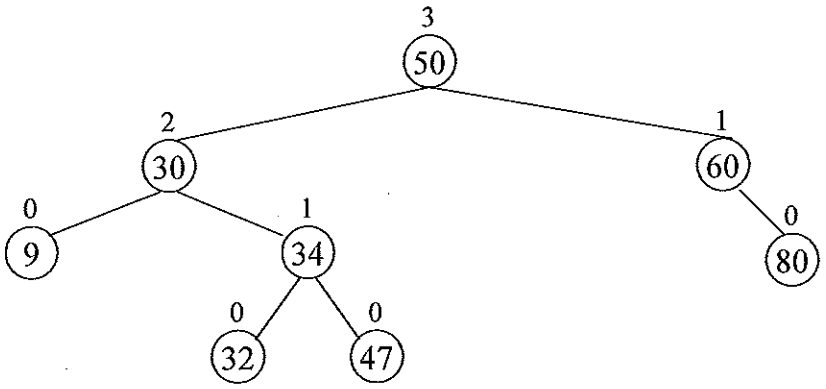
1. An *AVL Tree* is a special type of Binary Search Tree (BST) that it is *height balanced.* By height balanced I mean that the height of every node's left and right subtrees differ by at most one. This is enough to guarantee that a AVL tree with n nodes has a height no worst than $O(1.44 \log_2 n)$. Therefore, insertions, deletions, and search are worst case $O(\log_2 n)$. An example of an AVL tree with integer keys is shown below. The height of each node is shown.



Each AVL-tree node usually stores a *balance factor* in addition to its key and payload. The balance factor keeps track of the relative height difference between its left and right subtrees, i.e., height(left subtree) - height(right subtree).

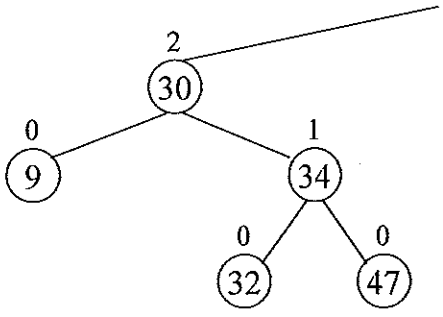
a) Label each node in the above AVL tree with one of the following *balance factors*:

- 0 if its left and right subtrees are the same height
- 1 if its left subtree is one taller than its right subtree
- -1 if its right subtree is one taller than its left subtree

b) We start a `put` operation by adding the new item into the AVL as a leaf just like we did for Binary Search Trees (BSTs). Add the key 90 to the above tree.

c) Identify the node "closest up the tree" from the inserted node (90) that no longer satisfies the height-balanced property of an AVL tree. This node is called the *pivot node.* Label the pivot node above.

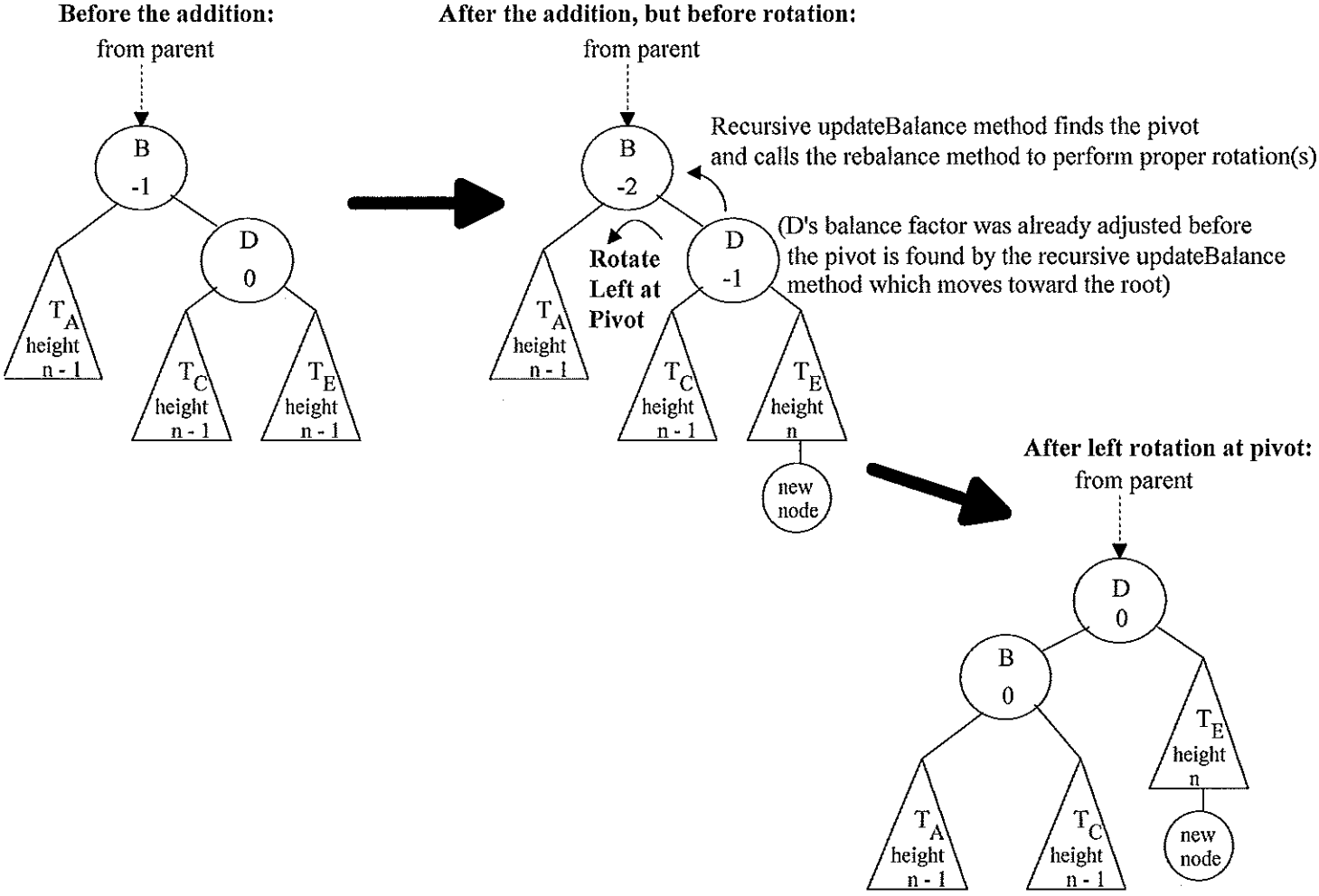
d) Consider the subtree whose root is the pivot node. How could we rearrange this subtree to restore the AVL height balanced property? (Draw the rearranged tree below)



2. Typically, the addition of a new key into an AVL requires the following steps:

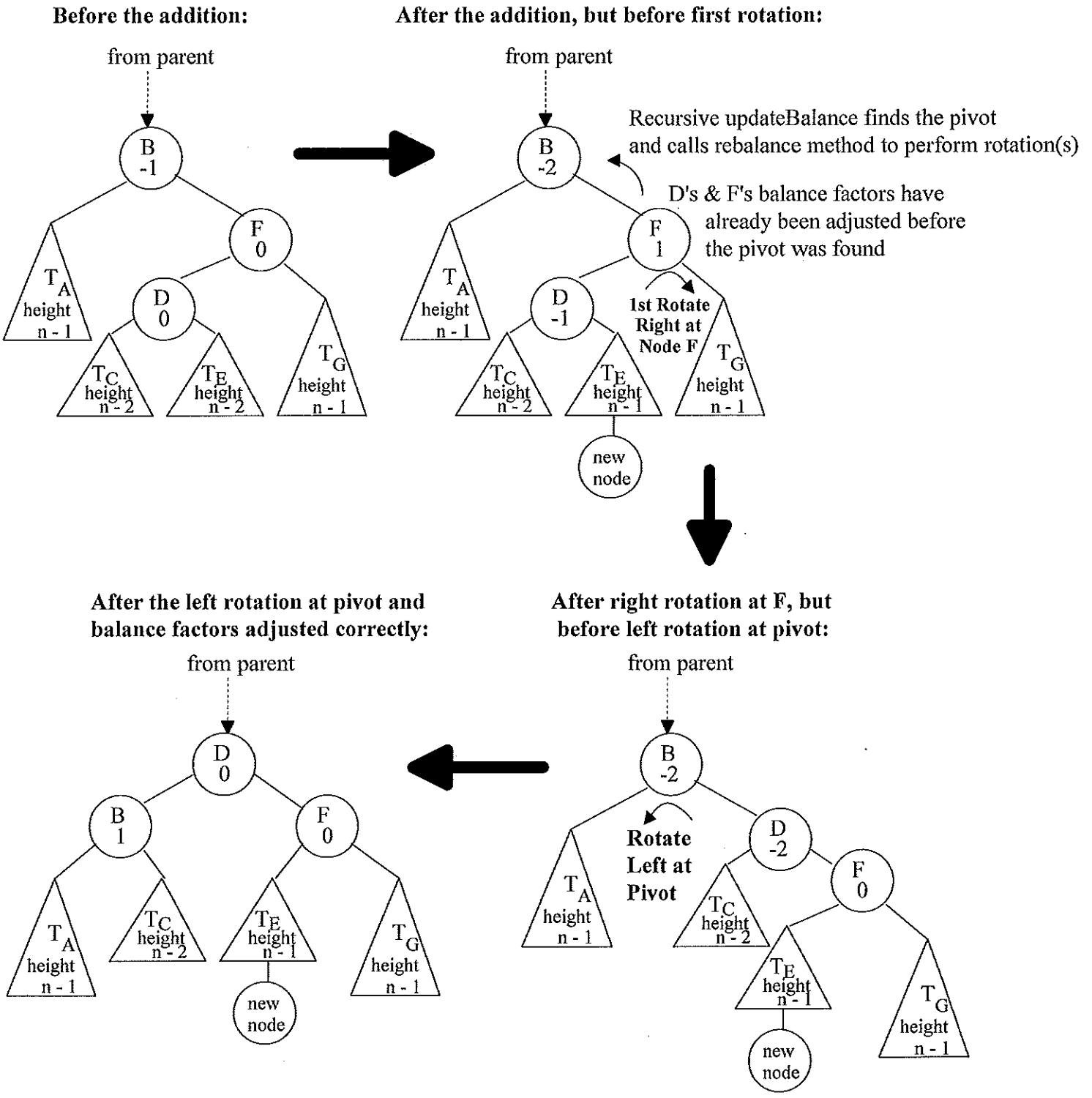
- compare the new key with the current tree node's key (as we did in the `_put` function called by the `put` method in the BST) to determine whether to recursively add the new key into the left or right subtree
- add the new key as a leaf as the base case(s) to the recursion
- recursively (`updateBalance` method) adjust the balance factors of the nodes on the search path from the new node back up toward the root of the tree. If we encounter a pivot node (as in question (c) above) we perform one or two "rotations" to restore the AVL tree's height-balanced property.

For example, consider the previous example of adding 90 to the AVL tree. Before the addition, the pivot node (60) was already -1 ("tall right" - right subtree had a height one greater than its left subtree). After inserting 90, the pivot's right subtree had a height 2 more than its left subtree (balance factor -2) which violates the AVL tree's height-balance property. This problem is handled with a *left rotation* about the pivot as shown in the following generalized diagram:



a) Assuming the same initial AVL tree (upper, left-hand of above diagram) if the new node would have increased the height of T_C (instead of T_E), would a left rotation about the node B have rebalanced the AVL tree?

b) Before the addition, if the pivot node was already -1 (tall right) and if the new node is inserted into the left subtree of the pivot node's right child, then we must do two rotations to restore the AVL-tree's height-balance property.



b) Suppose that the new node was added in T_C instead of T_E, then the same two rotations would restore the AVL-tree's height-balance property. However, what should the balance factors of nodes B, D, and F be after the rotations?

