9. A B+ Tree is a multi-way tree (typically in the order of 100s children per node) used primarily as a file-index structure to allow fast search (as well as insertions and deletions) for a target key on disk. Two types of pages (B+ tree "nodes") exist:
- Data pages - which always appear as leaves on the same level of a B+ tree (usually a doubly-linked list too)
- Index pages - the root and other interior nodes above the data page leaves. Index nodes contain some minimum and maximum number of keys and pointers bases on the B+ tree's branching factor (b) and fill factor. A 50% fill factor would be the minimum for any B+ tree. All index pages must have $\lfloor b/2 \rfloor \leq \# \text{ child} \leq b$, except the root which must have at least two children.

Consider an B+ tree example with $b = 5$.

a) How would you find 88?

b) The insert algorithm for a B+ tree is summarized by the below table. Where would you insert 50, 100, 105, 110, 180, 200, 210?

<table>
<thead>
<tr>
<th>Situation</th>
<th>Data Page Full?</th>
<th>Parent Index Page Full?</th>
<th>insertion Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td></td>
<td>Place record in sorted position in the appropriate data page.</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td></td>
<td>1. Split data page with records $&lt;$ middle key going in left data page and records $\geq$ middle key going in right data page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Place middle key in index page in sorted order with the pointer immediately to its left pointing to the left data page and the pointer immediately to its right pointing to the right data page.</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>1. Split data page with records $&lt;$ middle key going in left data page and records $\geq$ middle key going in right data page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Adding middle key to parent index page causes it to split with keys $&lt;$ middle key going into the left index page, keys $&gt;$ middle key going in right index page, and the middle key inserted into the next higher level index page. If the next higher index page is full continue to splitting index pages up the B+ tree as necessary.</td>
</tr>
</tbody>
</table>
c) For a B+ tree with a branch factor 201, what would be the worst case height of the tree if the number of keys was 1,000,000?

\[ \log_{201}(1,000,000) \]

\[ \approx 7 \text{ or 8 levels} \]

10. The deletion algorithm for a B+ tree is summarized by the below table.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Data Page Below Fill Factor?</th>
<th>Parent Index Page Below Fill Factor?</th>
<th>Deletion Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Delete record from the data page. Shifting records with larger keys to left to fill in the hole. If the deleted key appears in the index page, use the next key to replace it.</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>1. Combine data page and its sibling. Change the index page to reflect the change.</td>
</tr>
</tbody>
</table>
| Yes                 | Yes                          |                                      | 1. Combine data page and its sibling.  
2. Adjusting the index page to reflect the change causes it to drop below the fill factor, so combine the index page with its sibling.  
3. Continue combining the next higher level index pages until you reach an index page with the correct fill factor or you reach the root index page. |

Consider an B+ tree example with b = 5 and 50% fill factor. Delete 89, 65, and 88. What is the resulting B+ tree?
1. Consider the following directed graph (diagraph) \( G = (V, E) \):

![Graph Diagram]

a) What is the set of vertices? \( V = \{v_0, v_1, ..., v_5\} \)

b) An edge can be represented by a tuple (from vertex, to vertex [weight]). What is the set of edges?

\[ E = \{ (v_0, v_1, 1), (v_0, v_2, 3), (v_0, v_4, 5), \ldots \} \]

c) A path is a sequence of vertices that are connected by edges. In the graph \( G \) above, list two different paths from \( v_0 \) to \( v_3 \): \( v_0, v_1, v_2, v_3 \) \( \cup \) \( v_0, v_1, v_3, v_4, v_3 \)

d) A cycle in a directed graph is a path that starts and ends at the same vertex. Find a cycle in the above graph.

\( v_0, v_1, v_0 \) \( \cup \) \( v_0, v_3, v_4, v_0 \)

2. Like most data structures, a graph can be represented using an array, or as a linked list of nodes. The array representation is a two-dimensional array (called an adjacency matrix) whose elements contain information about the edges and the vertices corresponding to the indices. (Python could use a list-of-lists)

a) Complete the following adjacency matrix for the above graph. (Here a missing edge is represented by \( \infty \).)

