ECE 250 *Algorithms and Data Structures*
FINAL EXAMINATION

Douglas Wilhelm Harder dwharder@uwaterloo.ca EIT 4018 x37023

2012-12-13T12:30P2H30M

Rooms: RCH 301, 302

Instructions:

There are 99 marks. It will be marked out of 90.
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 the examination period.
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								

IF YOU ARE WRITING THIS AS A *SUPPLEMENTAL* EXAMINATION TO CLEAR A PREVIOUSLY FAILED COURSE, PLEASE CHECK HERE: ☐

I have read the above instructions:

Signature: ____

Asking Any Question
-5

A Relations and Asymptotic Analysis

A.1 [2] Describe the difference between a linear ordering and a weak ordering.

A.2 [2] Explain how a tree can be used to store a hierarchical ordering.

1. It is never true that $x < x$,
2. If $x < y$, $y \nless x$, and
3. If $x < y$ and $y < z$, $x < z$.

A.3 [3] You are given two algorithms, `void algorithm_A( int *, int n )` and `void algorithm_B( int *, int n )`, and you are aware that the first has quadratic run-time characteristics while the second has of *n*-log-*n* run-time characteristics. You try each algorithm for various values of *n* and plot the run times to get the chart in Figure A.3. Implement an hybrid algorithm of these two, calling each as appropriate.

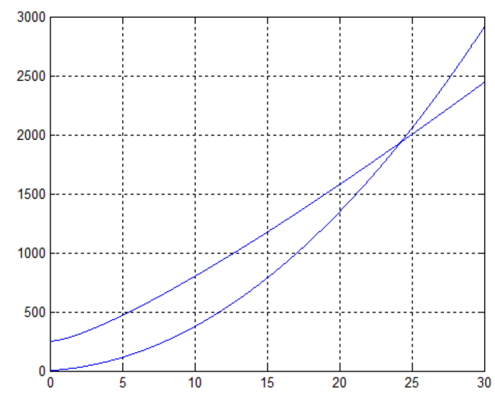


Figure A.3. A chart of run times.

```
void algorithm( int *array, int n ) {



}
```

A.4 [3] Write down the run time of

```
void FFT( std::complex<double> *array, int n ) {
    if ( n == 1 ) return;
    double const PI = 4.0*std::atan( 1.0 );
    std::complex<double> w = 1.0;
    std::complex<double> wn = std::exp( std::complex<double>( 0.0, -2.0*PI/n ) );
    std::complex<double> even[n/2];
    std::complex<double>  odd[n/2];
    for ( int k = 0; k < n/2; ++k ) {
        even[k] = array[2*k];
         odd[k] = array[2*k + 1];
    }
    FFT( even, n/2 );
    FFT(  odd, n/2 );
    for ( int k = 0; k < n/2; ++k ) {
        array[k]       = even[k] + w*odd[k];
        array[n/2 + k] = even[k] - w*odd[k];
        w = w * wn;
    }
}
```

by completing the following recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ & n \geq 1 \end{cases}$$

Based on other algorithms with this same run-time relationship, fill in $T(n) = \Theta(\quad)$

B Midterm Questions

B.1 [4] 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 B.1 would be

```
A0_B1_E2_C1_D1_F2_G2_
```

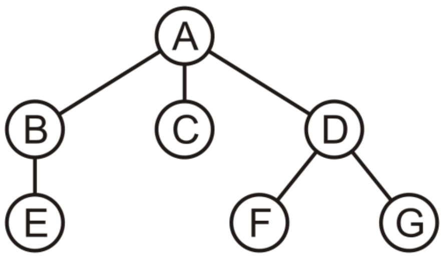


Figure B.1. A tree.

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 {
```

B.2 [4] 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 B.2 may be used to determine how you should implement the algorithm. Recall that you can use `nullptr` to indicate that a search on a sub-branch was unsuccessful.