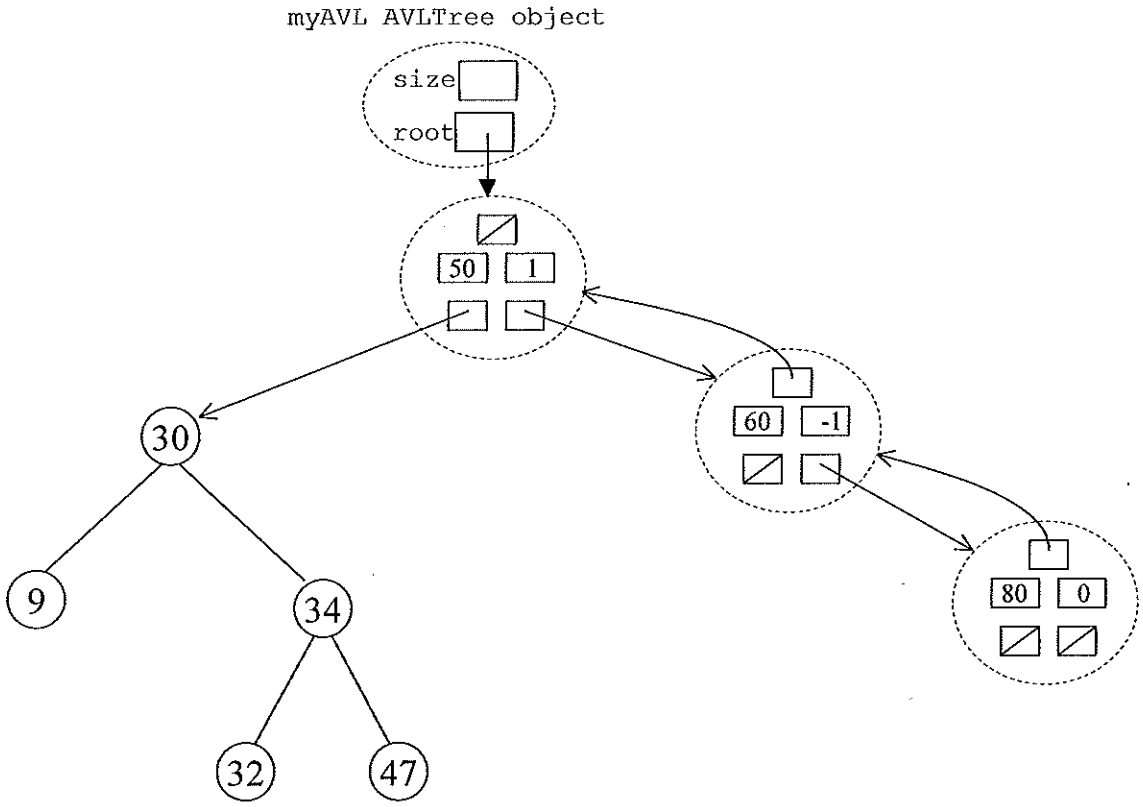
Consider the AVLTreeNode class that inherits and extends the TreeNode class to include balance factors.

```
from tree_node import TreeNode

class AVLTreeNode(TreeNode):
    def __init__(self,key,val,left=None,right=None,parent=None, balanceFactor=0):
        TreeNode.__init__(self,key,val,left,right,parent)
        self.balanceFactor = balanceFactor
```

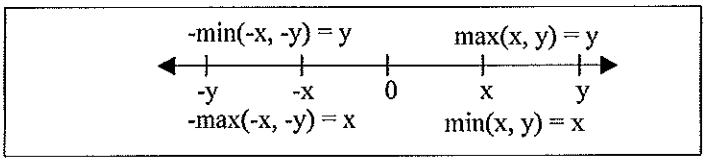
Now let's consider the partial AVLTree class code that inherits from the BinarySearchTree class:

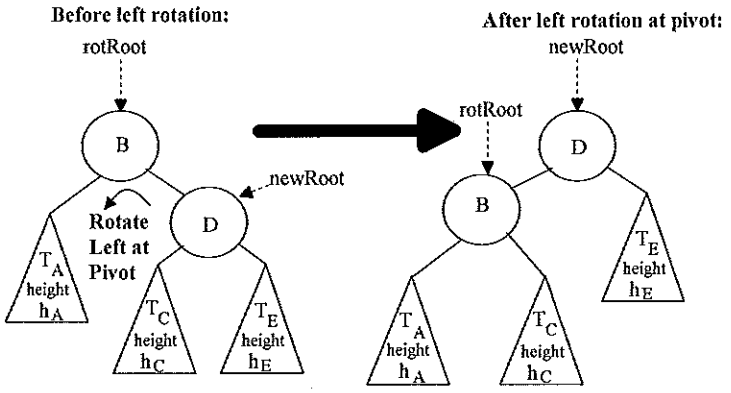
```
from avl_tree_node import AVLTreeNode
from binary_search_tree import BinarySearchTree

class AVLTree(BinarySearchTree):

    def put(self,key,val):
        if self.root:
            self._put(key,val,self.root)
        else:
            self.root = AVLTreeNode(key,val)
        self.size = self.size + 1

    def _put(self,key,val,currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
            else:
                currentNode.leftChild = AVLTreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.leftChild)
        elif key > currentNode.key:
            if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
            else:
                currentNode.rightChild = AVLTreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.rightChild)
        else:
            currentNode.payload = val
            self._size -= 1

    def updateBalance(self,node):
        if node.balanceFactor > 1 or node.balanceFactor < -1:
            self.rebalance(node)
            return
        if node.parent != None:
            if node.isLeftChild():
                node.parent.balanceFactor += 1
            elif node.isRightChild():
                node.parent.balanceFactor -= 1

            if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)

    def rotateLeft(self,rotRoot):                    ## NOTE:  You will complete rotateRight in Lab
        newRoot = rotRoot.rightChild
        rotRoot.rightChild = newRoot.leftChild
        if newRoot.leftChild != None:
            newRoot.leftChild.parent = rotRoot
        newRoot.parent = rotRoot.parent
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        newRoot.leftChild = rotRoot
        rotRoot.parent = newRoot
        rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFactor, 0)
        newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)

    def rebalance(self,node):
        if node.balanceFactor < 0:
            if node.rightChild.balanceFactor > 0:
                self.rotateRight(node.rightChild)
                self.rotateLeft(node)
            else:
                self.rotateLeft(node)
        elif node.balanceFactor > 0:
            if node.leftChild.balanceFactor < 0:
                self.rotateLeft(node.leftChild)
                self.rotateRight(node)
            else:
                self.rotateRight(node)
```

c) Trace the code for `myAVL.put(90,None)` by updating the below diagram:



Consider balance factor formulas for rotateLeft. We know: $newBal(B) = h_A - h_C$ and $oldBal(B) = h_A - (1+max(h_C, h_E))$
$newBal(D) = 1+ max(h_A, h_C) - h_E$ and $oldBal(D) = h_C - h_E$



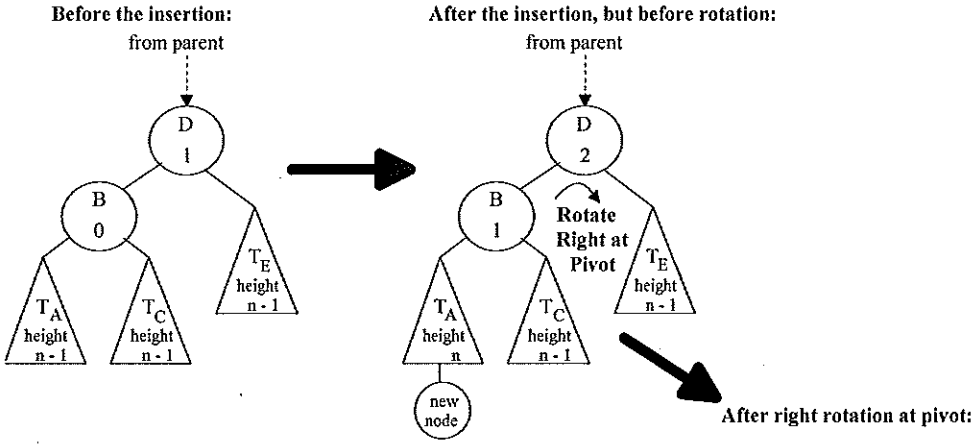


Consider: $newBal(B) - oldBal(B)$
$newBal(B) - oldBal(B) = (h_A - h_C) - [h_A - (1+max(h_C, h_E))]$
$newBal(B) - oldBal(B) = h_A - h_C - h_A + (1+max(h_C, h_E))$
$newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C$
$newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C$
$newBal(B) = oldBal(B) + 1 + max(h_C - h_C, h_E - h_C)$
$newBal(B) = oldBal(B) + 1 + max(0, -oldBal(D))$
$newBal(B) = oldBal(B) + 1 - min(0, oldBal(D))$, so

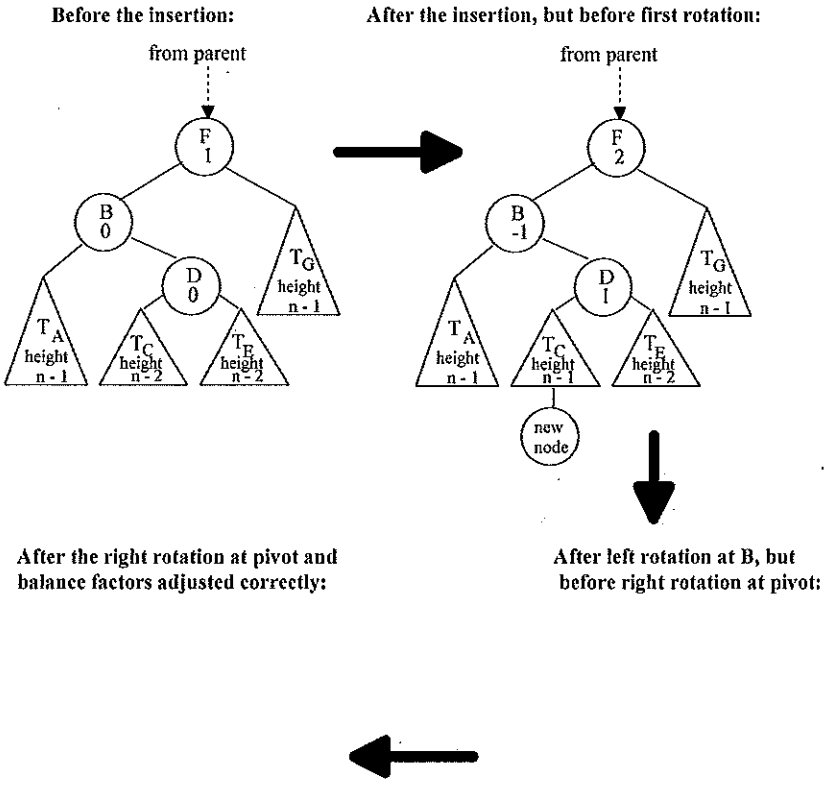
```
rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
                        min(newRoot.balanceFactor, 0)
```

d) Consider: newBal(D) - oldBal(D)

3. Complete the below figure which is a "mirror image" to the figure on page 2, i.e., inserting into the pivot's left child's left subtree. Include correct balance factors after the rotation.



b) Complete the below figure which is a "mirror image" to the figure on page 3, i.e., inserting into the pivot's left child's right subtree. Include correct balance factors after the rotation.



