The Final exam is Tuesday May 7th 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/cs1520s19/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:

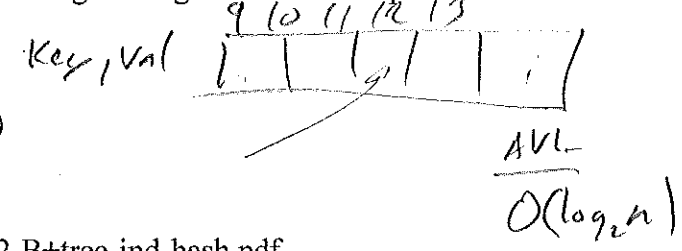
- layout (surfaces, tracks/cylinders, sectors, R/W heads)
- access time components (seek time - moving the R/W heads over the correct track, rotational delay - disk spins to R/W head, data transfer time - reading/writing of sector as it spins under the R/W head)

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



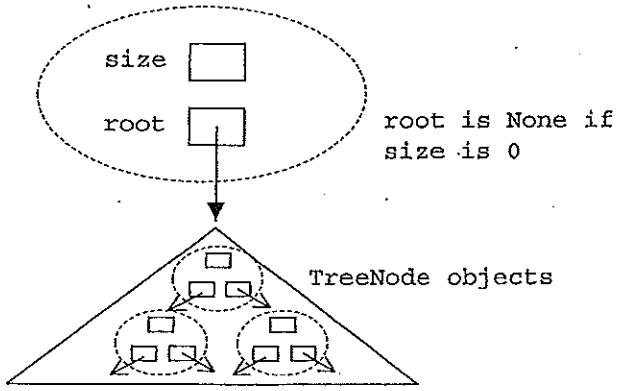
Chapter 7: Graphs

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.

1. Consider the partial TreeNode class and partial BinarySearchTree class.

```
class TreeNode:
   def __init__(self,key,val,left=None,right=None,
                                          parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and \
               self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and \
               self.parent.leftChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
      return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self

    def __iter__(self):
       if self:
          if self.hasLeftChild():
               for elem in self.leftChild:
                  yield elem
          yield self.key
          if self.hasRightChild():
               for elem in self.rightChild:
                  yield elem
```

A BinarySearchTree object




```
class BinarySearchTree:
  def __init__(self):
      self.root = None
      self.size = 0

  def length(self):
      return self.size

  def __len__(self):
      return self.size

  def __iter__(self):
      return self.root.__iter__()

  def __str__(self):
   """Returns a string representation of the tree
      rotated 90 degrees counter-clockwise"""

     def strHelper(root, level):
        resultStr = ""
        if root:
           resultStr += strHelper(root.rightChild,
                                  level+1)
           resultStr += "|  " * level
           resultStr += str(root.key) + "\n"
           resultStr += strHelper(root.leftChild,
                                  level+1)
        return resultStr

     return strHelper(self.root, 0)
```

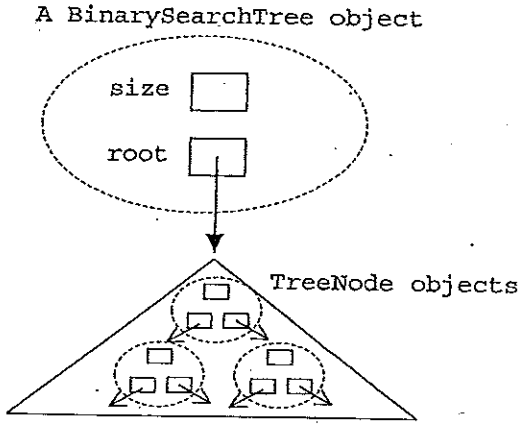
a) How do the BinarySearchTree __iter__ and __str__ methods work?

