This exam has 12 questions, for a total of 100 points.

1. **6 points** Describe what operations are supported by both a Binary Search Tree and a B-tree and which Abstract Data Types they implement. Describe under what conditions one should use a Binary Search Tree versus a B-tree.

   Solution: Both BST and B-tree’s support insert (2 points), search (2 points), and remove operations, with either just keys or key-value pairs. Thus, they can implement the Set (1 point) and Dictionary (aka. Map) ADT’s (1 point).

2. **11 points** Implement the `get` method for the following `HashTable` class that uses separate chaining. You may use the `hash` method (without implementing) to compute the hash of a key. You may assume that each entry in `array` is not `null`.

   ```java
   class Entry<K,V> {
       Entry(K k, V v) { key = k; value = v; }
       K key; V value;
   }
   public class HashTable<K,V> implements Map<K,V> {
       private LinkedList<Entry<K,V>>[] array;
       private int itemCount;
       private int hash(K k) { ... }

       public V get(K key) {
           LinkedList<Entry<K,V>> ls = array[hash(key)]; // 3 points
           for (Entry<K,V> e : ls) // 3 points
               if (e.key.equals(key)) // 3 points
                   return e.value; // 1 point
           return null; // 1 point
       }
   }
   ```

   Solution:

   ```java
   public V get(K key){
       LinkedList<Entry<K,V>> ls = array[hash(key)]; // 3 points
       for (Entry<K,V> e : ls) // 3 points
           if(e.key.equals(key)) // 3 points
               return e.value; // 1 point
       return null; // 1 point
   }
   ```
3. **10 points** Fill in the blanks to complete the `extract_max` method in this `Heap` class.

```java
class Heap {
    int[] data;
    int num_elts;
    Heap(int[] d, int n) { data = d; num_elts = n; }
    void max_heapify(int i) { ... }

    int extract_max() {
        int max = data[___(a)___];
        data[___(b)___] = data[___(c)___];
        num_elts -= 1;
        max_heapify(___(d)___);
        return ___(e)___;
    }
}
```

**Solution: (2 points each)**

- (a) 0
- (b) 0
- (c) `num_elts` - 1
- (d) 0
- (e) `max`
4. **8 points** Identify a depth-first tree rooted at vertex $d$ in the following directed graph by drawing dark lines over the edges in the depth-first tree.

Solution: There are three depth-first trees rooted at vertex $d$, so any of the following are correct solutions. (1 point per edge.)
5. [6 points] What is the worst-case asymptotic time complexity of the following algorithm expressed in terms of the number of vertices \( n \) and the number of edges \( m \) in the graph? Explain the reasoning that leads to your answer. Recall that the priority queue operations are logarithmic time with respect to the number of elements in the queue.

```java
static <V> void shortest_paths(Graph<V> G, V source,
        Map<V,Map<V,Double>> length, Map<V,Integer> position,
        Map<V,Double> distance, Map<V,V> parent, PriorityQueue<V> Q) {
    for (V v : G.vertices()) {
        distance.put(v, Double.MAX_VALUE);
        parent.put(v, v);
    }
    distance.put(source, 0.0);
    Q.push(source);
    while (! Q.empty()) {
        V u = Q.pop();
        for (V v : G.adjacent(u)) {
            if (parent.get(v) == v) Q.push(v);
            Double len = length.get(u).get(v);
            if (distance.get(v) > distance.get(u) + len) {
                Q.increase_key(v);
                distance.put(v, distance.get(u) + len);
                parent.put(v, u);
            }
        }
    }
}
```

**Solution:** The first `for` loop is \( O(n) \) (1 point). We can go through the `while` loop at most \( n \) times (1 point), and `Q.pop()` is \( O(\log n) \), so that’s \( O(n \log n) \) (1 point). The `while` loop also processes each edge once (1 point), and for each edge it can invoke `Q.push(v)` and `Q.increase_key(v)`, both of which are \( O(\log n) \), so that’s \( O(m \log n) \) (1 point). So the time complexity is

\[
O(n) + O(n \log n) + O(m \log n) = O((n + m) \log n) \quad (1 \text{ point})
\]
6. **10 points** Produce a schedule of execution for all of the instructions in the graph below. (Each box represents an instruction.) The edges on the graph indicate dependencies between the instructions. The number on the edge indicates how much time must elapse between when the source instruction is started and when the target instruction may start. If the instruction includes an array access (e.g. \texttt{A[0]}), then it uses the memory unit (MEM) of the computer for 1 unit of time. If the instruction includes arithmetic, such as addition, then it uses the arithmetic logic unit (ALU) of the computer for 1 unit of time. The MEM and ALU can each handle one instruction per unit time, so the computer can execute up to two instructions at the same time.

![Graph of instructions]

A schedule consists of a sequence of time slots with one or more instructions started at each time. The ALU and MEM columns can be used to indicate (with a ✓) when they are busy.