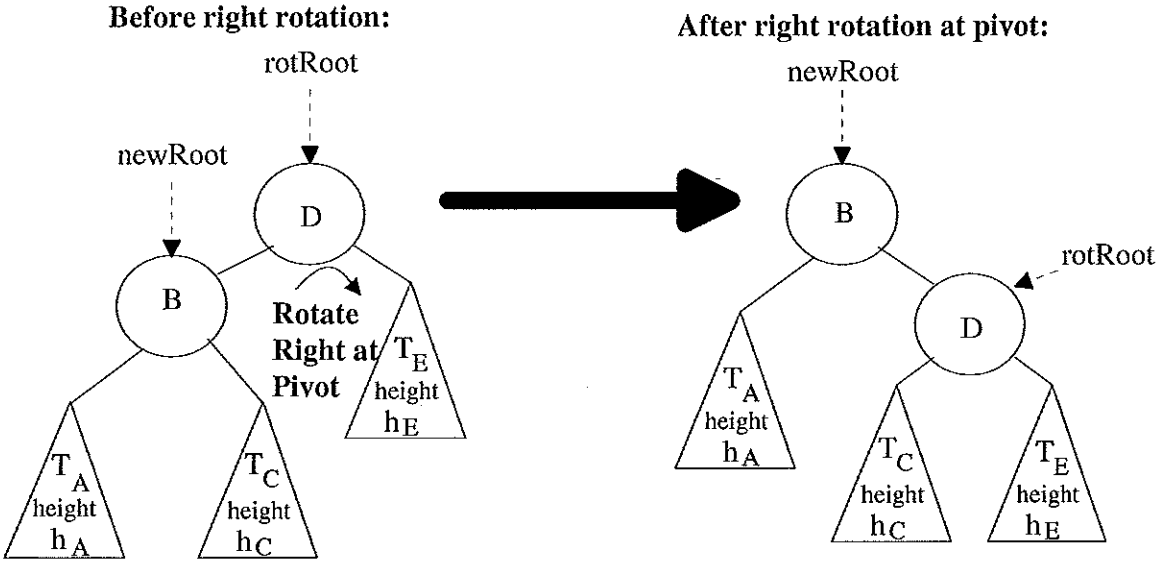
Data Structures (CS 1520) Lab 10 Name:_____

Objectives: You will gain experience with AVL `put` implementation

To start the lab: Download and unzip the file: http://www.cs.uni.edu/~fienup/cs1520f18/labs/lab10.zip

Part A: In lecture 23 we discussed the AVL tree `rotateLeft` method. For this lab you need to implement the `rotateRight` method. The `rotateRight` method has two parts:

- updating the "pointers" to the nodes to do the rotation (look at the `rotateLeft` method code)
- updating the balanceFactors for the rotRoot and newRoot nodes (you need to use math similar to lecture 23) (See below too)



Consider the balance factor formulas for `rotateRight`. We know from the above diagram:

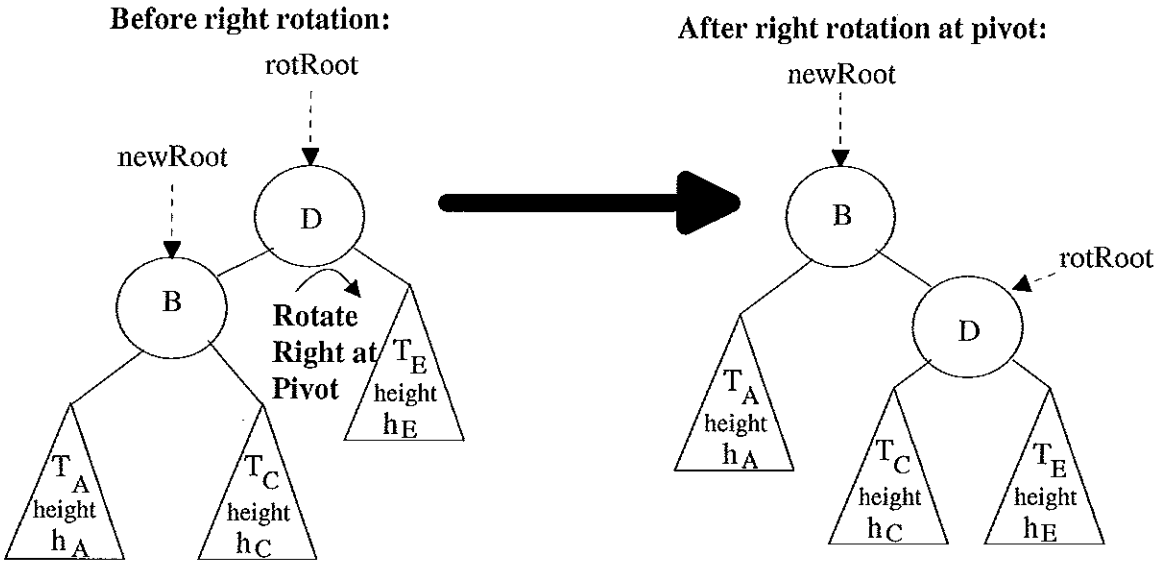
oldBal(B) = h_A - h_C and
oldBal(D) = (1+*max*(h_A , h_C)) - h_E

newBal(B) = h_A - (1 + *max*(h_C, h_E)) and
newBal(D) = h_C - h_E

To determine *newBal(D)*, consider:

newBal(D) - *oldBal(D)* =

(See back for *newBal(B)* calculation)



Consider the balance factor formulas for `rotateRight`. We know from the above diagram:

oldBal(B) = h_A - h_C and

oldBal(D) = (1+*max*(h_A, h_C)) - h_E

newBal(B) = h_A - (1 + *max*(h_C, h_E)) and

newBal(D) = h_C - h_E

To determine *newBal(B)*, consider:

newBal(B) - oldBal(B) =

After completing your implementation of rotateRight, test your code by running the avl_tree.py program. Once you think it is working, run the timeAVLTree.py program. The height of AVL tree after adding in sorted order should be 13, and the height of AVL tree after adding in shuffled order should be about 15.

When it is working, raise your hand and we will check you off.

(If you have extra time, consider working on previous labs or homeworks.)

Data Structures (CS 1520) Lecture 24 Name:_____

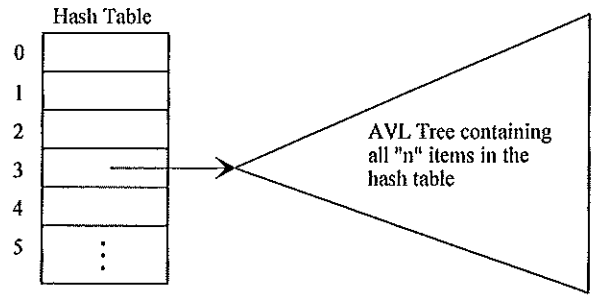
1. BST, AVL trees, and hash tables can all be used to implement a dictionary ADT.

Dictionary Successful Search Comparisons with 10,000 integer items (Time in seconds)						
	Items added in sorted order		Items added in random order		Order did not matter (Hash table sizes 2^{15} = 32K)	
	BST	AVL Tree	BST	AVL Tree	Open Addr. (Quadratic)	Closed Addr. (Chaining)
Total add/put time	47.785	0.205	0.119	0.195	0.064	0.074
Total search time	38.100	0.060	0.079	0.062	0.044	0.039
Height of resulting tree	9,999	13	30	15	NA	NA

a) The `puts` of these 10,000 randomly ordered items into the BST took 0.119 seconds and 0.195 seconds into the AVL tree. Why did the BST `puts` take less time eventhough the final height was 30 vs. a final AVL tree height of 15?

b) With a very, very poor hash function or very, very bad choice of keys, then all keys could hash to the same home address.

- What would be the worst-case big-oh of open-address hashing with quadratic probing?
- What would be the worst-case big-oh of chaining using a linked list at each home address (i.e., ChainingDict)?
- What would be the worst-case big-oh of chaining using an AVL tree at each home address?

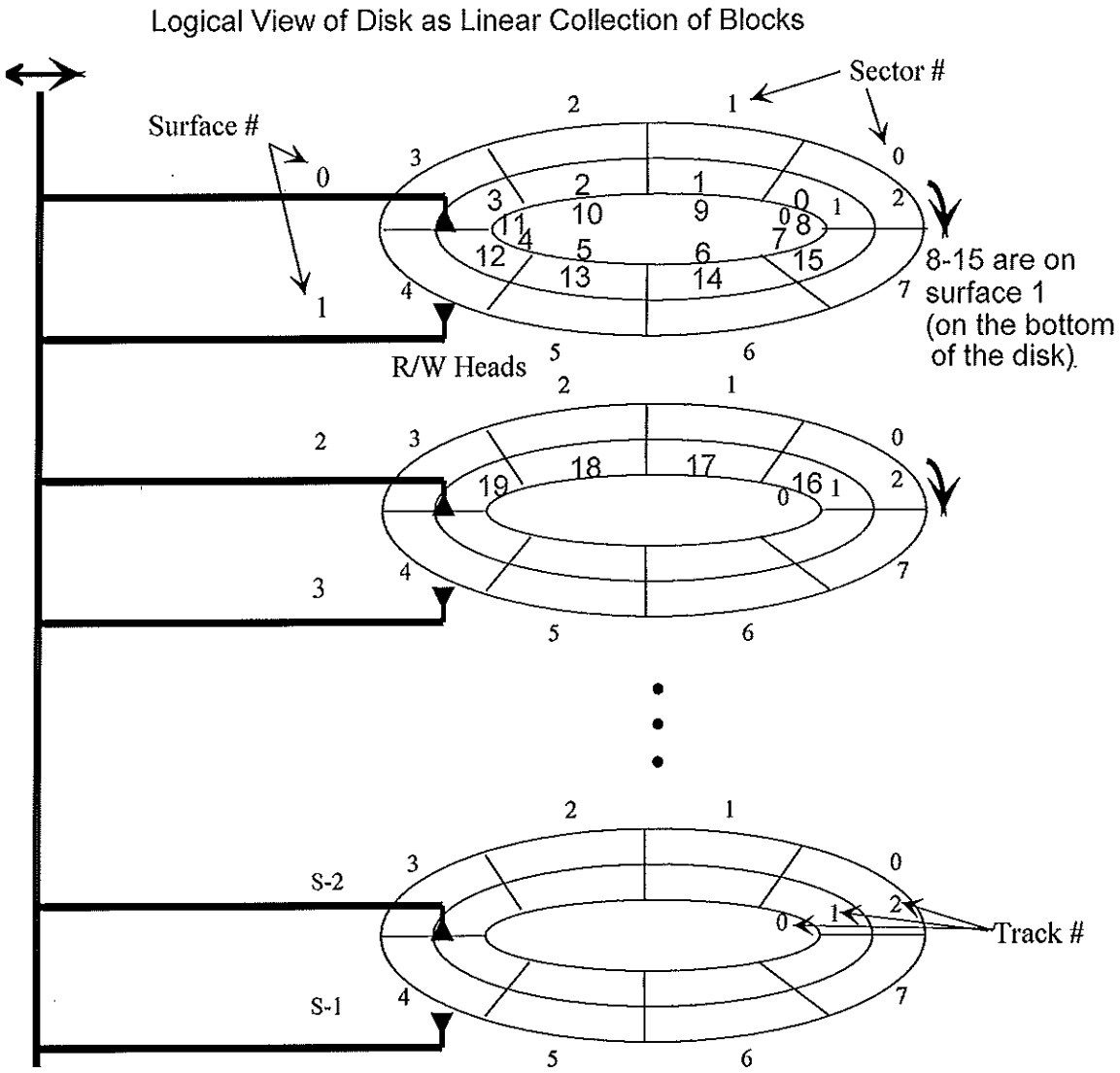


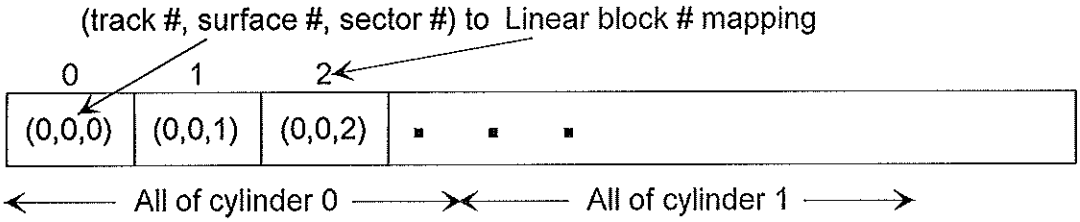
2. The data structures we have discussed so far are all in-memory, i.e., data is stored in main/RAM memory. Data can also be stored on secondary storage in a file (e.g., moiveData.txt file). Currently, most secondary storage consists of hard-disks.

a) Complete the following table comparing main/RAM memory vs. hard-disk:

Criteria	Main/RAM memory	Hard-disk Drive	Solid-State Drive
Size on a typical desktop computer			
Average access time			

b) Which criterion seems to be the most important difference between the main and secondary memories?





