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

1. **8 points** Given a B-tree that contains $n$ items and has minimum degree $t$, what is the time complexity of the below search method? (Recall that the linear_search(A, k) function finds the position of key $k$ inside the array $A$ with time complexity $O(m)$, where $m$ is the length of $A$.)

```python
class BTreeNode:
    def __init__(self, keys=[], children=[], leaf=True):
        self.keys = keys
        self.children = children
        self.leaf = leaf

    def search(self, key):
        i = linear_search(self.keys, key)
        if i < len(self.keys) and key == self.keys[i]:
            return (self, i)
        elif self.is_leaf:
            return None
        else:
            return self.children[i].search(key)
```

**Solution:** The depth of the recursion for `search` is $O(h)$, where $h$ is the height of the tree (2 points). The B-tree is balanced and has a branching factor of at least $t$, so the height is $O(\log_t n)$ (2 points). Inside each invocation of `search`, there is a call to `linear_search`, whose time is $O(t)$ (2 points). Multiplying this by the depth of the recursion gives the time complexity of B-tree `search` as $O(t \log_t n)$ (2 points).
2. **6 points** Implement the `heap_decrease_key` method of the following `Heap` class, which implements a min-heap. The parameter `i` of `heap_decrease_key` indicates the position in the heap’s array of the item whose key has decreased. The `heap_decrease_key` method should rearrange the heap so that it once again satisfies the heap property. (Assume that all the other methods of the `Heap` have already been implemented.) You may also use any of the helper functions on the right.

```python
class Heap:
    def __init__(self, array, less = less_than):
        self.array = array
        self.less = less
        self.build_min_heap()

    def heap_decrease_key(self, i):
        while i > 0 and self.less(self.array[i], self.array[parent(i)]):
            swap(self.array, i, parent(i))
            i = parent(i)

def less_than(x, y):
    return x < y

def swap(A, i, j):

def left(i):
    return 2 * i + 1

def right(i):
    return 2 * (i + 1)

def parent(i):
    return (i - 1) / 2

Solution:

```python
def heap_decrease_key(self, i):
    while i > 0 and self.less(self.array[i], self.array[parent(i)]):
        swap(self.array, i, parent(i))
        i = parent(i)
```
3. [6 points] Draw the trie that would result from inserting the following key-value pairs:

"hello" : 9, "its" : 3, "if" : 1, "these" : 1, "to" : 14, "they" : 1, "heal" : 1

Solution:

```
          o
         /|
        /  |
       h   i
      /|
     /  |
    e   t
   /|
  f:1|
  /|
 h   o:14
/|
l   a
 /|
 l   s:3
 /|
 l   e
 /|
 y:1|
 /|
n   e:1
```
4. **10 points** Fill in the missing blanks in the following implementation of a hashtable.

```python
def division_method(k, m):
    return ____(a)____

class Hashtable(dict):
    def __init__(self, size=101, hash_function=division_method):
        self.table = [___(b)___ for i in range(0, size)]
        self.hash_function = hash_function
        self.num_items = 0

    def __getitem__(self, key):
        h = self.hash_function(key, ___(c)___)
        for (k,v) in self.table[___(d)___]:
            if ___(e)___:
                return v
        raise KeyError(key)
```

**Solution:** (2 points each)

- (a) hash(k) % m
- (b) []
- (c) len(self.table)
- (d) h
- (e) key == k
5. [10 points] Fill in the blanks to complete the following implementation of an adjacency-list representation of a directed graph.

```python
class Edge:
    def __init__(self, src, tgt):
        self.source = src; self.target = tgt

class DirectedAdjList:
    def __init__(self, num_vertices):
        self.array = [[], for i in range(0, num_vertices)]

    def add_edge(self, u, v):
        ___(a)___

    def out_edges(self, u):
        for v in ___(b)___:
            yield Edge(u, v)

    def vertices(self):
        for u in ___(c)___:
            yield u

    def edges(self):
        for u in ___(d)___:
            for e in ___(e)___:
                yield e
```

Solution: (2 points each)

(a) self.array[u].append(v)
(b) self.array[u]
(c) len(self.array)
(d) self.vertices()
(e) self.out_edges(u)
6. **10 points** What operations are provided by the Disjoint Sets data structure? Describe a problem that you can solve using Disjoint Sets?

**Solution:** A Disjoint Sets data structure ds provides:

- `ds.make_set(x)`: puts x in a set by itself. (2 points)
- `ds.find(x)`: returns the representative for x. (2 points)
- `ds.union(x,y)`: combines the two sets containing x and y into a single set. (2 points)

The Disjoint Sets data structure can be used to

- compute connected components in a graph, or
- find the minimum spanning tree of a graph (Kruskal’s algorithm), or
- unify two terms (i.e., solve equations over trees), or
- keep track of which people are related to other people via marriage or parent/child.

(4 points for one example problem)
7. [8 points] Apply Knuth’s algorithm to topologically sort the following graph. Show your work by writing down the contents of the queue at each iteration of the algorithm. At the end, write down the topological ordering of the vertices.

Solution: Contents of the queue at each iteration:

```
g | c, k | k, d | d, j | j, h | h, f | f, l | l, b | b | a | e | i   (4 points)
```

A topological ordering is