```
template <typename Type>
Binary_search_node<Type> *Binary_search_tree<Type>::next_largest( Type const &obj ) {
        return 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 {









        }
}
```

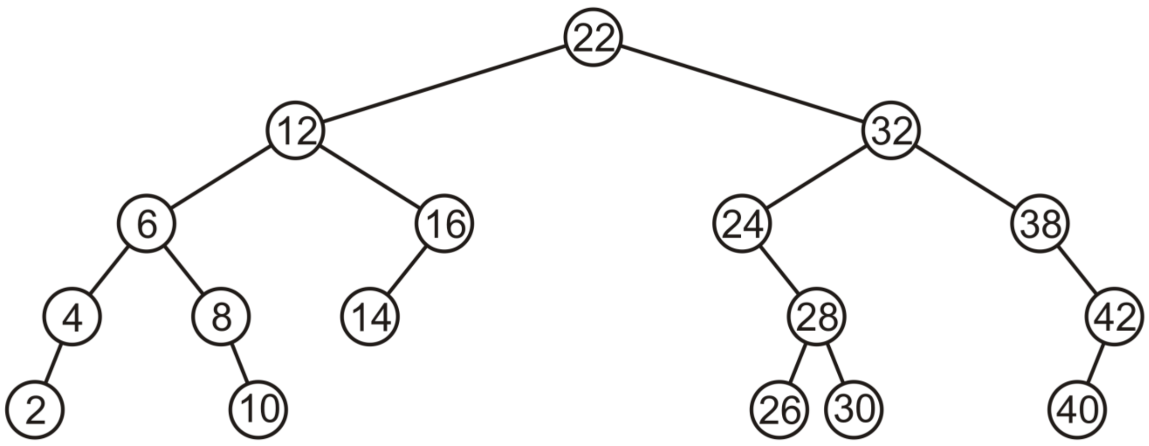


Figure B.2. A binary search tree.

C Linearly Ordered Array-based and Node-based Data Structures

C.1 [4] You'd like to change your `Single_list` implementation to be a priority queue with the smallest object at the front. You will have to replace

```
      void push_front( Type const & )
      void push_back( Type const & )
```

with a single push function. Being smart, instead of removing the `push_front` routine, you make it a private member function so that you can still call it, but a general user of the class cannot. Implement the push function:

```
template <typename Type>
void Single_list<Type>::push( Type const &obj ) {











}
```

C.2 [2] Would you use such a class as implemented in Question C.1 for a priority queue in general? Would this be a good design if it was known that, on average, the priority queue was usually empty?

C.3 [5] Suppose we have a queue where the capacity doubles when a new object is inserted into a full queue, and where the capacity halves when, after deletion, the queue is **one-third** full. It is claimed that, in the worst case and in the long term, approximately 2.5 copies are made during the processes of resizing the array for each push and pop. Is this claim reasonable? You might want to start by experimenting with a queue of size *n* = 9.

D Hierarchical Orderings and Tree-Based Data Structures

D.1 [3] Determine a formula for the height of a complete tree with 2^h nodes and prove your claim by using induction. You may assume that a tree of height h has 2^h leaf nodes.

D.2 [2] In class, we discussed how we could use a queue for a breadth-first traversal. Describe the traversal that would result if we used a priority queue (where the minimum element is on the top) as a container and used this to perform a traversal on a binary search tree. You may wish to refer back to Figure B.2.

D.3 [2] What does the following function do for a simple tree that has no duplicate elements? If both trees are height h, what is the best description of the run time of this algorithm? Recall that a simple tree has children that are themselves simple trees and that `parent()` returns a pointer to the parent of the current node.

```
template <typename Type>
Type Simple_tree<Type>::f( Simple_tree<Type> *ptr ) {
    Single_list<Type> s1, s2;

    Simple_tree *ptr = this;

    while ( ptr != nullptr ) {
        s1.push_front( ptr->retrieve() );     ptr = ptr->parent();
    }

    ptr = tree;

    while ( ptr != nullptr ) {
        s2.push_front( ptr->retrieve() );     ptr = ptr->parent();
    }