Bits of linear block # :

track #	surface #	sector #

3. Disk-access time = (seek time) + (rotational delay) + (date transfer time). How is each component of the disk-access time effected by increasing the disk's RPMs (revolutions per minute)?

b) If we want fast access to a collection of sectors, where can we place them to minimize seek time and rotational delay?

User Program - HLL programming language make system calls to OS to:

1. open file - establish a link between file variable and file for either reading, writing, or both
2. access file - read or write one piece of data at a time (e.g., char., record, etc.)
3. close file - flush changes to disk

Operating System - manage and control access to secondary storage through its file system which contains information about every file: location on disk, ownership and security/protection

- OS views disk as linear sequence of blocks (block 0, block 1, etc.), but assumes closeness in block # means close with respect to access time.
- OS buffers some blocks in memory to improve efficiency
- OS maintains free disk space "list"

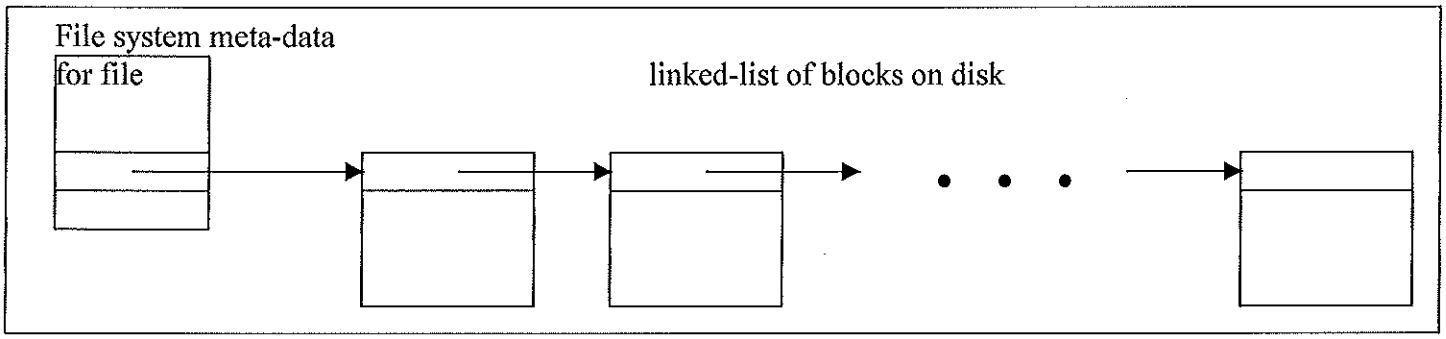
Secondary Storage Device - accepts R/W requests from OS for block# and maps block# to internals physical address (e.g., (track #, surface #, sector #) - more complex than above picture!)

Kinds of File Access:

- serial/sequential files - open at the beginning and read sequentially from beginning to end linearly
- random-access files - "*seek*" to any position by specifying a byte-offset from the beginning of the file, record #, etc.
- random-access of a record by key

Implementation of Files on Disk- how are blocks allocated?

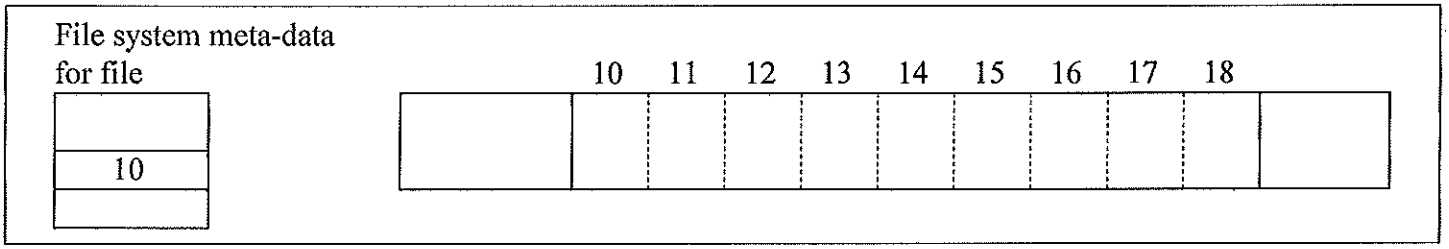
4. non-contiguous - scattered across linear address space of OS and disk



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

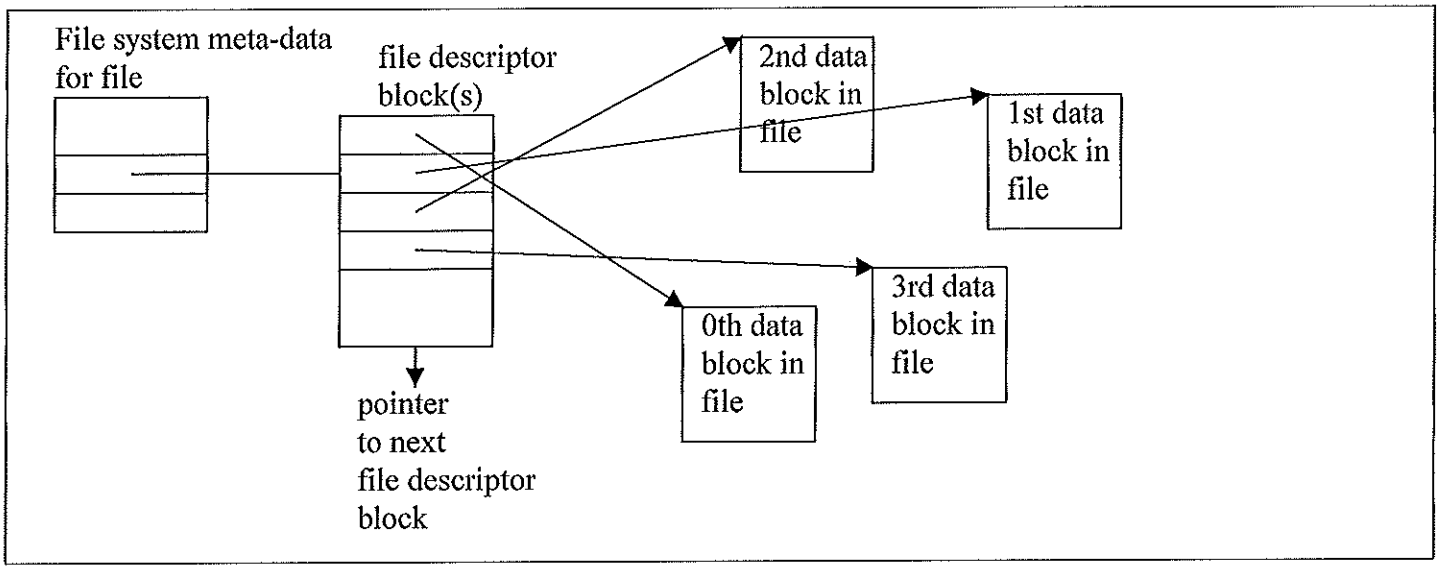
5. contiguous - sequential collection of blocks from OS linear view of disk



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

6. file descriptor blocks - list of blocks hold the address of the physical location of data blocks



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

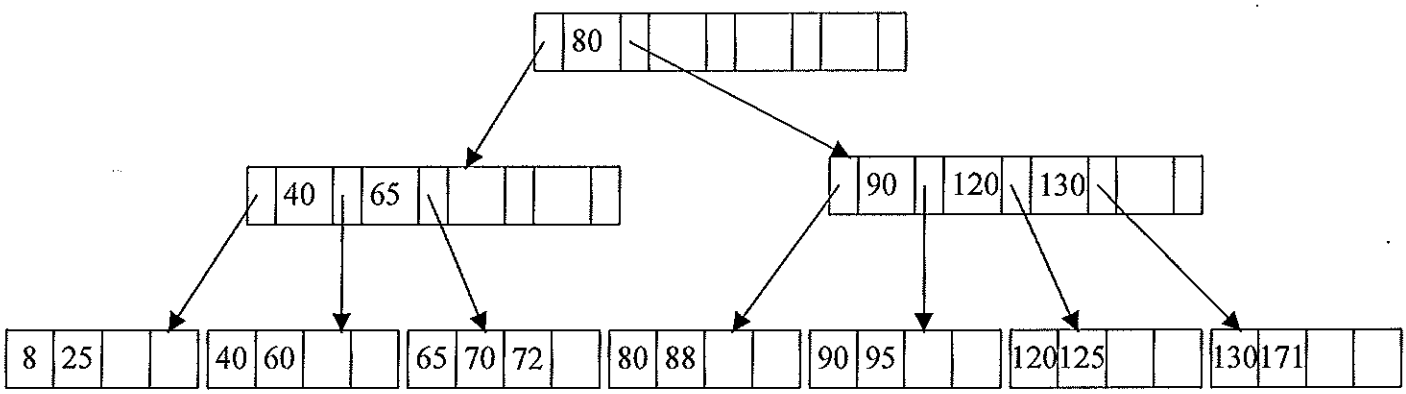
7. To implement "random-access of a record by key" in a file how might we use hashing?

8. To implement "random-access of a record by key" in a file why would an AVL tree not work well?

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 $\lceil b/2 \rceil \leq$ # 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?

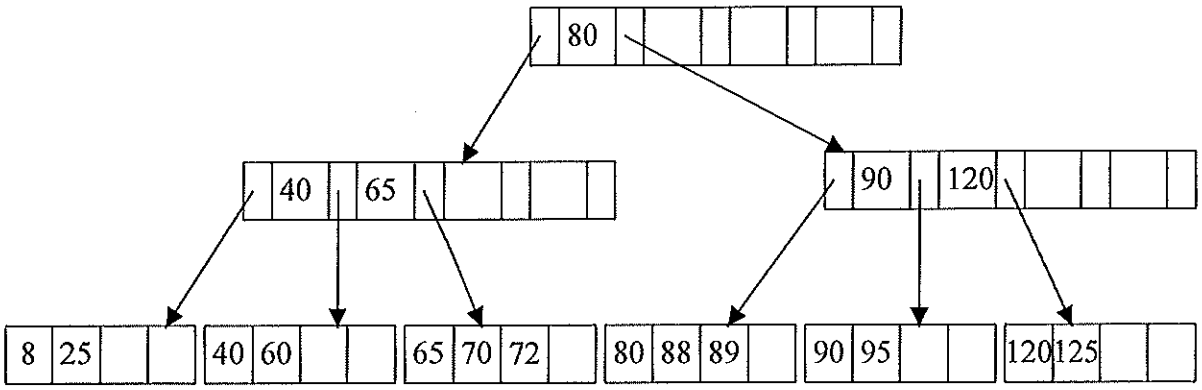
Situation		insertion Algorithm
Data Page Full?	Parent Index Page Full?	
No	No	Place record in sorted position in the appropriate data page.
Yes	No	1. Split data page with records < middle key going in left data page and records ≥ middle key going in right data page. 2. Place middle key in index page 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.
Yes	Yes	1. Split data page with records < middle key going in left data page and records ≥ middle key going in right data page. 2. Adding middle key to parent index page causes it to split with keys < middle key going into the left index page, keys > 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.

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?

