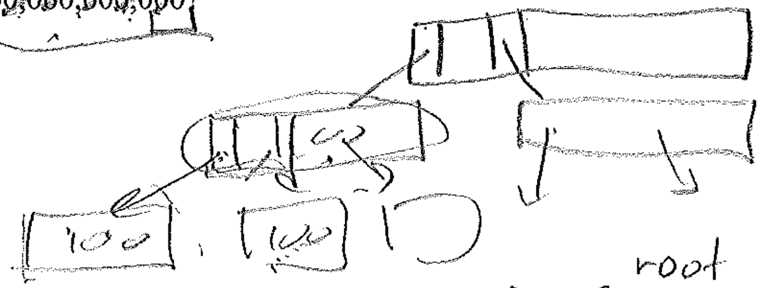
c) For a B+ tree with a branch factor 201, what would be the worst case height of the tree if the number of keys was 1,000,000,000,000?



1

2x100

2x100x100

⋮

pick up x100 multiplier at each level

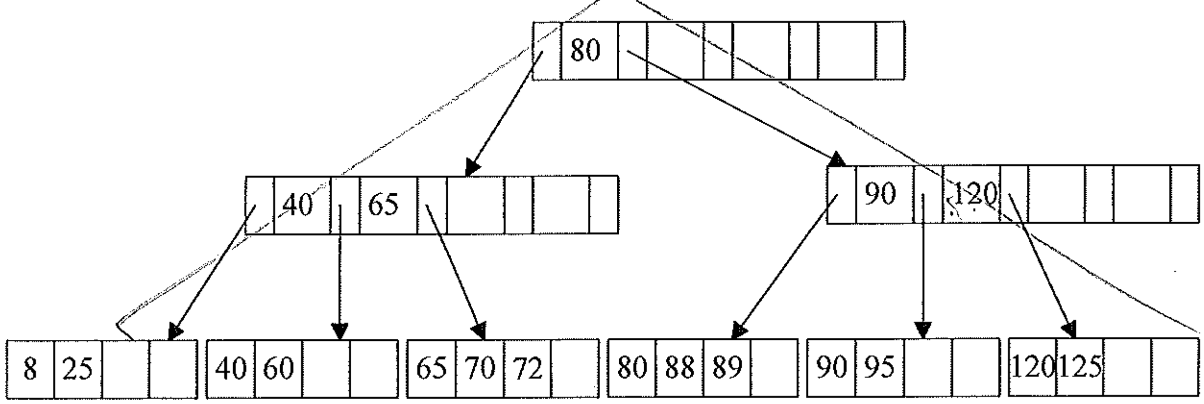
root

$\rightsquigarrow$ height $= O(\log_{100} n) + 1$

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

Situation		
Data Page Below Fill Factor?	Parent Index Page Below Fill Factor?	deletion Algorithm
No	No	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.
Yes	No	1. Combine data page and its sibling. Change the index page to reflect the change.
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.

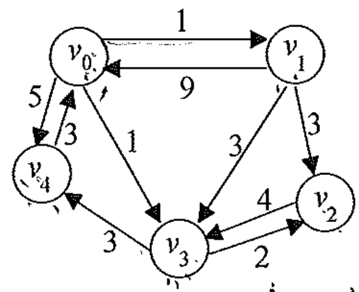
Consider an B+ tree example with b = 5 and 50% fill factor. Delete 89, 65, and 88. What is the resulting B+ tree?



Delete from B+ tree not on Final Exam.

Data Structures (CS 1520) Lecture 25 Name:________________

1. Consider the following directed graph (diagraph) $G = (V, E)$:



a) What is the set of vertices? $V =$ $\{v_0, v_1 \ldots, v_4\}$ $|V| = 5$

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

$E = \{(v_0, v_1, 1), (v_0, v_3, 1), (v_0, v_4, 5), \ldots, (v_4, v_0, 3)\}$

c) A path is a sequence of vertices that are connected by edges. In the graph G above, list two different pathes from v_0 to v_3. v_0, v_3 or v_0, v_1, v_2, v_3 etc.

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_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.

a) The array representation is called an *adjacency matrix* which consists of a two-dimensional array (matrix) whose elements contain information about the edges and the vertices corresponding to the indices.

Complete the following adjacency matrix for the above graph.

O(1) check edge wt.