    Type rv;

    while ( s1.front() == s2.front() ) {
        rv = s1.front();
        s1.pop_front();    s2.pop_front();
    }

    return rv;
}
```

E Linearly Ordered Tree-Based Data Structures

E.1 [7] Given the AVL tree shown in Figure E.1, perform the required operations. In each case, you only need to draw the parts of the tree that changed.

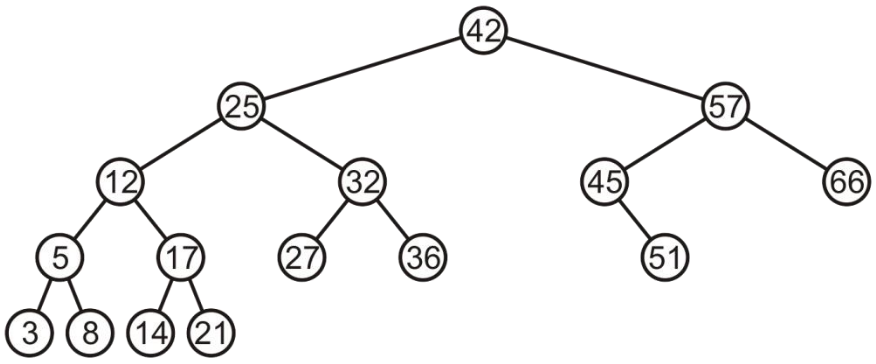


Figure E.1. An AVL tree.

a. Into the AVL tree in Figure E.1, insert 13.

b. Into the AVL tree in Figure E.1, insert 9.

c. From the AVL tree in Figure E.1, erase 66.

E.2 [3] Provide an argument as to whether the depth of a leaf node in an AVL tree is o(ln(n)), O(ln(n)) or Θ(ln(n)).

Figure E shows a B+-tree with $M = L = 5$. In questions E.3 and E.4, perform the appropriate insertions in the correct leaf nodes and if a node is full, it must be split.

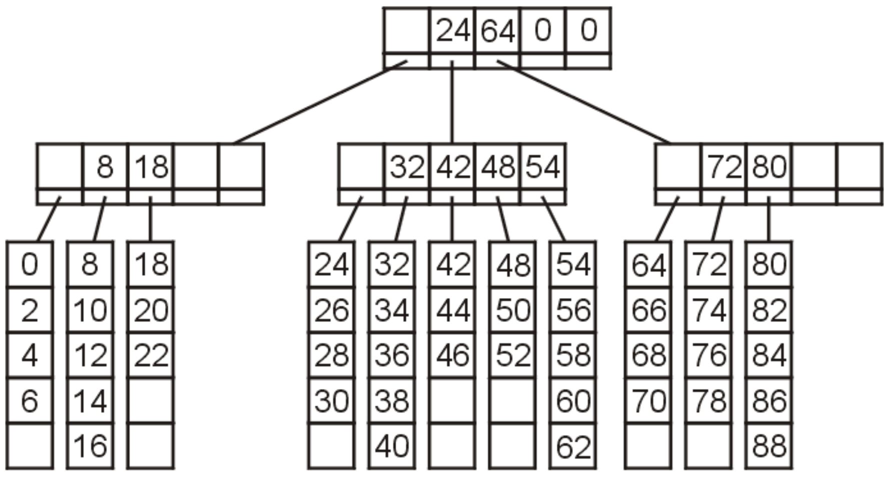


Figure E. A B+-tree.

E.3 [3] Redrawing only those nodes which changed, insert 59 into the B-tree shown in Figure E.

F Binary Min Heaps

F.1 [3] Pop the front of the binary min-heap shown in Table F.1a three times. Enter the resulting heap in Table F.1b. If you wish, show your intermediate answers on the reverse of the previous page.

Table F.1a. A binary min heap.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	1	2	5	6	4	12	17	9	8	15	13	23	35	18	24	20	10	

Table F.1b. Your answer.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

F.2 [2] Suppose that the top is popped from the binary min-heap shown in Figure F.2. Only two entries are shown, but assume that all the other 24 entries are unique and satisfy the definition of a min-heap. After we have popped the top element using our algorithm shown in class, the value 42 will be moved to the root and percolated down. Mark with an X all nodes within the heap where it will be impossible for the value 42 to appear under any possible intermediate values.

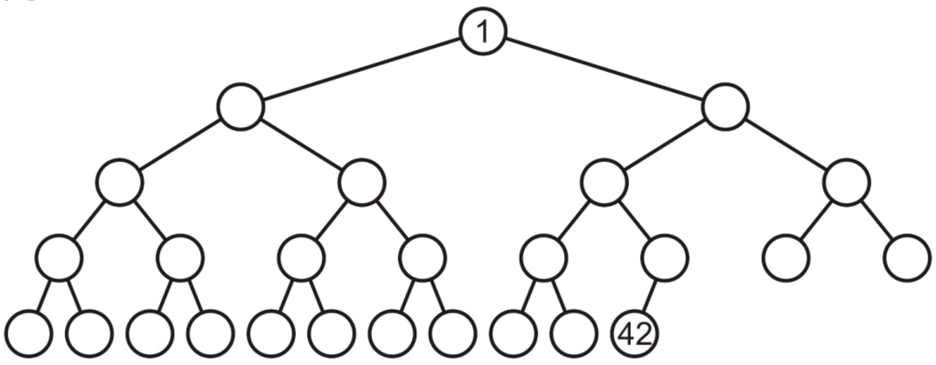


Figure F.2. A binary min-heap.

G Hash Tables

G.1 [3] Consider Project 5 where the nodes in a directed acyclic graph are associated with priorities. The priorities were required to be unique. Thus, when setting a priority, it is necessary to search through all of the priorities to ensure that the new priority is not a duplicate of one that has already been applied. Such a search is O(*n*). Suppose, instead, you store the priorities in a hash table of size of 4*n*. Would you recommend linear probing or double hashing? Recall that the average number of bins checked during an unsuccessful search are given by $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$ and $\frac{1}{1-\lambda}$, respectively.