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

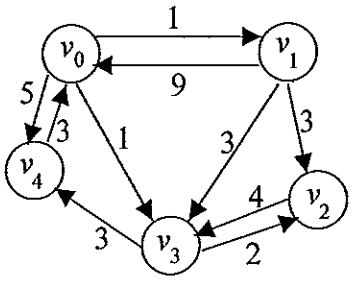
Situation		deletion Algorithm
Data Page Below Fill Factor?	Parent Index Page Below Fill Factor?	
No	No	Delete record from the data page. Shifting records with larger keys to left to fill in the hole. If the deleted key appears in the index page, use the next key to replace it.
Yes	No	1. Combine data 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?



Data Structures (CS 1520) Lecture 25 Name:_____

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



a) What is the set of vertices? $V =$

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

$E =$

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.

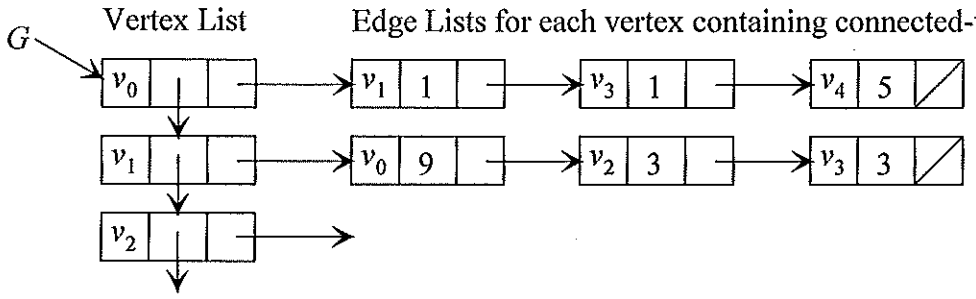
d) A cycle in a directed graph is a path that starts and ends at the same vertex. Find a cycle in the above graph.

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 ∞.)

(from vertex) \ (to vertex)	v_0	v_1	v_2	v_3	v_4
v_0	0	1	∞	1	5
v_1	9	0	3	3	∞
v_2					
v_3					
v_4					

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:



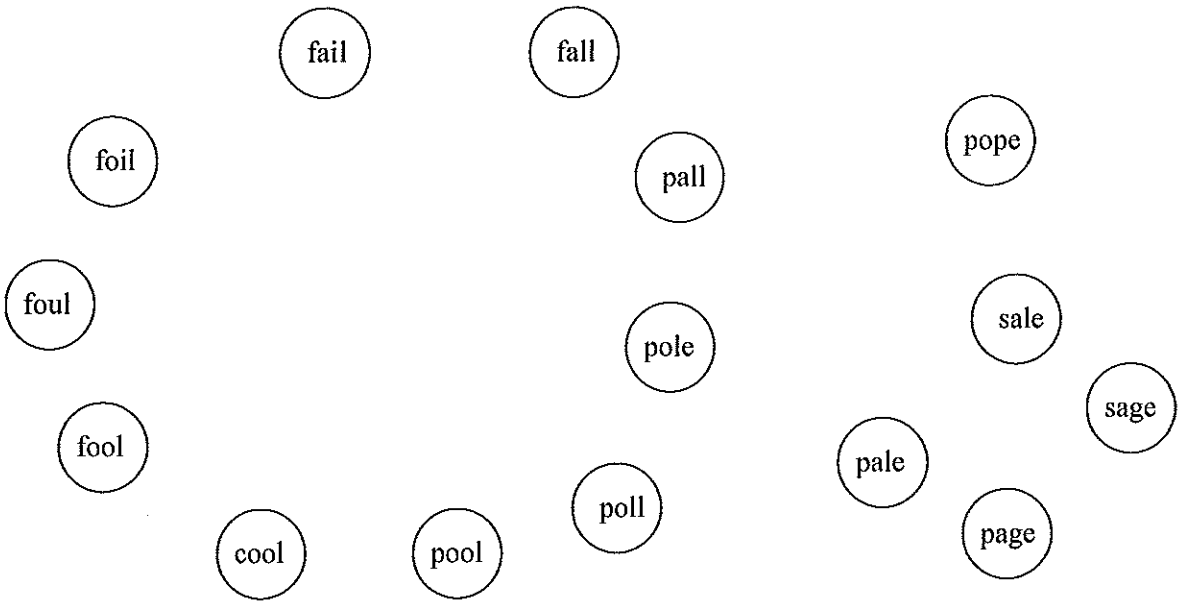
3. Graphs can be used to solve many problems by modeling the problem as a graph and using "known" graph algorithm(s). For example, consider the *word-ladder puzzle* where you tranform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL →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

- a word ladder transformation from one word to another represents?

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



a) List a different transformation from FOOL to SAGE

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

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?

Data Structures **Lab 11 Graphs** Name:_____

Objectives: To understand how a graph can be represented and traversed.

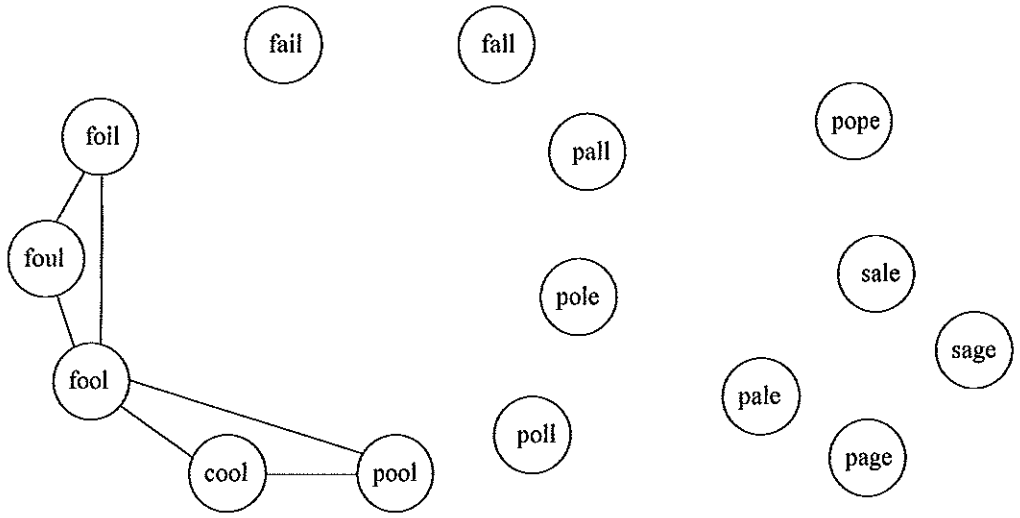
To start the lab: Download and unzip the file http://www.cs.uni.edu/~fienup/cs1520f18/labs/lab11.zip

Part A: In a word-ladder puzzle (discussed in class) transforms 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 used a graph algorithm to solve this problem by constructing a graph such that

- words are represented by the vertices, and
- edges connect vertices containing words that differ by only one letter

a) For the words listed below, complete the graph by adding edges as defined above.

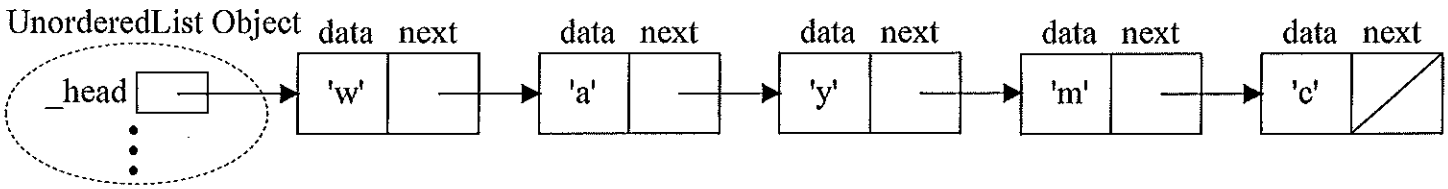


b) To find the shortest transformation from FOOL to SAGE, why did we decide on using a Breadth First Search (BFS) traveral (i.e., where you find all vertices a distance 1 (directly connected) from FOOL, before finding all vertices a distance 2 from FOOL, etc) instead of a Depth-First Search (DFS) traversal?

c) Run the `lab11/word_ladder_BFS.py` program. Examine the "enqueue" and "dequeue" lines of output produced by the `bfs(g,g.getVertex("fool"))` call. Does this output match the expected "enqueues" and "dequeues" performed during a bfs of the above graph starting at "fool"?

d) The `bfs` algorithm sets the value of each vertex's `predecessor` to point to the vertex object that enqueued it. **Add code to the end of the `word_ladder_BFS.py` program** that traverses the "linked list" of `predecessor` references from "sage" to "fool." and prints the corresponding word ladder from "fool" to "sage."

Hint: The code you need to write is similar to the `__str__` code for traversing a singly-linked list:



```
def __str__(self):
    resultStr = ""
    current = self._head
    while current != None:
        resultStr += " " + str(current.getData())
        current = current.getNext()
    return resultStr
```

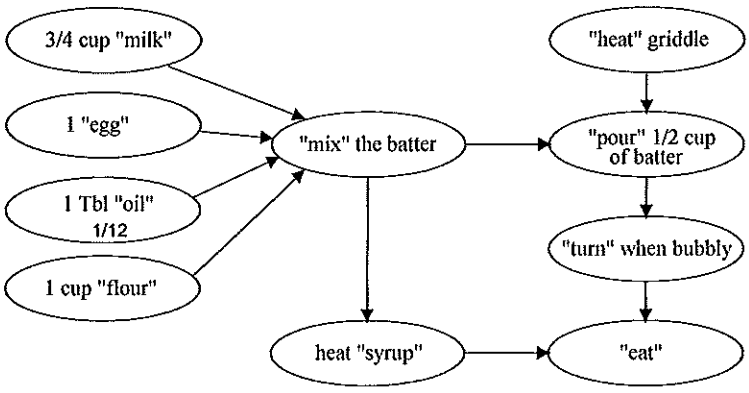
Your code (see partial code at the end of the `word_ladder_BFS.py` program) can:

- walk `currentVert` down the linked list of `Vertex` objects using `getPred` instead of `getNext`
- append to `wordLadderList` the `currentVert`'s word gotten using `getId` instead of string concatenating using `getData`

After your while-loop executes, you can reverse the `wordLadderList` and print the transformation from "fool" to "sage."

After you have answered the above questions and completed the code, raise your hand and explain your answers.

Part B: Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The `DFSGraph` uses a `time` attribute to note when a vertex if first encountered (`discovery` attribute) in the depth-first search and when a vertex in backtracked through (`finish` attribute). Consider the graph for making pancakes where vertices are steps and edges represents the partial order among the steps.




```
from graph import Graph
class DFSGraph(Graph):

    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self,startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in
startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

a) Run the `lab11/make_pancake_DFS.py` program. Write on the above graph the `discovery` and `finish` attributes (e.g., 1 / 12 of "oil") assigned to each vertex by executing the `dfs` method..

b) A *topological sort* algorithm can use the dfs `finish` attributes to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour ½ cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. Outline the steps to perform a topological sort from the `finish` attributes.

After you have answered the above questions, raise your hand and explain your answers.

EXTRA CREDIT:
Add code to the end of the `made_pancake_DFS.py` program to print the topological sort for making pancakes.

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?

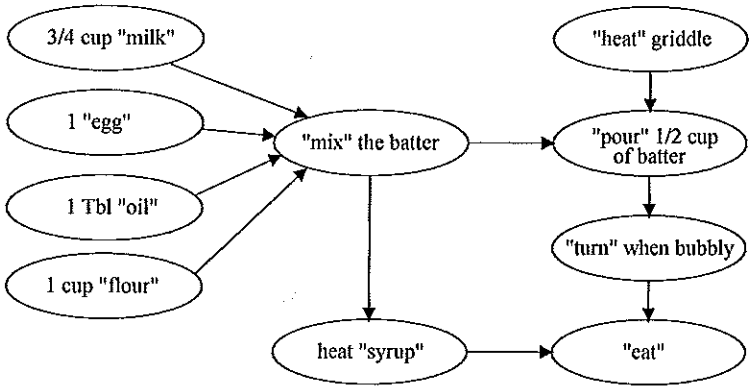
3. Assuming a graph `g` containing the word-ladder graph from lecture 25, on the diagram trace the `bfs` algorithm by showing the value of each vertex's `color`, `predecessor`, and `distance` attributes?