More partial TreeNode class and partial BinarySearchTree class.

```
class BinarySearchTree:
   . . .
    def __contains__(self,key):
        if self._get(key,self.root):
            return True
        else:
            return False

    def get(self,key):
        if self.root:
            res = self._get(key,self.root)
            if res:
                return res.payload
            else:
                return None
        else:
            return None

    def _get(self,key,currentNode):
        if not currentNode:
            return None
        elif currentNode.key == key:
            return currentNode
        elif key < currentNode.key:
            return self._get(key,currentNode.leftChild)
        else:
            return self._get(key,currentNode.rightChild)

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,k,v):
        self.put(k,v)

    def put(self,key,val):
        if self.root:
            self._put(key,val,self.root)
        else:
            self.root = TreeNode(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 = TreeNode(key,val,parent=currentNode)

        elif key > currentNode.key

        else: # key =
            currentNode.payload = val
            self.size -= 1
```

currentNode

leftChild



b) The `_get` method is the "work horse" of BST search. It recursively walks `currentNode` down the tree until it finds `key` or becomes `None`
In English, what are the base and recursive cases?

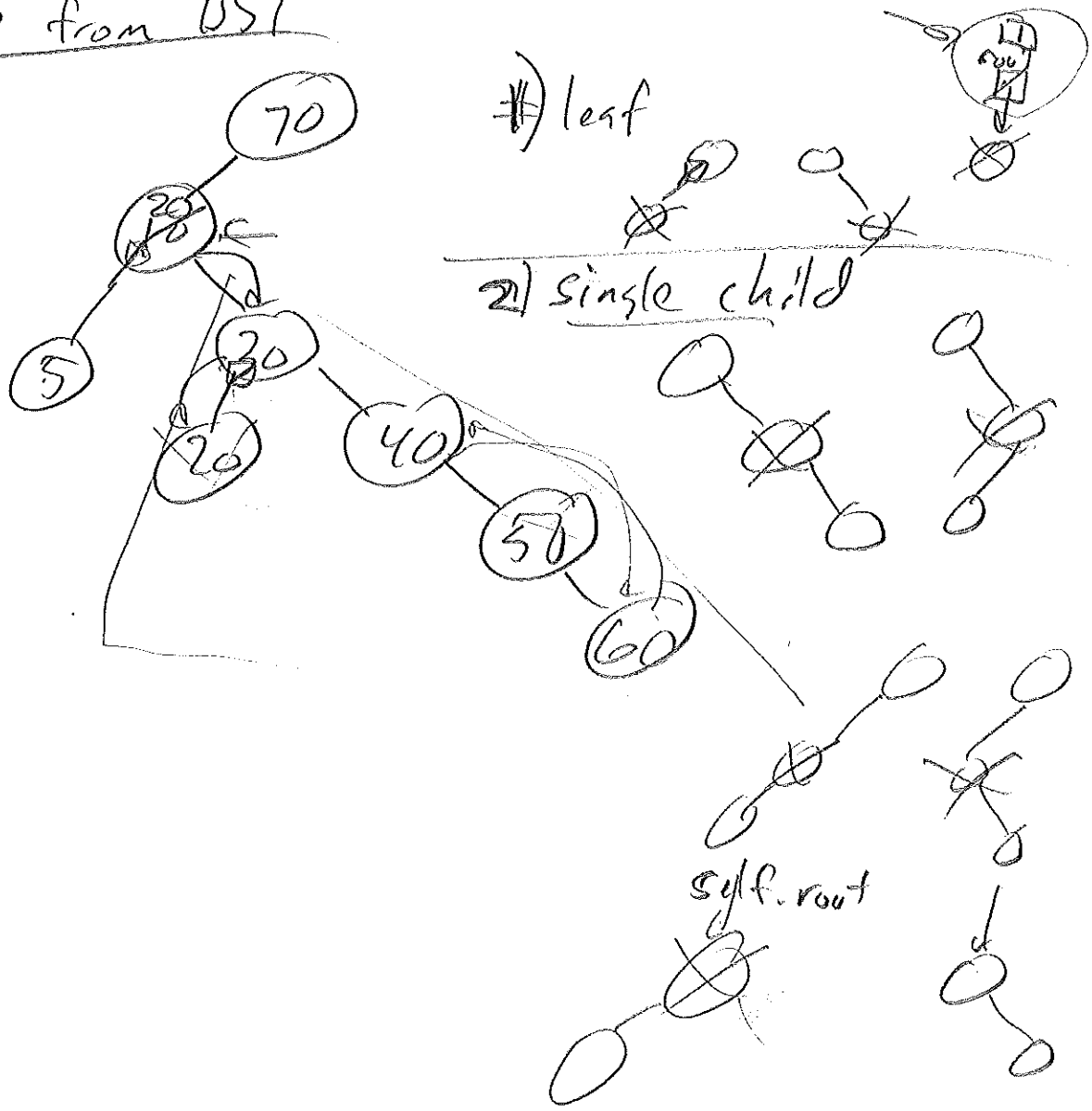
c) What is the `put` method doing?