```
g, c, k, d, j, h, f, l, b, a, e, i   (4 points)
```

There are many other solutions.
8. **10 points** Five new islands have appeared in the Florida Keys due to a mysterious drop in ocean levels and the government wants to build bridges to connect them so that each island can be reached from any other one via one or more bridges. The cost of constructing a bridge is proportional to its length. The distances in miles between pairs of islands are given in the following table, in which the islands are numbered 1 through 5. State which bridges to build so that the total construction cost is minimized. Additionally, state the total cost.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>24</td>
<td>21</td>
<td>34</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>26</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>27</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Solution:** This solution applies Kruskal’s algorithm to compute the minimum spanning tree (MST). (3 points)

The first step is to order the edges by weight: (2 points)

\[
(11)3 - 5, (16)4 - 5, (17)2 - 4, (21)1 - 3, (22)2 - 5, (24)1 - 2, (26)2 - 3, \\
(27)3 - 4, (28)1 - 5, (34)1 - 4
\]

Next we process each edge, deciding to make it part of the MST or not, depending on whether the endpoints of each edge are in different disjoint sets.

\[
3 - 5(yes), 4 - 5(yes), 2 - 4(yes), 1 - 3(yes), 2 - 5(no), 1 - 2(no), 2 - 3(no), \\
3 - 4(no), 1 - 5(no), 1 - 4(no)
\]

So we build the bridges as indicated by the “yes” answers above. (3 points)
The total cost is 65. (2 points)
Implement depth-first search in Python. The function, shown below, should have four parameters: a graph, source vertex u, a parent array, and a done array. You may assume that the graph is an instance of the DirectedAdjList of question 5, so vertices are represented by integers. You may assume that the parent array initially maps each vertex to None and the done array initially maps each vertex to False. You may use the done array to keep track of which vertices have already been visited. The result of your depth-first search should be a depth-first tree that spans all of the vertices reachable from u and the tree should be encoded in the parent array. (You do not need to “restart” the DFS to visit every vertex in the graph.)

```python
def depth_first_search(graph, u, parent, done):
    done[u] = True  # (2 points)
    for e in g.out_edges(u):  # (2 points)
        v = e.target
        if not done[v]:  # (2 points)
            parent[v] = u  # (2 points)
            depth_first_search(g, v, parent, done)  # (2 points)
```

**Solution:**

```python
def depth_first_search(graph, u, parent, done):
    done[u] = True  # (2 points)
    for e in g.out_edges(u):  # (2 points)
        v = e.target
        if not done[v]:  # (2 points)
            parent[v] = u  # (2 points)
            depth_first_search(g, v, parent, done)  # (2 points)
```
10. [8 points] Identify a breadth-first tree rooted at vertex $d$ in the following directed graph by drawing dark lines over the edges in the breadth-first tree.

```
  a ------ b <------ c
    |         |         |
    |         v         |
    d ------ e ------ f
    |
    v
  g ------ h <------ i
```

**Solution:** While in general there can be many BFS trees, for source $d$ in this graph, there is only one solution: (1 point per correct edge.)

```
  a ------ b <------ c
    |         |         |
    |         v         |
    d ------ e ------ f
    |
    v
  g ------ h <------ i
```
11. **6 points** Given the following B-tree with minimum degree \( t = 2 \), insert an element with key 4. Show all of your work.

![B-tree diagram](image)

**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).

The first step is to split the root node: (2 points)

![Root node split](image)

The second step is to split the leaf node with the keys 3, 5, 6. The key 5 moves up to its parent. (2 points)

![Leaf node split](image)

The final step is to insert 4 into the leaf node with key 3. (2 points)

![Leaf node insertion](image)
12. [8 points] Apply Dijkstra’s shortest paths algorithm to the following graph, starting at vertex $i$. Show your work by recording what the priority queue looks like and which vertex gets popped at each iteration of the algorithm. Indicate the resulting shortest paths tree by making the tree edges into solid lines. Next to each vertex, write the distance of the shortest path from $i$ to that vertex.

\[
\begin{array}{c}
\text{a} \xrightarrow{2} \text{b} \xrightarrow{0.5} \text{c} \\
2 & 1 & 2 \\
\text{d} \xrightarrow{1.5} \text{e} \xrightarrow{4} \text{f} \\
3 & 4 & 1 \\
\text{g} \xrightarrow{2} \text{h} \xrightarrow{2} \text{i}
\end{array}
\]

**Solution:** The priority queue at each step: (2 points)

\[
\begin{align*}
\{i : 0\} & \quad \text{pop } i \\
\{f : 1, h : 2\} & \quad \text{pop } f \\
\{h : 2, c : 3, e : 5\} & \quad \text{pop } h \\
\{c : 3, g : 4, e : 5\} & \quad \text{pop } c \\
\{b : 3.5, g : 4, e : 5\} & \quad \text{pop } b \\
\{g : 4, e : 4.5, a : 5.5\} & \quad \text{pop } g \\
\{e : 4.5, a : 5.5, d : 7\} & \quad \text{pop } e \\
\{a : 5.5, d : 6\} & \quad \text{pop } a \\
\{d : 6\} & \quad \text{pop } d
\end{align*}
\]

The shortest paths tree and shortest distances from $i$ to all the other vertices:

\[
\begin{array}{c}
\text{a}^{5.5} \xrightarrow{2} \text{b}^{3.5} \xrightarrow{0.5} \text{c}^{3} \\
2 & 1 & 2 \\
\text{d}^{6} \xrightarrow{1.5} \text{e}^{1.5} \xrightarrow{4} \text{f}^{1} \\
3 & 4 & 1 \\
\text{g}^{4} \xrightarrow{2} \text{h}^{2} \xrightarrow{2} \text{i}^{0}
\end{array}
\]

(3 points for the correct tree) (3 points for the correct distances)