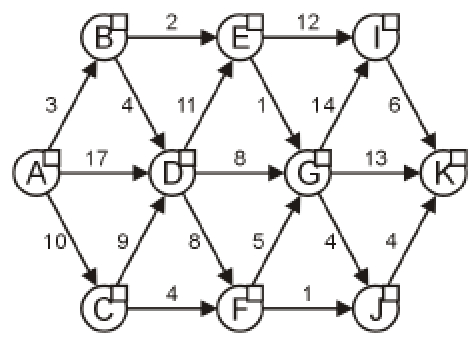
Figure G.1 A directed acyclic graph.

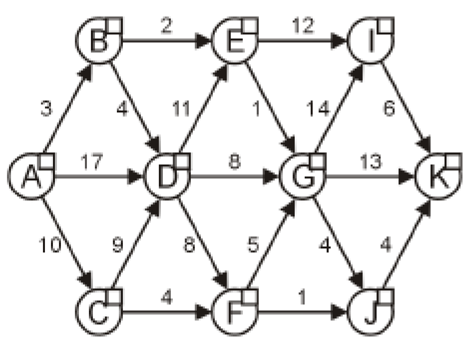
G.2 **[6]** The following is an implementation of Prim's algorithm, as done in Project 5. Indicate what must be modified to implement Dijkstra's algorithm to find the minimum distance between two vertices *v* and *w*. You should assume that the arguments do not require error checking and that a minimum path exists.

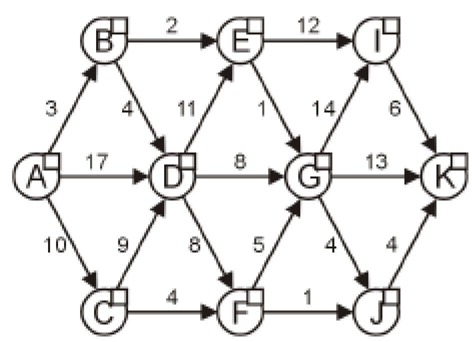
```
double WeightedUndirectedGraph::minimum_spanning_tree( int v ) const {
    double value = 0.0;

    double * table = new double[array_size];
    bool * visited = new bool[array_size];

    for ( int i = 0; i < array_size; ++i ) {
        table[i] = INF;
        visited[i] = false;
    }

    table[v] = 0.0;

    while ( true ) {
        bool found = false;
        double max = INF;
        int posn;

        for ( int i = 0; i < array_size; ++i ) {
            if ( !visited[i] && table[i] < max ) {
                posn = i;
                max = table[i];
                found = true;
            }
        }

        if ( !found ) {
            delete [] table;
            delete [] visited;

            return value;
        }

        visited[posn] = true;
        value += max;

        for ( int i = 0; i < array_size; ++i ) {
            if ( !visited[i] && adjacent( i, posn ) < table[i] ) {
                table[i] = adjacent( i, posn );
            }
        }
    }
}
```

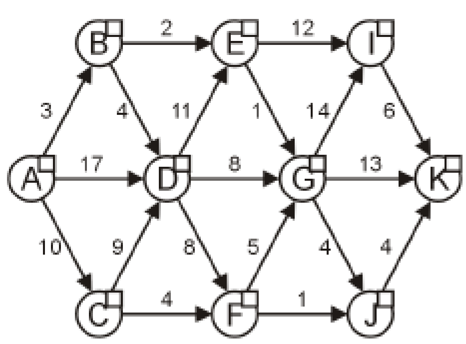
G.3 **[6]** Use Dijkstra's algorithm to find the **length** of the minimum path from vertex A to vertex K. You **must** use a separate graph for each visited node in the graph. Indicate the order in which you are using the graphs (1, 2, 3, ...) and cross out any unused graphs. You do not have to record the pointers. Use the check box to indicate which vertices have been visited.