<table>
<thead>
<tr>
<th></th>
<th>( v_0 )</th>
<th>( v_1 )</th>
<th>( v_2 )</th>
<th>( v_3 )</th>
<th>( v_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_0 )</td>
<td>0</td>
<td>1</td>
<td>( \infty )</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>( v_1 )</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>0</td>
<td>4</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>2</td>
<td>( \infty )</td>
<td>3</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>3</td>
<td>( \infty )</td>
<td>3</td>
<td>( \infty )</td>
<td>0</td>
</tr>
</tbody>
</table>

b) The linked representation maintains a linked-list (or Python dictionary) of vertices with each vertex maintaining a linked list of other vertices that it connects to. Complete the adjacency list representation below:

![Adjacency List Diagram]
3. Graphs can be used to solve many problems by modeling the problem as a graph and using "known" graph algorithm(s). For example, consider the word-ladder puzzle where you transform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → PALL → PALE → SALE → SAGE.

We can use a graph algorithm to solve this problem by constructing a graph such that
- a word represents a vertex
- an edge represents that connects two words that differ in one letter
- a word ladder transformation from one word to another represents a path

4. For the words listed below, draw the graph of question 3

![Graph Image]

a) List a different transformation from FOOL to SAGE

```
fool → cool → pool → poll → pole → pole → page → sage
```

b) If we wanted to find the shortest transformation from FOOL to SAGE, what does that represent in the graph?

```
Shortest path
```

c) There are two general approaches for traversing a graph from some starting vertex s:

- Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from s, before finding all vertices a distance 2 from s, etc.
- Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a “dead end,” we backtrack to the deepest vertex that allows us to try a different path.

Which of these traversals would be helpful for finding the shortest solution to the word-ladder puzzle?

```
BFS
```
1. There are two general approaches for traversing a graph from some starting vertex $s$:

- **Depth First Search (DFS)** where you explore as deeply into the graph as possible. If you reach a “dead end,” we backtrack to the deepest vertex that allows us to try a different path.

- **Breadth First Search (BFS)** where you find all vertices a distance 1 (directly connected) from $s$, before finding all vertices a distance 2 from $s$, etc.

What data structure would be helpful in each type of search? Why?

a) **Breadth First Search (BFS):**

b) **Depth First Search (DFS):**

2. On the next page is the textbook’s edge, vertex, and graph implementations.

a) How does this graph implementation maintain its set of vertices?

b) How does this graph implementation maintain its set of edges?

3. Assuming a graph $G$ containing the word-ladder graph from lecture 25, on the diagram trace the **BFS** algorithm by showing the value of each vertex’s `color`, `predecessor`, and `distance` attributes?
Data Structures (CS 1520)  

Lecture 26

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        self.numVertices += 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vertList

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(
            self.vertList[t], cost)

def getVertices(self):
    return self.vertList.keys()

def __iter__(self):
    return iter(self.vertList.values())

from graph import Graph
from vertex import Vertex
from linked_queue import LinkedQueue

def bfs(g, start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = LinkedQueue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance()+1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
        currentVert.setColor('black')

from vertex import Vertex
class Vertex:
    def __init__(self, key, color = 'white',
                 dist = 0, pred = None):
        self.key = key
        self.connectedTo = {}
        self.color = color
        self.predecessor = pred
        self.distance = dist
        self.discovery = 0
        self.finish = 0

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo:
        ' + str([x.id for x in self.connectedTo])

    def getId(self):
        return self.id

    def getWeight(self, nbr):
        return self.connectedTo[nbr]

    def getColor(self):
        return self.color

    def setColor(self, newColor):
        self.color = newColor

    def getPred(self):
        return self.predecessor

    def setPred(self, newPred):
        self.predecessor = newPred

    def getDiscovery(self):
        return self.discovery

    def setDiscovery(self, newDiscovery):
        self.discovery = newDiscovery

    def getFinish(self):
        return self.finish

    def setFinish(self, newFinish):
        self.finish = newFinish

    def getDistance(self):
        return self.distance

    def setDistance(self, newDistance):
        self.distance = newDistance

""" File: graph.py """
from vertex import Vertex

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

def addVertex(self, key):
    self.numVertices += 1
    newVertex = Vertex(key)
    self.vertList[key] = newVertex
    return newVertex

def getVertex(self, n):
    if n in self.vertList:
        return self.vertList[n]
    else:
        return None

def __contains__(self, n):
    return n in self.vertList

def addEdge(self, f, t, cost=0):
    if f not in self.vertList:
        nv = self.addVertex(f)
    if t not in self.vertList:
        nv = self.addVertex(t)
    self.vertList[f].addNeighbor(t, cost)

def getVertices(self):
    return self.vertList.keys()

def __iter__(self):
    return iter(self.vertList.values())

""" File: graph_algorithms.py """
from vertex import Vertex
from linked_queue import LinkedQueue

def bfs(g, start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = LinkedQueue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance()+1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
        currentVert.setColor('black')