G.2 [4] Using the least-significant digit as the hash function when using linear probing, remove the following three entries from the hash table in Table G.2*a*, in the order given, and put your answer in Table G.2*b*.

926, 488, 251

Table G.2*a*. The initial hash table.

0	1	2	3	4	5	6	7	8	9
108	251	681	353	207		926	496	488	477

Table G.2*b*. Your result.

0	1	2	3	4	5	6	7	8	9

G.3 [2] For double hashing, if the size of the table was a power of two, we required that the second hash function is odd. What are the restrictions on the second hash function if the size of the hash table is prime?

H Sorting Algorithms

H.1 [2] Define an inversion in an unsorted list with entries $a_1, \ldots, a_n$.

H.2 [2] Even though insertion sort runs in $\mathbf{O}(n^2)$ time, why would it be preferable to use insertion sort over quicksort or merge sort for small arrays.

H.3 [3] Suppose we are implementing quicksort and we have two indices pointing to entries within the array where

`array[lower] > pivot` and `array[upper] < pivot`.

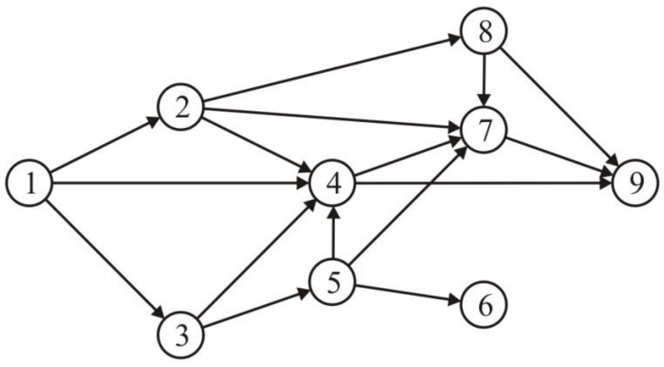
What is the minimum and maximum number of inversions that can be removed if we swap the two entries? You may express your answer under the assumption that that there are k entries strictly between `lower` and `upper`.

Hint: consider the following tables where the pivot is 42:

72	11	22	33	44	55	66	**35**
72	40	45	50	55	60	65	**35**
72	40	50	60	70	80	90	**35**

I Graph Algorithms

I.1 [2] To determine whether or not two vertices in a graph are connected, one possibility is to do a breadth-first traversal of the graph using a queue. In the example below, suppose we are attempting to determine if 1 and 6 are connected. Why is essential to ensure that each node is only enqueue once into the queue?



$$\begin{tikzcd}
 & & & 8 \arrow[d] \arrow[drr] & & \\
 & 2 \arrow[urr] \arrow[rr] \arrow[dr] & & 7 \arrow[rr] & & 9 \\
1 \arrow[ur] \arrow[rr] \arrow[ddr] & & 4 \arrow[ur] \arrow[urrr] & & & \\
 & & 5 \arrow[u] \arrow[uur] \arrow[r] & 6 & & \\
 & 3 \arrow[uur] \arrow[ur] & & & &
\end{tikzcd}$$

I.2 [4] In light of the discussion in Question I.1, the following is an unsafe implementation of a connected algorithm on a directed acyclic graph:

```
bool Directed_acyclic_graph::connected( int i, int j ) const {
    if ( i < 0 || j < 0 || i >= NUM_VERTICES || j >= NUM_VERTICES ) {
        throw illegal_argument();
    }

    if ( i == j ) {
        return true;
    }

    std::queue q;
    q.push( i );

    while ( !q.empty() ) {
        int vertex = q.top();
        q.pop();

        for ( int k = 0; k < NUM_VERTICES; ++k ) {
            if ( adjacent( vertex, k ) ) {
                if ( k == j ) {
                    return true;
                }

                q.push( k );
            }
        }
    }

    return false;
}
```

Modify this function to be safe to ensure that a vertex is never enqueued twice. You do not have to write out the entire function again, only make it clear what is being added or modified.

I.3 [2] Suppose we find a minimum spanning tree of a graph. Either prove or give a counterexample against the assertion that the shortest path between two nodes must follow the edges of the minimum spanning tree.

I.3 [4] Apply Dykstra's algorithm to find the minimum distance from vertex C to vertex J of the graph shown in Figure I.3. A number of copies of the graph are provided for your work. You will place the final state of the table in Table I.3. This will include the minimum distance to each of the vertices at the time the algorithm ends, whether a vertex was visited or not, and the parent of the edge leading to that vertex.

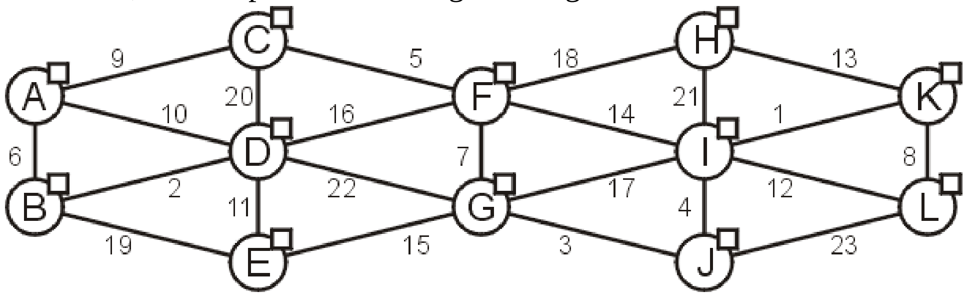
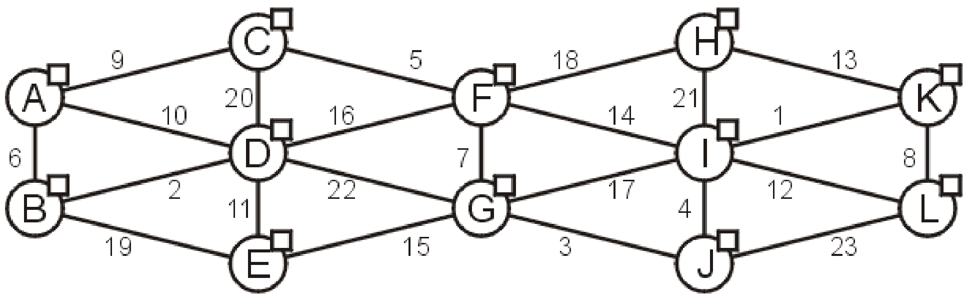
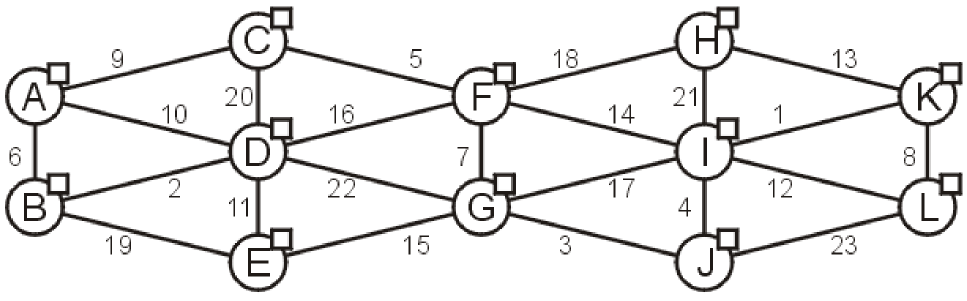


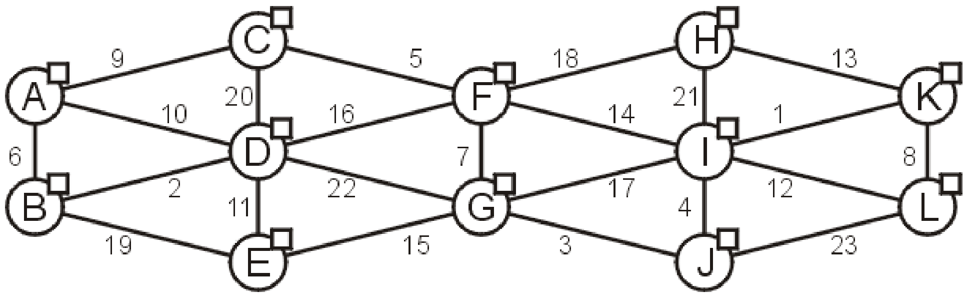
Figure I.3. An undirected weighted graph.

Table I.3. The final state of the table with the minimum distance to each node and visited vertices.

	A	B	C	D	E	F	G	H	I	J	K	L
Distance			0									
Visited												
Parent												







J Algorithm Designs

J.1 **[3]** Show how we can derive the run time of a recursive algorithm which runs according to $\mathrm{T}(n) = a\mathrm{T}(n/b) + \mathbf{O}(n^k)$. Assume that $n = b^m$ and show how

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

may be simplified when it is known $b^k < a$.

J.2 [1] In Question I.2, we stored additional information to track those vertices that had already been visited. What algorithm design technique is this?

J.3 [4] The following is an implementation of calculating the coefficients of the Newton polynomials:

```
double newton( int i, int j, double *x, double *y ) {
       assert ( i < j );
       if ( i == j ) {
              return y[i];
       } else {
              return (newton(i, j – 1) – newton(i + 1, j ))/(x[i] – x[j]);
       }
}
```

If $j - i + 1 = n$, fill in the question marks in this description of the run time

$$\mathrm{T}(n) = \begin{cases} \Theta(?) & n = 1 \\ ?\mathrm{T}(?) + \Theta(?) & n > 1 \end{cases}$$

How does this compare with the use of the divided difference table which allows the calculation of these coefficients in $\mathbf{\Theta}(n^2)$ time?

K Sparse Matrices

K.1 [2] What matrix is represented by the following old-Yale representation of a sparse matrix?

0	1	2	3	4	5	6
0	2	4	6	9	11	12

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	3	3	4	6	4	5	5
1.3	2.4	4.2	3.5	3.2	4.5	0.3	2.1	4.5	2.4	2.1	1.1

L Bonus Questions

L.1 [3] Demonstrate how the sets {2, 9}, {4, 11}, {1, 3, 5, 7, 8}, {0, 6, 10} may be represented using the disjoint set data structure (there are multiple solutions; however, there is one which is easiest).

0	1	2	3	4	5	6	7	8	9	10	11

L.2 [2] Which entry or entries of the array would you change in Question L.1 if you were to take the union of the sets containing 3 and 11? (Your answer will depend on your answer in Question L.1.)

L.3 [1] Is a splay tree a binary search tree? (Yes or No)

L.4 [4] Demonstrate how the splay tree in Figure L.4 would change if a search is made for the number 7.

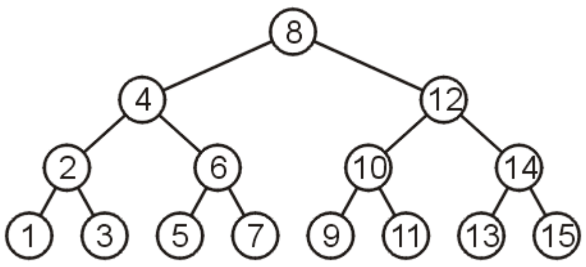


Figure L.4 A splay tree.