CSE 373 Spring 2011 Final

1. Sorting (12 Points)

For the following questions, indicate which sorting algorithm will successfully sort the elements in the least amount of time. Additionally, for each sorting algorithm, indicate the expected runtime of the algorithm in terms of Big-Oh under the conditions given.

- Choose between **bubble sort**, **selection sort**, **insertion sort**, **merge sort**, **quick sort**, and **bucket sort**.
- The algorithm chosen should be the one that completes fastest, without crashing.
- Assume that bubble sort is *not* optimized in any way.
- Assume that the quick sort algorithm chooses the first element as its pivot at each pass.
- Assume stack overflow occurs on 5000+ stacked method calls.

a. array size 700000, ascending order

Sorting Algorithm: ______________________________

Expected Runtime: ______________________________

b. array size 350000, random order, no extra memory may be allocated

Sorting Algorithm: ______________________________

Expected Runtime: ______________________________

c. array size 1000000, descending order

Sorting Algorithm: ______________________________

Expected Runtime: ______________________________

d. array size 2500000 containing zip codes (i.e. values between 0 - 99999), random order

Sorting Algorithm: ______________________________

Expected Runtime: ______________________________

2. AVL Trees (10 Points)

Given the following AVL Tree:

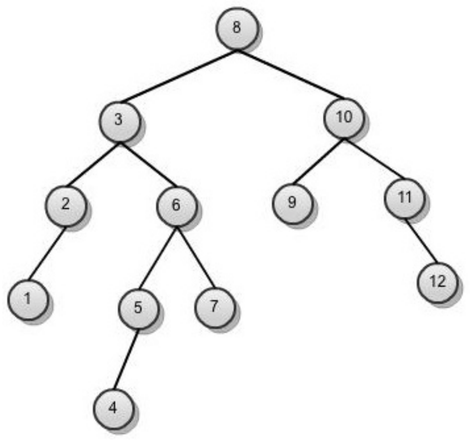


Figure 1 AVL Tree for Deletion

a. Draw the resulting **BST** after **10** is removed, but *before* any rebalancing takes place. Label each node in the resulting tree with its **balance factor**. Hint: Replace a node with both children using the maximum value from the node's left subtree.

b. Now rebalance the tree that results from (a). **Draw a new tree** for each rotation that occurs when rebalancing the AVL Tree (you only need to draw one tree that results from a RL/LR rotation). You do not need to label these trees with balance factors.

3. Heap Implementation (12 Points)

Part A: Implementation

In lecture we wrote a class named `IntBinaryHeap` that was an implementation of the Priority Queue ADT for `int`s. Implement a method named `delete` which takes a position (i.e. an index) as an argument and deletes the node at the given position from the heap while maintaining the heap structure and order properties (even if the node isn't the root and its item isn't the minimum). The method does not return anything, but does make modify the heap.

If an invalid position is given as an argument, a `NoSuchElementException` should be thrown. Recall that our heap was implemented with an array and the root element (i.e. the most minimum element) of the heap was stored at index 1 (not 0) to make the calculations for finding parents/children easier.

For example, if our initial heap `h` was `[0, -32, -27, -2, 7, 41, 2, 11, 38, 20]`, the following calls to the `delete` method would produce the following resulting results:

Method Call	**State of the Heap after `delete`**
`h.delete(0)`	`NoSuchElementException` thrown
`h.delete(1)`	`[0, `**`-27`**`, 7, -2, 20, 41, 2, 11, 38, 0]`
`h.delete(4)`	`[0, -32, -27, -2, `**`20`**`, 41, 2, 11, 38, 0]`
`h.delete(9)`	`[0, -32, -27, -2, 7, 41, 2, 11, 38, `**`0`**`]`
`h.delete(13)`	`NoSuchElementException` thrown

Hints: `Integer.MIN_VALUE` is useful.

```
public class IntBinaryHeap implements IntPriorityQueue {
    private int[] array;
    private int size;

    // Inserts the specified value into this priority queue.
    public void add(int value) { … }

    // Returns true if this priority queue contains no elements.
    public boolean isEmpty() { … }

    // Retrieves, but does not remove, the value at the top of this priority queue.
    public int peek() { … }

    // Retrieves and removes the value at the top of this priority queue.
    public int remove() { … }

    // Bubbles the last value up, maintaining the heap order property.
    private void bubbleUp() { … }

    // Bubbles the value at the ith index up, maintaining the heap order property.
    private void bubbleUp(int i) { … }

    // Bubbles the first value down, maintaining the heap order property.
    private void bubbleDown() { … }

    // Bubbles the value at the ith index down, maintaining the heap order property.
    private void bubbleUp(int i) { … }

    // YOUR METHOD GOES HERE!!
}
```

Part B: Analysis

What is the worst case runtime of your `delete` method. Give your answer in Big Oh notation and explain how you arrived at your answer. Be explicit about where terms used in your Big Oh come from.