d) Complete the recursive `_put` method.

e) Draw the "shape" of the BST after `put`s of:
50, 60, 30, 70, 90, 40, 65

f) If "n" items are in the BST, what is `put`'s: Best-case $O(\ \ )$? Worst-case $O(\ \ )$? Average-case $O(\ \ )$?

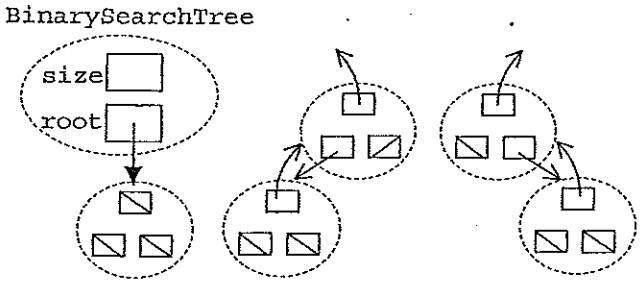
delete from BST



2. More partial TreeNode class and partial BinarySearchTree class.

```
class BinarySearchTree:
   . . .
 def delete(self,key):
   if self.size > 1:
       nodeToRemove = self._get(key,self.root)
       if nodeToRemove:
           self.remove(nodeToRemove)
           self.size = self.size-1
       else:
           raise KeyError('Error, key not in tree')
   elif self.size == 1 and self.root.key == key:
       self.root = None
       self.size = self.size - 1
   else:
       raise KeyError('Error, key not in tree')

 def __delitem__(self,key):
     self.delete(key)


 def remove(self,currentNode):
   if currentNode.isLeaf(): #leaf
     if currentNode == currentNode.parent.leftChild:
         currentNode.parent.leftChild = None
     else:
         currentNode.parent.rightChild = None
   elif currentNode.hasBothChildren(): #interior
     succ = currentNode.findSuccessor()
     succ.spliceOut()
     currentNode.key = succ.key
     currentNode.payload = succ.payload

   else: # this node has one child
     if currentNode.hasLeftChild():
       if currentNode.isLeftChild():
           currentNode.leftChild.parent = currentNode.parent
           currentNode.parent.leftChild = currentNode.leftChild
       elif currentNode.isRightChild():
           currentNode.leftChild.parent = currentNode.parent
           currentNode.parent.rightChild = currentNode.leftChild
       else:
           currentNode.replaceNodeData(currentNode.leftChild.key,
                               currentNode.leftChild.payload,
                               currentNode.leftChild.leftChild,
                               currentNode.leftChild.rightChild)

     else:
       if currentNode.isLeftChild():
           currentNode.rightChild.parent = currentNode.parent
           currentNode.parent.leftChild = currentNode.rightChild
       elif currentNode.isRightChild():
           currentNode.rightChild.parent = currentNode.parent
           currentNode.parent.rightChild = currentNode.rightChild
       else:
           currentNode.replaceNodeData(currentNode.rightChild.key,
                               currentNode.rightChild.payload,
                               currentNode.rightChild.leftChild,
                               currentNode.rightChild.rightChild)
```

a) Update picture where we delete a leaf.



b) Where in the code is each handled?

c) Draw all pictures deleting all nodes with one child.