```
""" 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')
```

4. Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The `DFSGraph` uses a `time` attribute to note when a vertex if first encountered (`discovery` attribute) in the depth-first search and when a vertex in backtracked through (`finish` attribute). Consider the graph for making pancakes where vertices are steps and edges represents the partial order among the steps.

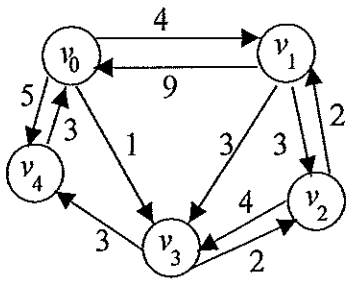



```
from graph import Graph
class DFSGraph(Graph):

    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self,startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

a) Assume (why is this a bad assumption???) that the for-loops alway iterate through the vertexes alphabetically (e.g., "eat", "egg", "flour", ...) by their id. **Write on the above graph the `discovery` and `finish` attributes assigned to each vertex by executing the `dfs` method.**

b) A *topological sort* algorithm can use the dfs `discovery` and `finish` attributes to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour ½ cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. Outline the steps to perform a topological sort.

5. Consider the following directed graph (diagraph).



Dijkstra's Algorithm is a *greedy algorithm* that finds the shortest path from some vertex, say v_0, to all other vertices. A *greedy algorithm,* unlike divide-and-conquer and dynamic programming algorithms, DOES NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices). Dijkstra's algorithm builds a subgraph by repeatedly selecting the next closest vertex to v_0 that is not already in the subgraph. Initially, only vertex v_0 is in the subgraph with a distance of 0 from itself.

a) What would be the order of vertices added to the subgraph during Dijkstra's algorithm?

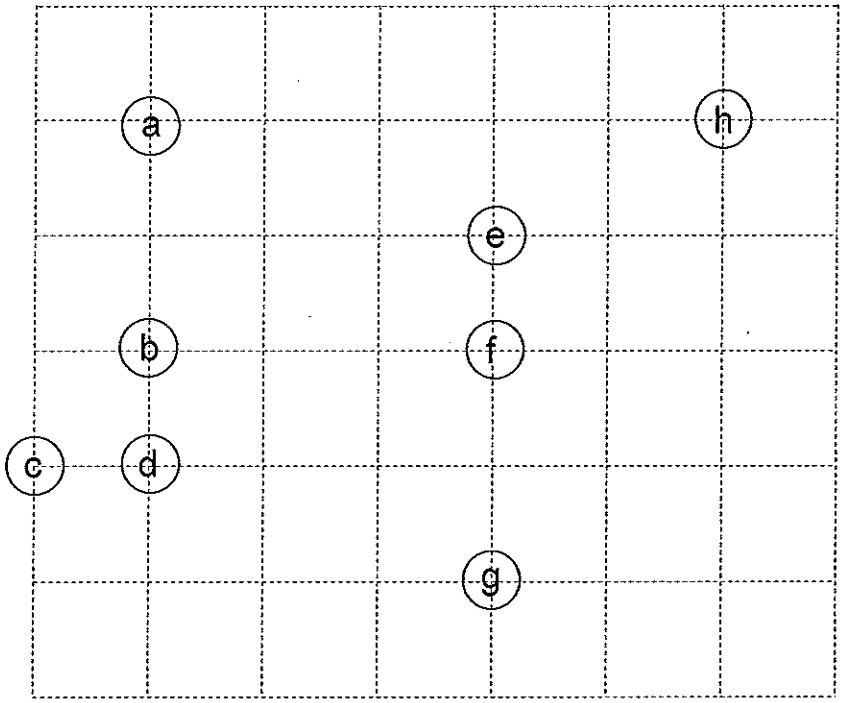
v_0,

b) What *greedy criteria* did you use to select the next vertex to add to the subgraph?

c) What data structure could be used to efficiently determine that selection?

d) How might this data structure need to be modified?

1. Suppose you had a map of settlements on the planet X
   (Assume edges could connecting all vertices with their Euclidean distances as their costs)



We want to build roads that allow us to travel between any pair of cities. Because resources are scarce, we want the total length of all roads build to be minimal. Since all cities will be connected anyway, it does not matter where we start, but assume we start at "a".

a) Assuming we start at city "a" which city would you connect first? Why this city?

b) What city would you connect next to expand your partial road network?

c) What would be some characteristics of the resulting "graph" after all the cities are connected?

d) Does your algorithm come up with the overall best (globally optimal) result?

2. Prim's algorithm for determining the minimum-spanning tree (MST) of a graph is another example of a *greedy algorithm*. Unlike divide-and-conquer and dynamic programming algorithms, greedy algorithms DO NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices).

a) What greedy criteria does Prim's algorithm use to select the next vertex and edge to the partial minimum spanning tree?

b) Consider the textbook's Prim's Algorithm code (Listing 7.12 p. 346) which is incorrect.

```
def prim(G,start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert) \
                    + currentVert.getDistance()
          if v in pq and newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```

c) What is wrong with the code? (Fix the above code.)

3. To avoid "massive" changes to the `BinHeap` class, it can store `PriorityQueueEntry` objects:

```
class PriorityQueueEntry:
    def __init__(self,x,y):
        self.key = x
        self.val = y

    def getKey(self):
        return self.key

    def getValue(self):
        return self.val

    def setValue(self, newValue):
        self.val = newValue
```

```
    def __lt__(self,other):
        return self.key < other.key

    def __gt__(self,other):
        return self.key > other.key

    def __eq__(self, other):
        return self.val == other.val

    def __hash__(self):
        return self.key
```

a) Update the above Prim's algorithm code to use `PriorityQueueEntry` objects.
b) Why do the `__lt__` and `__gt__` methods compare `key` attributes, but `__eq__` compare `val` attributes?

c) When used for Prim's algorithm what type of objects are the `vals` compared by `__eq__`?

d) What changes to the Graph and Vertex classes need to be made?

e) Complete the `__contains__` and `decreaseKey` methods.

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def buildHeap(self,alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1

    def percDown(self,i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self,i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i*2] < self.heapList[i*2+1]:
                return i * 2
            else:
                return i * 2 + 1

    def percUp(self,i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i//2]:
               tmp = self.heapList[i // 2]
               self.heapList[i // 2] = self.heapList[i]
               self.heapList[i] = tmp
            i = i // 2

    def insert(self,k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def isEmpty(self):
        return self.currentSize == 0

    def size(self):
        return self.currentSize

    def __str__(self):
        return str(self.heapList[1:])
```

```
def __contains__(self, value):
```

```
def decreaseKey(self, decreasedValue):
    """Precondition:  decreasedValue
       in heap already"""
```

Data Structures **Lab 12 Graphs 2** Name:_____

Objectives: To understand how a graph can be used to solve graph algorithms.

To start the lab: Download and unzip the file lab12.zip

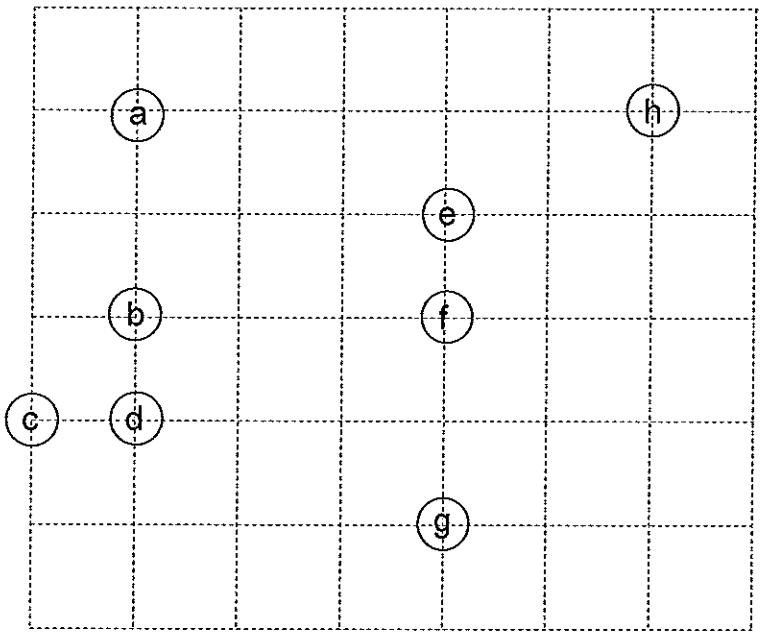
Part A: In IDLE open the binary heap class file: `lab12/binheap.py`. The changes discussed in lecture are included:

- `PriorityQueueEntry` class, and
- additional `BinHeap` methods: `__contains__` and `decreaseKey`.

a) Look at the `buildHeap` method. Explain how it takes a list of values and builds them into a heap.

b) Run the `lab12/make_min_spanning_tree.py` program which uses Prim's algorithm on the graph from lecture. Does it give the expected output?

c) Predict the order of edges added by Prim's algorithm if we start at vertex "e":



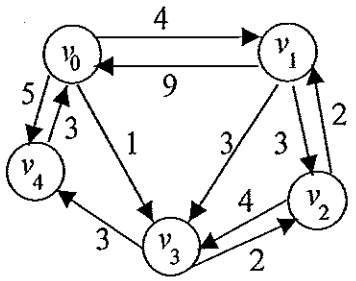
d) Modify the `lab12/make_min_spanning_tree.py` program to verify your prediction. NOTE: This is a very easy modification. You just need to start Prim's algorithm starting at the vertex labeled "e".

After you have answered the above questions and completed the code, raise your hand and explain your answers.

Part B: The textbook's Dijkstra's Algorithm code (Listing 7.11 p. 341) needs to be updated similarly to Prim's algorithm.

a) Modify the `dijkstra` method in the `lab12/graph_algorithms.py` file similar to Prim's so that it uses `PriorityQueueEntry` objects among other corrections.

b) Run the `lab12/test_dijkstra.py` program which uses Dijkstra's algorithm on the graph from lecture. Does it give the expected output?



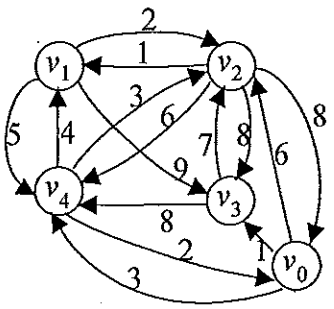
After you have fixed the dijksta method in `lab12/graph_algorithms.py`, raise your hand and demonstrate your code.

EXTRA CREDIT Opportunities:

1. Add code to the end of the `test_dijkstra.py` program to print the shortest paths from v_0 to each of the other vertices. One line of output might look something like:
   "Shortest path from v0 to v4 is v0 > v3 > v4 with a total distance of 4"

2. As implemented the decreaseKey method must do a O(n) search for the item whose value is being decreased. To avoid this search, modify the PriorityQueueEntry object to maintains an "index" data attribute indicating its index in the BinHeap's heapList. Several of the BinHeap methods must also be modified to keep this index data attribute up to date.

1. *Traveling Salesperson Problem* (TSP) -- Find an optimal (i.e., minimum length) tour when at least one tour exists. A *tour* (or *Hamiltonian circuit*) is a path from a vertex back to itself that passes through each of the other vertices exactly once. (Since a tour visits every vertice, it does not matter where you start, so we will generally start at v_0.)