4. Hashing (12 Points)

Simulate the behavior of a **hash set** given the following conditions:

- we are storing values of type `int` into a hash table with array length of **10**
- the set uses **quadratic probing** for collision resolution
- the hash function uses the `int` value, mod the size of the table, plus any probing needed
- the table does not enlarge or rehash itself during this problem
- an insertion **fails** if more than half of the buckets are tried or if the **properties of a set** are violated
- the table stores a special character "R" for any array entry that is used but later removed

a. Given the following lines of code, draw the final state of the hash table below. Leave a box empty if an array element is unused.

```
Set<Integer> set = new HashSet<Integer>(10);
set.add(53);
set.add(21);
set.add(82);
set.add(55);
set.add(92);
set.add(66);
set.add(10);
set.add(86);
set.add(55);
set.remove(21);
set.remove(66);
set.add(22);
set.add(75);
```

b. Did any values fail to be inserted? If so, which ones and why did the insertion fail?

c. What is the size of the final `set`?

d. What is the capacity of the final `set`?

e. What is the load factor of the `set`?

5. Topological Sort (10 Points)

Do a topological sort of the following graph. List the vertices as they would appear in a valid topological sort ordering.

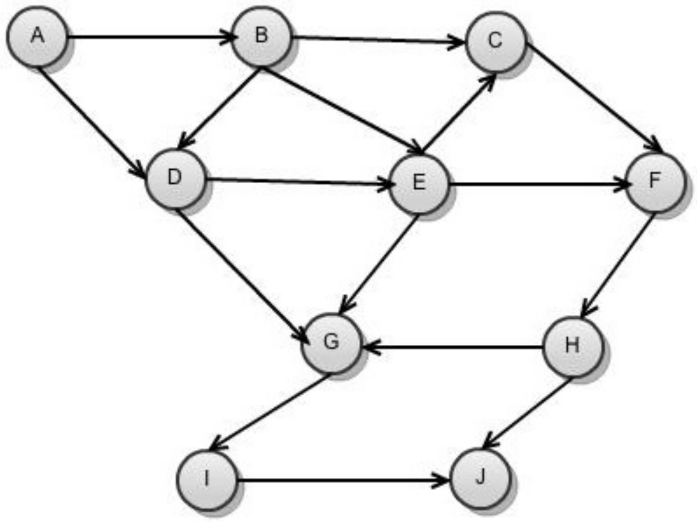
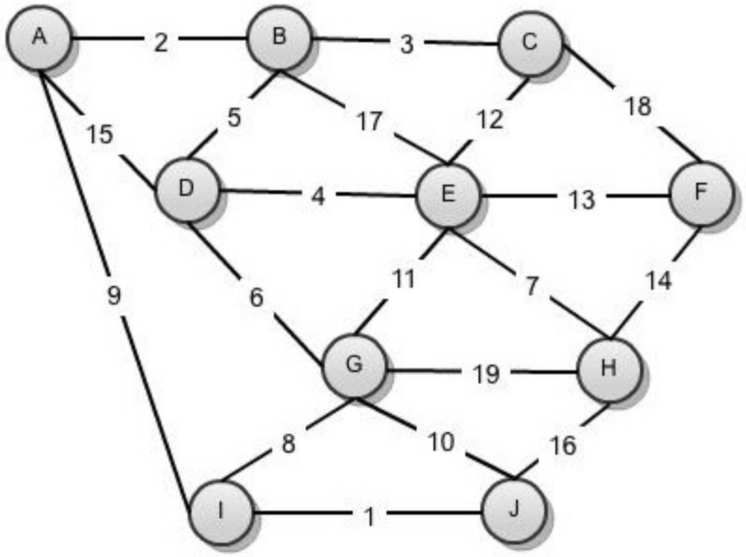


Figure 2 Graph on which to do a Topological Sort

6. Minimum Spanning Trees (12 Points)



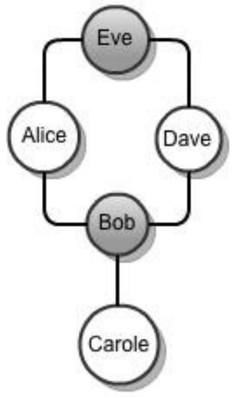
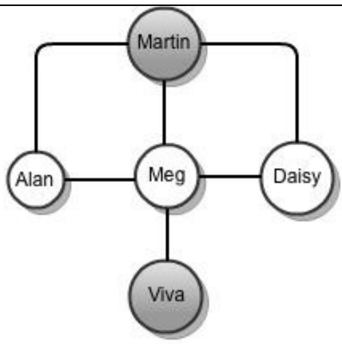
a. Using Prim's Algorithm starting with vertex "A", list the vertices in the order they are added to the MST.

b. Using Kruskal's Algorithm, list the edges of the MST in the order that they are added.