3. Yet even more partial TreeNode class and partial BinarySearchTree class.

```
class TreeNode:
   ...
      def findSuccessor(self):
         succ = None
         if self.hasRightChild():
             succ = self.rightChild.findMin()
         else:
             if self.parent:
                 if self.isLeftChild():
                     succ = self.parent
                 else:
                     self.parent.rightChild = None
                     succ = self.parent.findSuccessor()
                     self.parent.rightChild = self
         return succ

      def findMin(self):
         current = self
         while current.hasLeftChild():
             current = current.leftChild
         return current

      def spliceOut(self):
         if self.isLeaf():
            if self.isLeftChild():
               self.parent.leftChild = None
            else:
               self.parent.rightChild = None
         elif self.hasAnyChildren():
            if self.hasLeftChild():
               if self.isLeftChild():
                  self.parent.leftChild = self.leftChild
               else:
                  self.parent.rightChild = self.leftChild
               self.leftChild.parent = self.parent
            else:
               if self.isLeftChild():
                  self.parent.leftChild = self.rightChild
               else:
                  self.parent.rightChild = self.rightChild
               self.rightChild.parent = self.parent
```

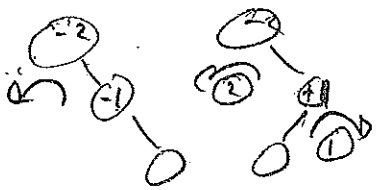
Consider the AVLTreeNode class that inherits and extends the TreeNode class to include balance factors.

```
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)
```

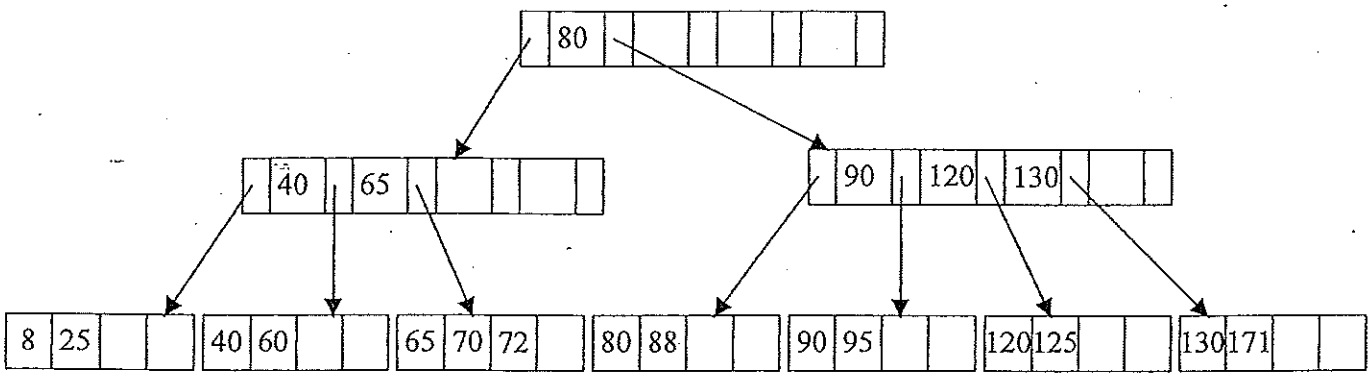
```
    def rotateLeft(self,rotRoot):                    ## NOTE:  You will complete rotateRight in Lab
        newRoot = rotRoot.rightChild
        rotRoot.rightChild = newRoot.leftChild
        if newRoot.leftChild != None:
            newRoot.leftChild.parent = rotRoot
        newRoot.parent = rotRoot.parent
        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)
```



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 $\lceil b/2 \rceil \leq \#\text{ child} \leq b$, except the root which must have at least two children.

Consider an B+ tree example with b = 5.