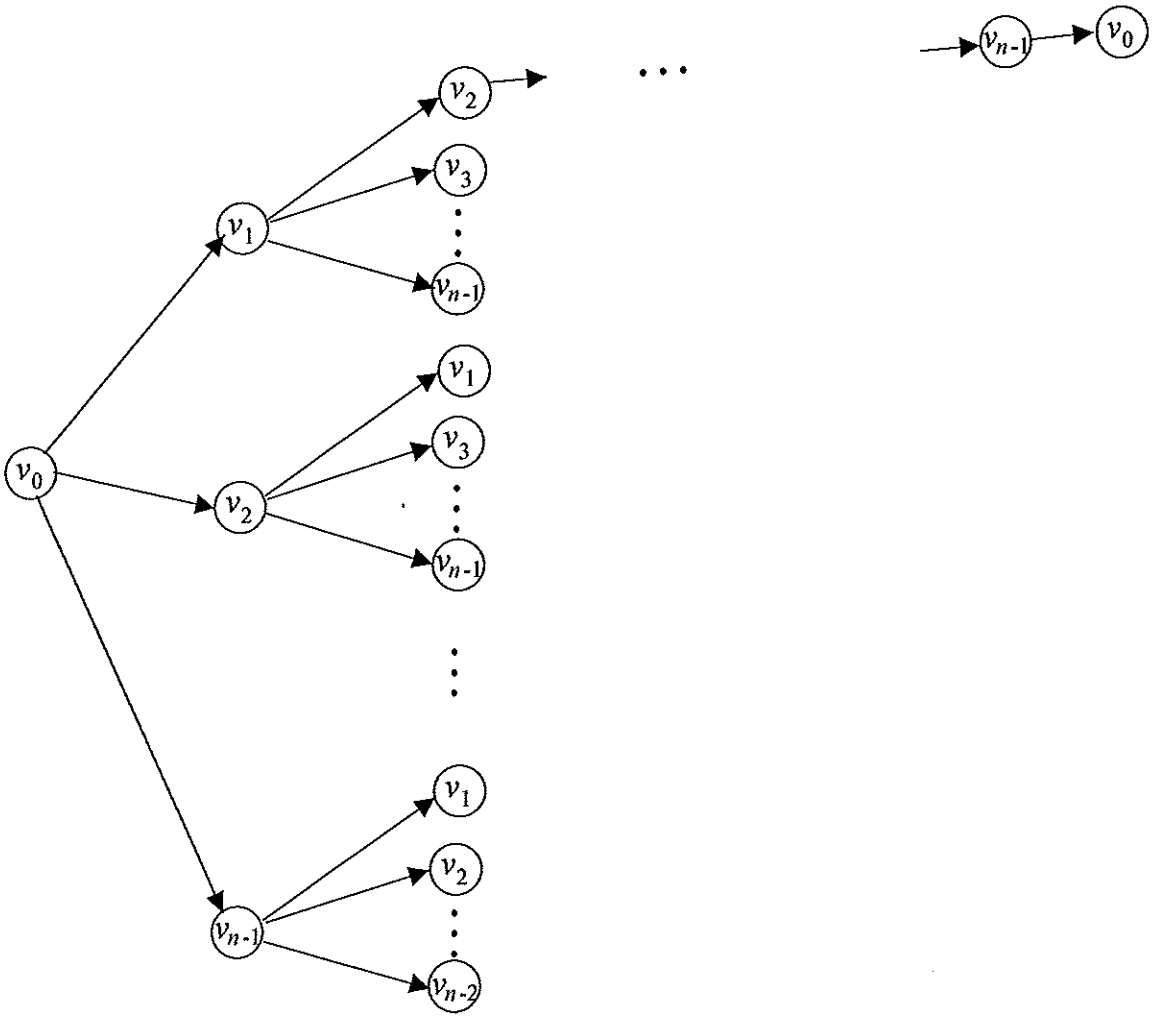


What are the length of the following tour?

a) $[v_0, v_3, v_4, v_1, v_2, v_0]$

b) List another tour starting at v_0 and its length.

c) For a graph with "n" vertices ($v_0, v_1, v_2, \ldots, v_{n-1}$), one possible approach to solving TSP would be to brute-force generate all possible tours to find the minimum length tour. "Complete" the following decision tree to determine the number of possible tours.



Unfortunately, TSP is an "*NP-hard*" problem, i.e., no known polynomial-time algorithm.

2. **Handling "Hard" Problems**: For many optimization problems (e.g., TSP, knapsack, job-scheduling), the best known algorithms have run-time's that grow exponentially ($O(2^n)$ or worse). Thus, you could wait centuries for the solution of all but the smallest problems!

Ways to handle these "hard" problems:

- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the 2^n worse solutions, e.g, Backtracking (section 4.6) and Best-first-search-branch-and-bound
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., $O(n^3)$) solution. e.g., TSP problem satisfying the triangle inequality, Fractional Knapsack problem
- Use an approximation algorithm to find a good, but not necessarily optimal solution

Backtracking general idea: (Recall the coin-change problem from lectures 10 and 13)

- Search the "*state-space tree*" using depth-first search to find a suboptimal solution quickly
- Use the best solution found so far to prune partial solutions that are not "promising,", i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time. However, in the worst case, the algorithm is still exponential.

My simple backtracking solution for the coin-change problem **without pruning**:

```
def recMC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
       return 1
    else:
       for i in coinValueList:
           if i <= change:
               numCoins = 1 + recMC(change - i, coinValueList)
               if numCoins < minCoins:
                   minCoins = numCoins
    return minCoins
```

Results of running this code:

```
Change Amount: 63 Coin types: [1, 5, 10, 25]
Run-time: 45.815 seconds
Fewest number of coins 6
Number of Backtracking Nodes: 67,716,925
```

Consider the output of running the backtracking code **with pruning** twice with a change amount of 63 cents.

```
Change Amount: 63  Coin types: [1, 5, 10, 25]
Run-time: 0.036 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 1-cent coins is 3
number of 5-cent coins is 0
number of 10-cent coins is 1
number of 25-cent coins is 2
Number of Backtracking Nodes: 4831
```

```
Change Amount: 63  Coin types: [25, 10, 5, 1]
Run-time: 0.003 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 25-cent coins is 2
number of 10-cent coins is 1
number of 5-cent coins is 0
number of 1-cent coins is 3
Number of Backtracking Nodes: 310
```

a) With the coin types sorted in ascending order what is the first solution found?

b) How useful is the solution found in (a) for pruning?

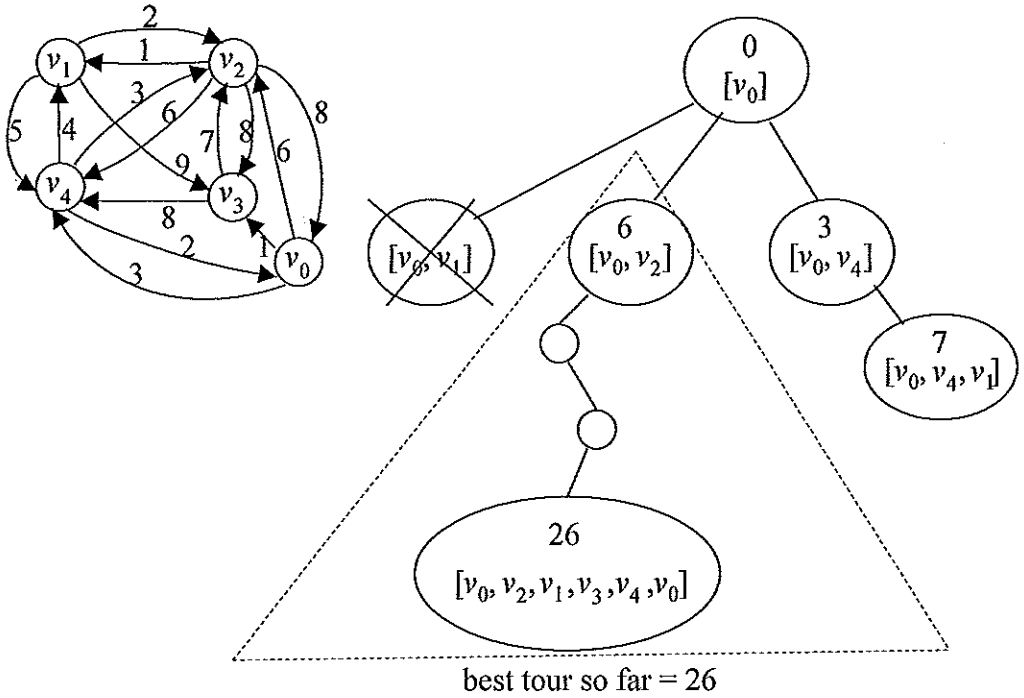
c) With the coin types sorted in descending order what is the first solution found?

d) How useful is the solution found in (c) for pruning?

e) For the coin-change problem, backtracking is not the best problem-solving technique. What technique was better?

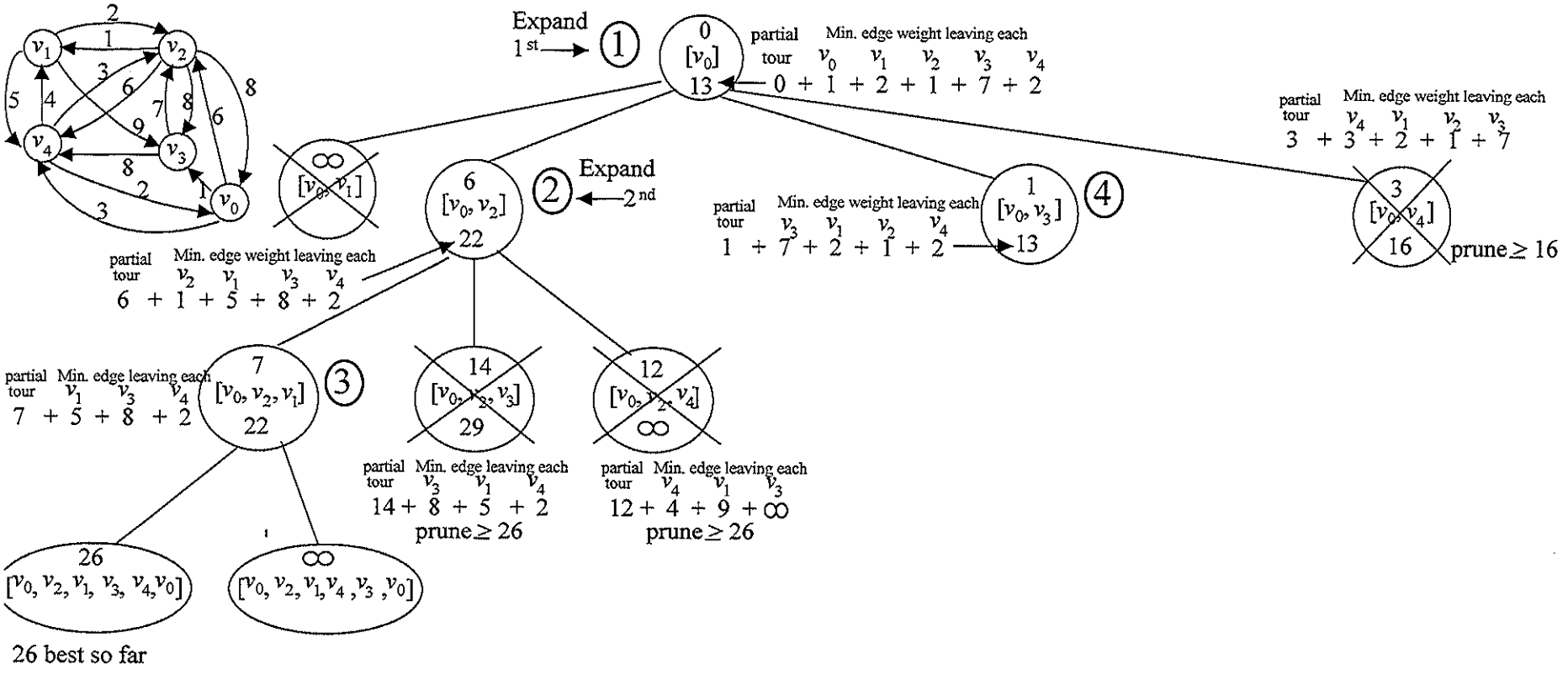
3. a) For the TSP problem, why is backtracking the best problem-solving technique?

b) To prune a node in the search-tree, we need to be certain that it cannot lead to the best solution. How can we calculate a "bound" on the best solution possible from a node (e.g., say node with partial tour: $[v_0, v_4, v_1]$)?



best tour so far = 26

2. To prune a node in the search-tree, we need to be certain that it cannot lead to the best solution. We can calculate a "bound" on the best solution possible from a node (e.g., say node with partial tour: $[v_0, v_4, v_1]$) by summing the partial tour with the minimum edges leaving the remaining nodes going some place reasonable. Complete the backtracking state-space tree with pruning.