7. Graph Implementation (12 Points)

Write a method named `friendsAndEnemies` that could be added to the `Graph` class from Programming Project #4. The method accepts three parameters: a vertex, *v*, and two `Set`s. The method puts all vertices into one of the two given sets, the first representing *friends* and the second representing *enemies*, according to the following rules:

- **The vertex parameter, *v*, is a *friend*.** In Graph 1 `"Eve"` is the vertex passed so `"Eve"` is in the resulting *friend* set (i.e. `f`). Similarly, in Graph 2 `"Martin"` is in the resulting *friend* set (i.e. `f`).
- **If a vertex is a *friend*, all of its neighbors are *enemies*.** In Graph 1 below, `"Eve"` is a *friend* so all of her neighbors (`"Alice"`, `"Dave"`) are in the resulting *enemy* set (i.e. `e`).
- **If a vertex is an *enemy*, all of its neighbors are *friends*.** In Graph 1 below, `"Alice"` is an *enemy* so all of her neighbors (`"Eve"`, `"Bob"`) are in the resulting *friend* set (i.e. `f`).
- **If there is a way for a vertex to be both a *friend* and an *enemy*, use the shortest path between the parameter *v* and the vertex to determine which set the vertex should be in.** In Graph 2 below, `"Martin"` is a *friend* so `"Meg"` is an *enemy*. However, `"Meg"` is also a *friend* since `"Daisy"` is an *enemy* and `"Meg"` is a neighbor of `"Daisy"`. However, since the shortest path between `"Martin"` and `"Meg"` is `"Martin"` → `"Meg"` (i.e. friend → enemy), `"Meg"` is in the resulting *enemy* set (i.e. `e`).

Graph	 *Graph 1*	 *Graph 2*

Method Call	`Set f = new TreeSet();`
	`Set e = new TreeSet();`
	`friendsAndEnemies("Eve", f, e);`
		`Set f = new TreeSet();`
		`Set e = new TreeSet();`
		`friendsAndEnemies("Martin", f, e);`

Resulting Sets	`f = {"Bob", "Eve"}`
	`e = {"Alice", "Carole", "Dave"}`
		`f = {"Martin", "Viva"}`
		`e = {"Alan", "Daisy", "Meg"}`

You may assume all arguments are valid and that the two `Set`s passed have been initialized (are not `null`) and are empty. For full credit, your method's runtime should be O(V + E). You may not add any additional data fields or public methods to the graph class, but local variables and private methods are allowed.

API Reference for #7:

`Set` methods:

```
boolean add(E e)
boolean addAll(Collection c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection c)
boolean equals(Object o)
boolean isEmpty()
boolean remove(Object key)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
int size()
```

`Graph` fields:

```
protected final Map<V, Map<V, EdgeInfo<E>>> adjacencyMap
protected final Map<V, VertexInfo<V>> vertexInfo
protected final List<E> edgeList
```

`Graph` methods:

```
void addEdge(V v1, V v2, E e)
void addEdge(V v1, V v2, E e, int weight)
void addVertex(V v)
void clearVertexInfo()
boolean containsEdge(V v1, V v2)
boolean containsVertex(V v)
E edge(V v1, V v2)
Collection edges()
int edgeWeight(V v1, V v2)
Collection neighbors(V v)
Collection vertices()
```

`VertexInfo` fields:

```
V v
V previous
boolean visited
int distance
```

8. Disjoint Sets (10 Points)

Trace the sequence of operations below, using a disjoint set with union by size (without path compression). Assume there are initially 14 sets `{1},{2},{3},…,{14}`.

```
union(1, 3);
union(5, 7);
union(8, 9);
union(1, 8);
union(2, 13);
union(1, 10);
union(11, 14);
union(5, 11);
union(2, 6);
union(4, 12);
x = find(9);
y = find(7);
union(x, y);
```

a. Draw the final forest of up-trees that result from the operations above.

b. Draw the new forest of up-trees that results from doing a `find(14)` with path compression on your forest of uptrees from (a).

9. B-Trees (10 Points)

a. Given the following B-Tree with M = 5 and L = 5, draw the B-Tree that results when you insert the value 6 into the tree.

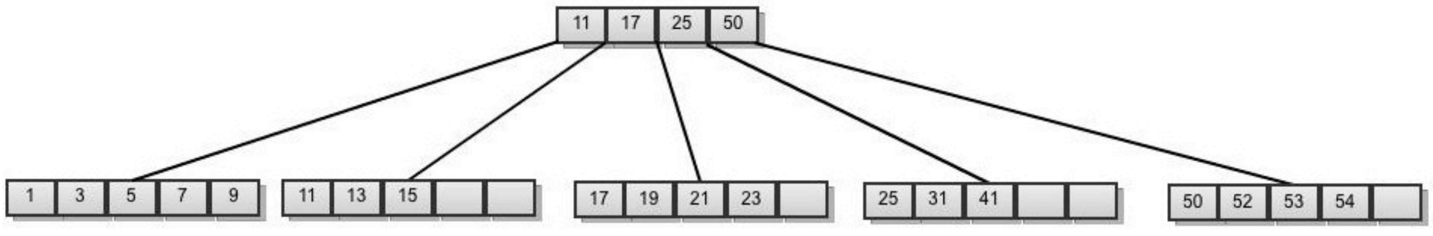


Figure 3 B-Tree in which to Insert 6

b. Explain why a B-Tree might have different values for M and L for internal nodes and leaves, respectively.