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/~acciario/bpiutrees.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.
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 \( b = 5 \).

\[
\begin{array}{cccc}
  8 & 25 & 40 & 65 \\
  80 & 120 & 90 & 120 \\
  70 & 72 & 90 & 95 \\
  80 & 88 & 120 & 125 \\
  130 & 171 & 180 & 210 \\
  180 & 200 & 210 \\
\end{array}
\]

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>Place record in sorted position in the appropriate data page.</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>1. Split data page with records &lt; middle key going in left data page and records ≥ middle key going in right data page.</td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>1. Split data page with records &lt; middle key going in left data page and records ≥ middle key going in right data page.</td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
</tbody>
</table>
Add: 50 90 30 40 35 70 80 60

BST:

30
|
40
|
35
|
60
|
50
|
90
|
70
|
80
|
60
|
3
|
1
|
2

inorder: 30 35 40 50 60 70 80 90
preorder: 50 30 40 35 90 70 60 80
postorder: 35 40 30 60 80 70 90 50

height

return max(height leftsubtree, height rightsubtree) + 1
AVL

The balance factor $= \text{height left subtree} - \text{height right subtree}$

$-1, 0, 1$

AVL

50 90 30 40 35 70 80 60

AVL

pivot

$50$

$30$

$40$

$35$

$70$

$80$

$60$

pivot

$50$

$30$

$35$

$90$

$90$

$50$

$50$

$35$

$60$

$60$

$35$

$90$

$90$

$50$
class Vertex:
    def __init__(self, key, color = 'white',
                 id = 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

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')