Storage: $O(|V|^2)$

(to vertex)

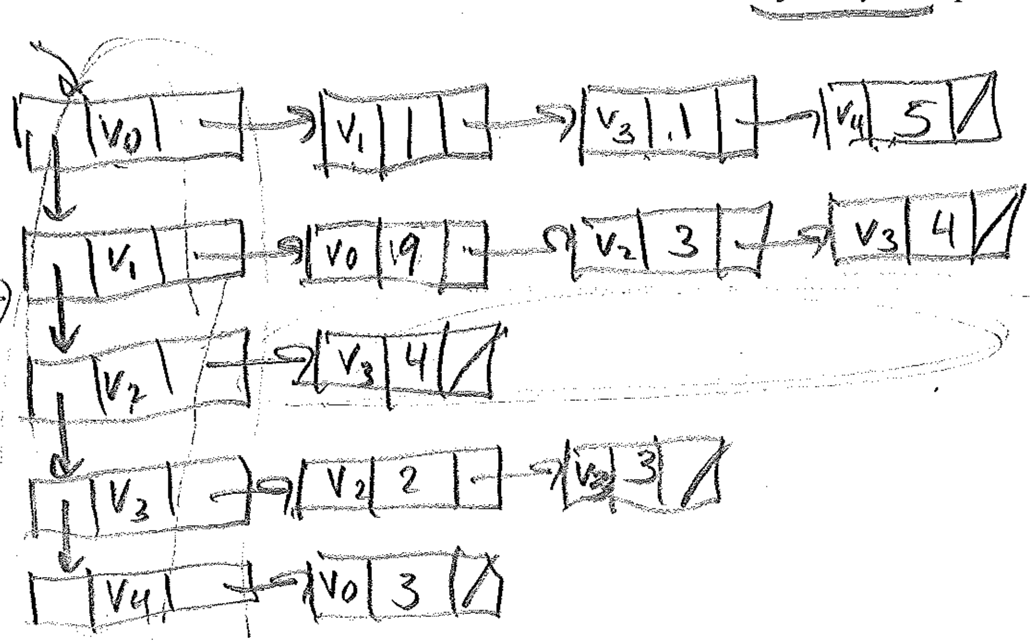
(from vertex)	v_0	v_1	v_2	v_3	v_4
v_0		1	∞	1	5
v_1	9		3	3	
v_2				4	
v_3			2	3	
v_4	3				

v_2 → v_0

3. The linked representation maintains a array/Python list (or Python dictionary) of vertices with each vertex maintaining a linked list of other vertices that it connects to. Draw the adjacency list representation below:

O(|V|) better

storage: O(|V|+|E|)



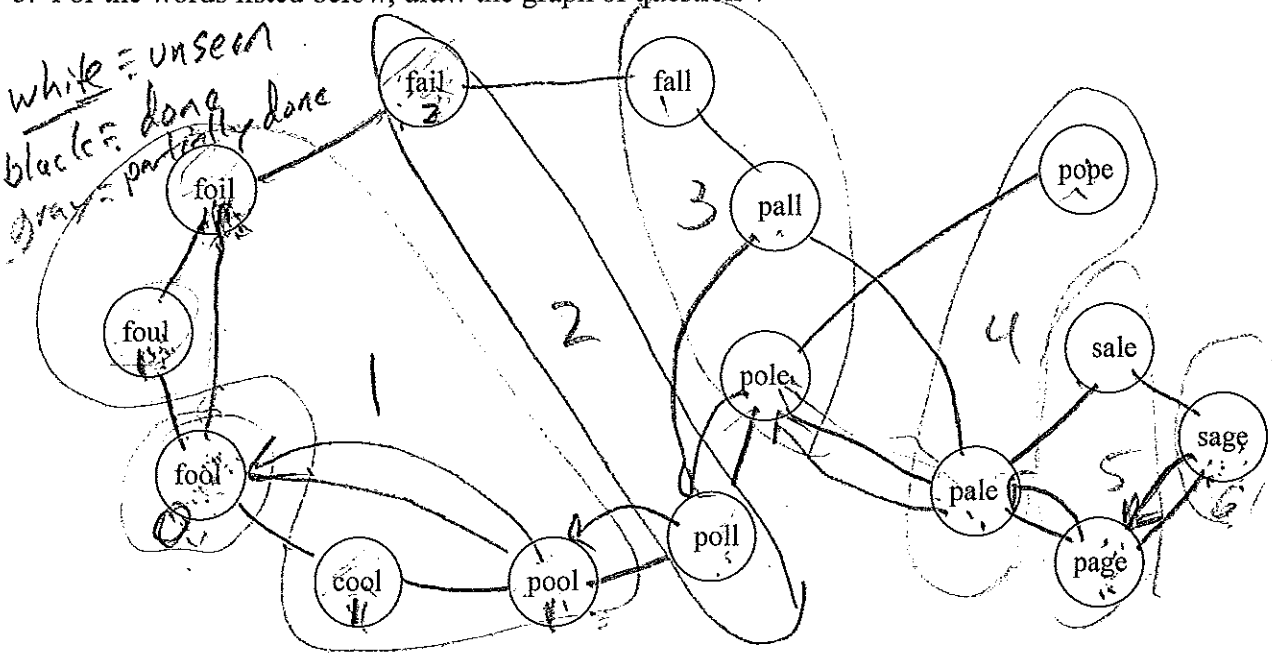
4. Graphs can be used to solve many problems by modeling the problem as a graph and using "known" graph algorithm(s). For example, consider ~~the word-ladder puzzle~~ where you tranform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → 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? connect two words/vertices that differ by a single letter
- a word ladder transformation from one word to another represents?

  path from starting word to ending word

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



a) List a different transformation from FOOL to SAGE

fool → cool → pool → poll → pole → pale → sale → sage

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

shortest path from starting word (Fool) to (sage)

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 - when find sage it will be by shortest path.