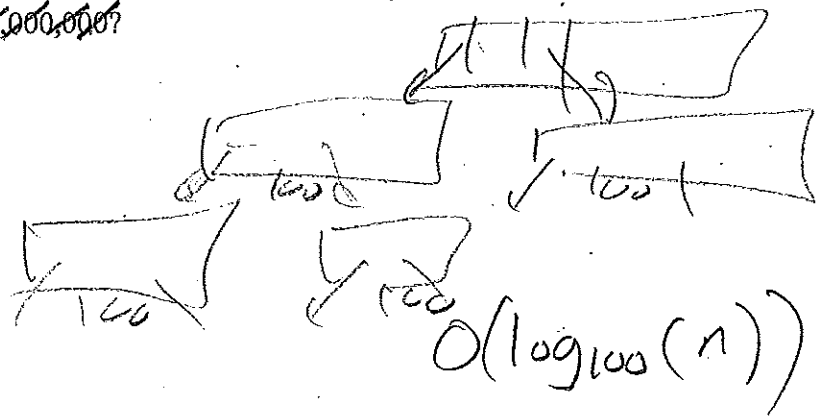


a) How would you find 88?

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?

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?



2

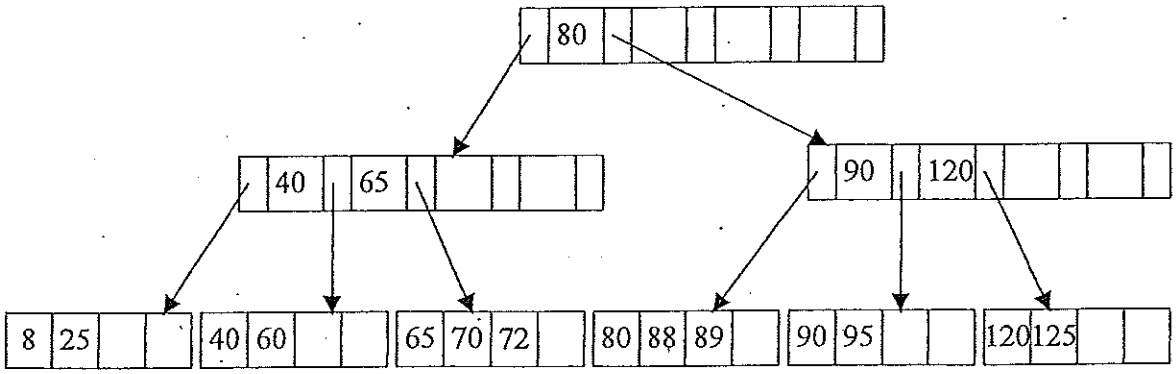
2X100

2X100X100

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?

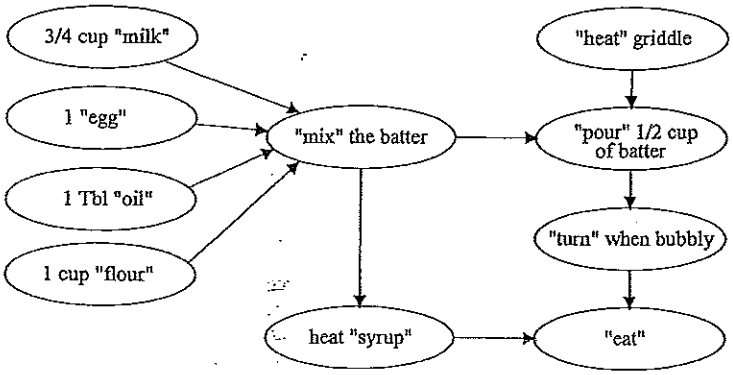


```
""" File:  vertex.py """
class Vertex:
    def __init__(self, key, color = 'white',
                 dist = 0, pred = None):
        self.id = key
        self.connectedTo = {}
        self.color = color
        self.predecessor = pred
        self.distance = dist
        self.discovery = 0
        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.