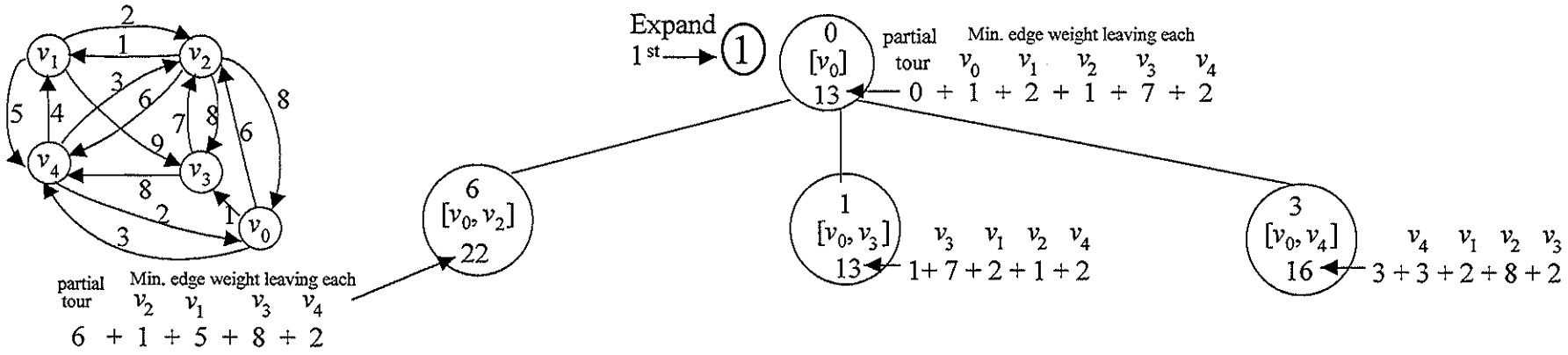


3. An alternative to backtracking, is the *Best-First search with Branch-and-Bound* approach:

- It does not limit us to any particular search pattern in the state-space tree like backtracking's left-most branch to right-most branch
- It calculates a "bound" estimate for each node that indicates the "best" possible solution that could be obtained from any node in the subtree rooted at that node, i.e., how "promising" following that node might be
- It expands the most promising ("best") node first by visiting its children

a) What type of data structure would we use to find the most promising node to expand next?

b) Complete the best-first search with branch-and-bound state-space tree with pruning. Indicate the order of nodes expanded.



Approximation Algorithm for TSP with Triangular Inequality

Restrictions on the weighted, undirected graph G=(V, E):

1. There is an edge connecting every two distinct vertices.

2. Triangular Inequality: If W(u, v) denotes the weight on the edge connecting vertex u to vertex v, then for every other vertex y,

$$W(u, v) \leq W(u, y) + W(y, v).$$

NOTES:

- These conditions satisfy automatically by a lot of natural graph problems, e.g., cities on a planar map with weights being as-the-crow-flys (Euclidean distances).
- Even with these restrictions, the problem is still NP-hard.

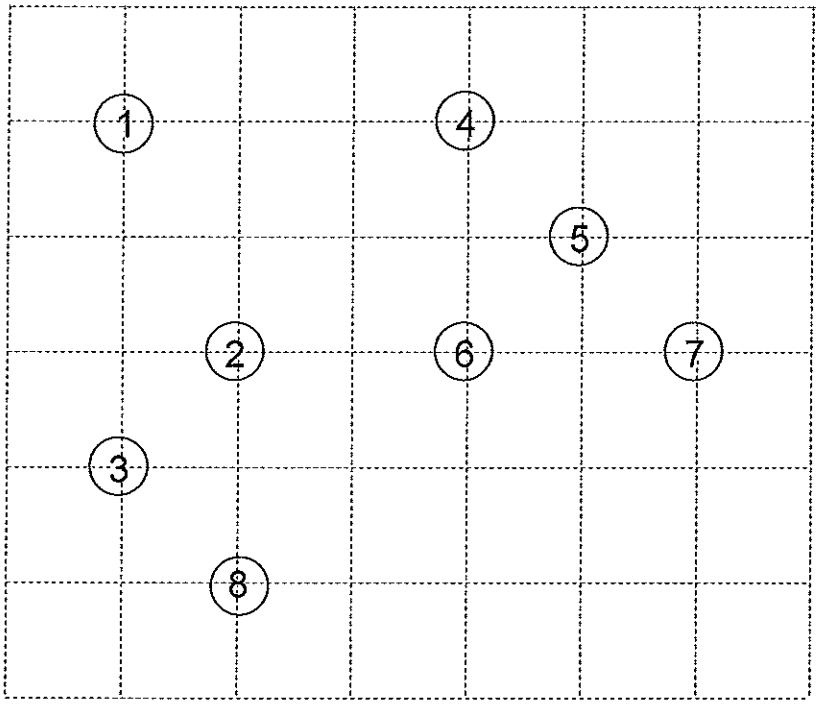
A simple TSP approximation algorithm:

Step 1. Determine a Minimum Spanning Tree (MST) for G (e.g., Prim's Algorithm section 7.8.3)

Step 2. Construct a path that visits every node by performing a preorder walk of the MST. (A *preorder walk* lists a tree node every time the node is encounter including when it is first visited and "backtracked" through.)

Step 3. Create a tour by removing vertices from the path in step 2 by taking shortcuts.

1) (Step 1) Determine a Minimum Spanning Tree (MST) for G (e.g., Prim's Algorithm) if we start with vertex 1 in the MST. (Assume edges connecting all vertices with their Euclidean distances)



Prim's algorithm is a greedy algorithm that performs the following:

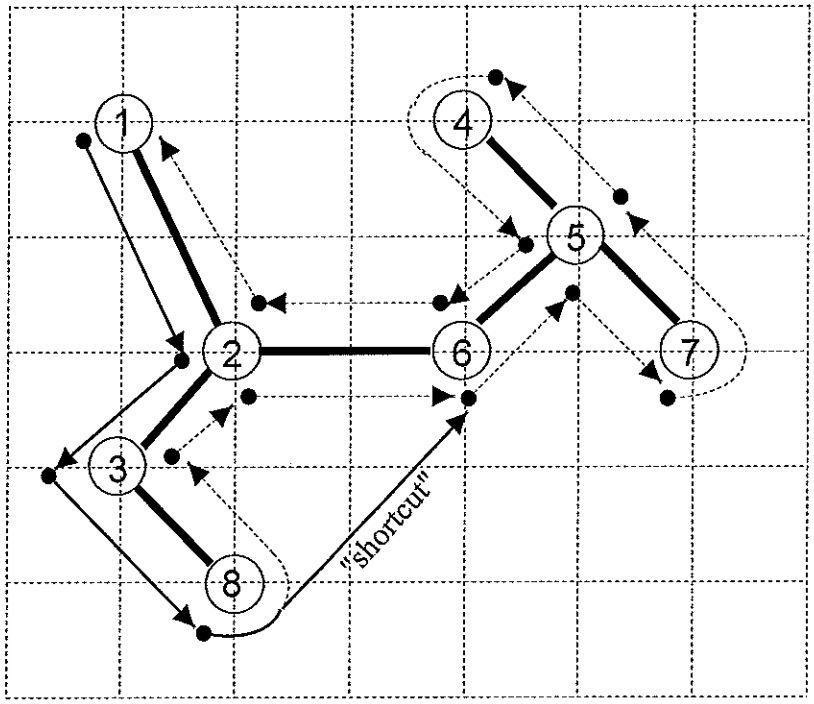
a) Select a vertex at random to be in the MST.

b) Until all the vertices are in the MST:
   - Find the closest vertex not in the MST, i.e., vertex closest to any vertex in the MST

Add this vertex using this edge to the MST

2) (Step 2) Determine the preorder walk of the MST.

3) (Step 3) Complete a tour by removing vertices from the path in step 2 by taking shortcuts.



a) Finish removing vertices from the preorder-walk path to create a tour by taking shortcuts:
[1 2 3 8 ~~3~~ ~~2~~ 6 5 7 5 4 5 6 2 1]

b) When scanning the above path, how did you know which vertices to eliminate to take a shortcut?

4) Let's determine how close our approximation algorithm gets to the actual TSP tour.

a) If we take the optimal TSP tour and remove an edge, what do we have?

b) What is the relationship between the distance of the MST and the optimal TSP tour?

c) What is the relationship between the distance of the MST and the distance of the preorder-walk of the MST?

d) What is the relationship between the distance of the preorder-walk of the MST and the tour obtained from the preorder-walk of the MST?

e) What is the relationship between the tour obtained from the preorder-walk of the MST and the optimal TSP tour?

The Final exam is Tuesday December 11th from 8:00 - 9:50 AM in ITT 328. It will be closed-book and notes, except for **three** 8" x 11" sheets of paper containing any notes that you want. (Plus, the Python Summary Handout) About 75% of the test will cover the following topics (and maybe more) since the second mid-term test, and the remaining 25% will be comprehensive (mostly big-oh analysis and general questions about stacks, queues, priority queues/heaps, lists, and recursion).

Chapter 6: Trees

Terminology: node, edge, root, child, parent, siblings, leaf, interior node, branch, descendant, ancestor, path, path length, depth/level, height, subtree
General and binary tree recursive definitions
Tree shapes and their heights: full binary tree, balanced binary tree, complete binary tree
Applications: parse tree, heaps, binary search trees, expression trees
Traversals: inorder, preorder, postorder
Binary search tree ADT: interface, implementation, big-oh of operations
Balanced binary search trees: AVL tree ADT: interface, implementation, big-oh of operations

File Structures - Lecture 24 handout:

http://www.cs.uni.edu/~fienup/cs1520f18/lectures/lec24_questions.pdf

We talked about how the in memory data structures need to be adapted for slow disks.
From this discussion you should understand the general concepts of Magnetic disks:

- layout (surfaces, tracks/cylinders, sectors, R/W heads)
- access time components (seek time - moving the R/W heads over the correct track, rotational delay - disk spins to R/W head, data transfer time - reading/writing of sector as it spins under the R/W head)

Hash Table as a useful file structure
B+ trees as a useful file structure - see web resources:

http://www.sci.unich.it/~acciaro/bpiutrees.pdf

http://en.wikipedia.org/wiki/B%2B_tree

http://www.ceng.metu.edu.tr/~karagoz/ceng302/302-B+tree-ind-hash.pdf

Chapter 7: Graphs

Terminology: vertex/vertices, edge, path, cycle, directed graph, undirected graph
Graph implementations: adjacency matrix and adjacency list
Graph traversals/searches: Depth-First Search (DFS) and Breadth-First Search (BFS)
General Idea of the following algorithms: topological sort, Dijkstra's algorithm (single-source, shortest path), Prim's algorithm (determines the minimum-spanning tree), TSP (Traveling-Saleperson Problem)
Approximation algorithm to solve TSP, general idea of backtracking and best-first search branch-and-bound.
You should understand the graph implementations and algorithms listed above. You should be able to trace the algorithms on a given graph.