Data Structures (CS 1520) Lecture 26 Name:________________

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):

FIFO queue

(see attached)

b) Depth First Search (DFS):

DFS can use a similar algorithm to BFS, but instead of a queue use a stack, or use the run-time stack and recursion

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?

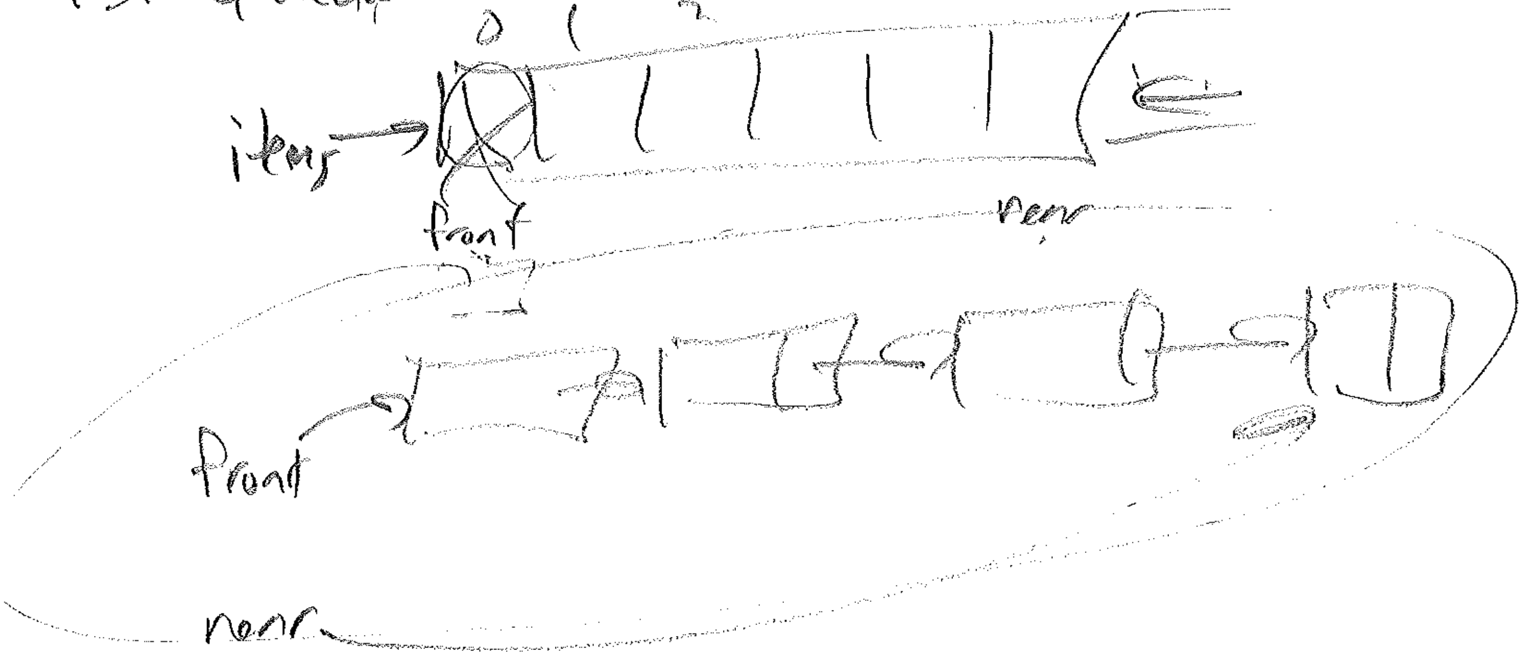
A dictionary (Graph class self.vertList), so a vertex can be found in O(1) using its id/label.

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

Each Vertex object uses a dictionary (Vertice class self.connectedTo), so an edge can be found in O(1) using its connecting vertex id/label.

3. Assuming a graph g containing the word-ladder graph from lecture 26, on the diagram trace the `bfs` algorithm by showing the value of each vertex's `color`, `predecessor`, and `distance` attributes?

FIFO Queue



$O(1)$

BFS alg.

emptyQ
enqueue start vertex
while Q is not empty do
    dequeue the current vertex
    for every nextVertex connected to currentVertex
        if nextVertex still 'white' then
            update distance ← 1+currentVertex distance
            mark nextVertex as gray
            enqueue nextVertex

```
""" File:  vertex.py """
class Vertex:
    def __init__(self, key, color = 'white',
                 dist = 0, pred = None):
        self.id = 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 getConnections(self):
        return self.connectedTo.keys()

    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 = 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())
```

```
""" File:  graph_algorithms.py """

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')
```