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Instruction</th>
<th>ALU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\texttt{y = A[2];}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\texttt{x = A[0];}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\texttt{y = y + z;}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>\texttt{A[3] = y;}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>\texttt{B[0] = B}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

**Solution:**

Here is one possible solution:

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Instruction</th>
<th>ALU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{y = A[2];}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\texttt{x = A[0];}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\texttt{y = y + z;}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\texttt{y = y + x;} \texttt{A[1] = x;}</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>\texttt{A[3] = y;}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>\texttt{B[0] = B}</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Here is another solution:
<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Instruction</th>
<th>ALU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$x = A[0];$</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$y = A[2];$</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$A[1] = x;$</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$y = y + z;$</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$y = y + x;$</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$B[0] = B;$</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
7. 12 points Implement the connected components algorithm in Java. Recall that a connected component is a set of vertices in which each vertex is reachable from every other vertex in the set. The input is a graph $G$ that supports the following `Graph` interface, and a union-find data structure $ds$ that implements the below `DisjointSets` interface. The output of the algorithm is encoded in $ds$. Every vertex in the same connected component should have the same representative as given by the `find` method.

```java
interface Graph<V> {
    int num_vertices();
    void add_edge(V source, V target);
    Iterable<V> adjacent(V source);
    Iterable<V> vertices();
}

interface DisjointSets<V> {
    void make_set(V x);
    V find(V x);
    V union(V x, V y);
}

static <V> void connected_components(Graph<V> G, DisjointSets<V> ds) {
    for (V v : G.vertices()) // 2 points
        ds.make_set(v); // 2 points
    for (V u : G.vertices()) // 2 points
        for (V v : G.adjacent(u)) // 2 points
            if (ds.find(u) != ds.find(v)) // 2 points
                ds.union(u, v); // 2 points
}
```

Solution:

```java
static <V> void connected_components(Graph<V> G, DisjointSets<V> ds) {
    for (V v : G.vertices()) // 2 points
        ds.make_set(v); // 2 points
    for (V u : G.vertices()) // 2 points
        for (V v : G.adjacent(u)) // 2 points
            if (ds.find(u) != ds.find(v)) // 2 points
                ds.union(u, v); // 2 points
}
```
8. **7 points** Draw the adjacency matrix representation of the following directed graph.

```
0 -> 1 -> 2
3   4   5
```

**Solution:** (1 point per correct edge)

```
   0  1  2  3  4  5
0  ✓  ✓  ✓  ✓  ✓  ✓
1  ✓  ✓  ✓  ✓  ✓  ✓
2  ✓  ✓  ✓  ✓  ✓  ✓
3  ✓  ✓  ✓  ✓  ✓  ✓
4  ✓  ✓  ✓  ✓  ✓  ✓
5  ✓  ✓  ✓  ✓  ✓  ✓
```

9. **10 points** Fill in the blanks to complete the following Java implementation of Breadth-First Search.

```java
static <V> void bfs(Graph<V> G, V start, Map<V, Boolean> visited, Map<V, V> parent) {
    for (V v : G.vertices()) visited.put(v, false);
    Queue<V> Q = new LinkedList<V>();
    Q.add(start); parent.put(start, start);
    visited.put(start, true);
    while (!Q.isEmpty()) {
        V u = Q.remove();
        for (V v : G.adjacent(u))
            if (!visited.get(v)) {
                parent.put(v, u);
                Q.add(v);
                visited.put(v, true);
            }
    }
}
```

**Solution:** (2 points each)

(a) true
(b) ! Q.isEmpty()
(c) G.adjacent(u)
(d) u
(e) v
10. **8 points** Given a hashtable with table size 6, insert the keys 15, 7, 0, 9, 21, 4 into the hashtable using open addressing. Use linear probing with the division method for the underlying hash function. For each key, give the sequence of table positions that led to its final position. Draw the resulting hashtable.

**Solution:**

(6 points, 1 per key)

<table>
<thead>
<tr>
<th>Key</th>
<th>Position sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>3, 4</td>
</tr>
<tr>
<td>21</td>
<td>3, 4, 5</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 0, 1, 2</td>
</tr>
</tbody>
</table>

Hashtable: (2 points)

<table>
<thead>
<tr>
<th>0</th>
<th>→ 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ 7</td>
</tr>
<tr>
<td>2</td>
<td>→ 4</td>
</tr>
<tr>
<td>3</td>
<td>→ 15</td>
</tr>
<tr>
<td>4</td>
<td>→ 9</td>
</tr>
<tr>
<td>5</td>
<td>→ 21</td>
</tr>
</tbody>
</table>
11. **9 points** Given the following B-tree with minimum degree \( t = 2 \), insert an element with keys \( W \) and \( M \). The keys are ordered alphabetically. Show your work, including the result of each split operation.

\[
\begin{array}{c}
F, Q \\
C \quad H, K, L \quad S, T, V
\end{array}
\]

**Solution:**
Recall that every node has at most 3 keys. Every node other than the root must have 1 or more keys. The B-tree needs to obey the ordering constraints for the keys (like a Binary Search Tree).

To insert \( W \), we first split the node \([S, T, V]\), lifting \( T \), then place \( W \) next to \( V \).

\[
\begin{array}{c}
F, Q, T \\
C \quad H, K, L \quad S \quad V, W
\end{array}
\]  
(3 points)

To insert \( M \), we must first split the root \([F, Q, T]\), lifting \( Q \).

\[
\begin{array}{c}
Q \\
F \quad T
\end{array}
\]

\[
\begin{array}{c}
C \quad H, K, L \quad S \quad V, W
\end{array}
\]
(3 points)

We then split \([H, K, L]\), lifting \( K \), and place \( M \) next to \( L \).

\[
\begin{array}{c}
Q \\
F, K \quad T
\end{array}
\]

\[
\begin{array}{c}
C \quad H \quad L, M \quad S \quad V, W
\end{array}
\]
(3 points)

12. **3 points** What advice would you give a student taking C343 Data Structures next year regarding what steps to take when implementing a data structure or algorithm.

**Solution:** Open ended.