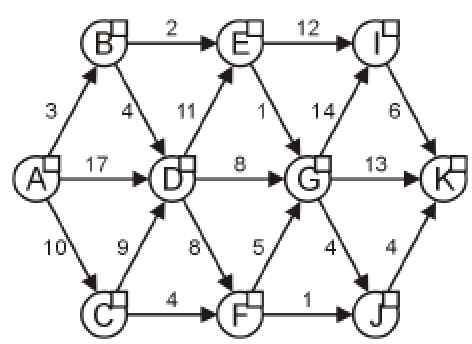


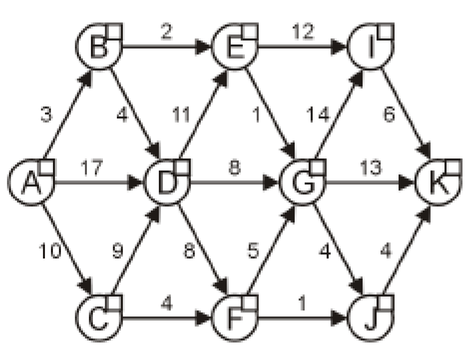


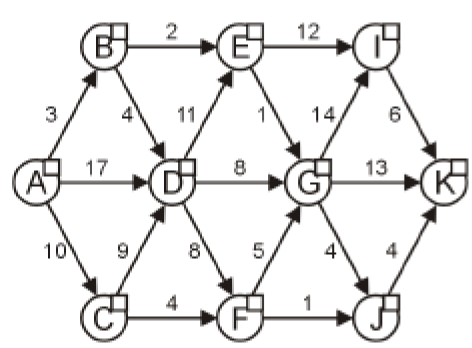


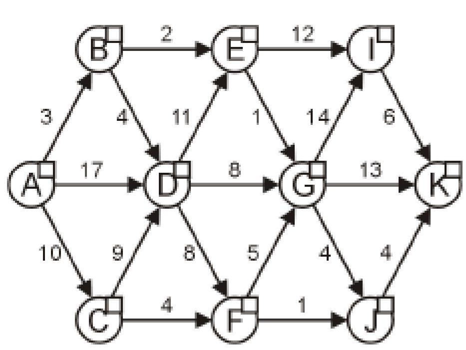


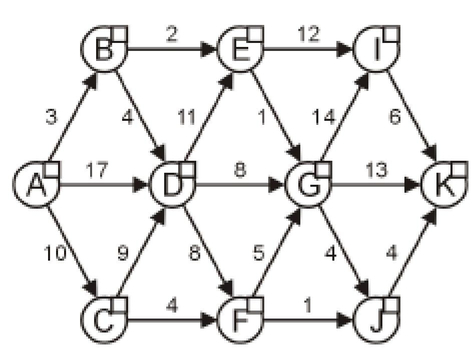












H. **Algorithms**

H.1 **[4]** Suppose a number of projects are being proposed for the next cycle of a product release. There are n projects, where each is associated with a projected revenue and an expected completion time. Justify heuristically why using a greedy algorithm using the highest cost density (expected revenue over completion time) is better than using a greedy algorithm which uses the highest expected revenue.

H.2 **[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 $T(n) = \mathbf{O}(n^k)$.

H.3 **[3]** Implement a version of factorial which stores the values of $n!$ when $n < 100$, thus, the second time the function is called with a particular argument, the run time is $\mathbf{O}(1)$. Use the given array.

```
int array[100];
array[0] = 1;
for ( int i = 1; i < 100; ++i ) {
    array[i] = 0;
}

int factorial( int n ) {









}
```

H.4 **[3]** Describe the ideal shape of a skip list with 15 entries.

I. **Sparse Matrices**

I.1 **[4]** Show how you would store the following matrix using the Harwell-Boeing sparce matrix format by filling in the appropriate tables below.

$$\begin{pmatrix} 3 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -3 & 0 & 0 \end{pmatrix}$$

Member Variable	1	2	3	4	5	6	7	8
`double * value;`								
`int * row;`								

Member Variable	1	2	3	4	5	6	7
`int * column;`							

I.2 **[4]** Suppose that you are writing a function to return the (i, j)th entry of a matrix stored in the Harwell-Boeing format. Assume the matrix is $n \times n$ and that the N entries are evenly distributed among the rows and columns. Describe (using pseudo-code) how you could implement the access function so that it runs in $\mathbf{O}(\ln(N/n))$ time.

J. **Unix**

J.1 **[5]** Assume that your home directory contains the four files **`Tester.h TestTester.h TestDriver.cpp t01.in`** and that you have just logged in. Create a directory called **`project`**, copy the files into that directory, change to that directory, edit a new file **`Test.h`** using whichever Unix editor you prefer, compile the appropriate tester file (assume it compiled), and then run the resulting executable by redirecting the contents of the **`t01.in`** file. Finally, exit the system.

`{eceunix:1}`

`{eceunix:2}`

`{eceunix:3}`

`{eceunix:4}`

`{eceunix:5}`

`{eceunix:6}`

`{eceunix:7}`

K. **Self Study**

K.1 [5] Describe, in your own words, either splay trees or disjoint sets. You should describe the purpose, appropriate implementations, and properties. Provide references.