Data Structures with Professor Sesh Venugopal

Chapter 3 - Efficiency

Section 2 - Basic Operations

- Generally, people talk about programs in terms of clock time.
- It is more helpful to talk about running time independent of CPU speed and system load
- Every program can be reduced into a basic operation, a comparison, a parse, or a function
- It is also helpful to think of every operation as an algorithm independent of language

Section 3 - Input Size

- If a two programs perform the same task basic operation is comparison, and yet for the same input one compares more than the other, one is faster than the other.
- Speed as a function of input

Section 5 - Order and Big Oh

In order from fastest to slowest:

1. Constant: $O(1)$ As denoted by the lack of n, this is independent of input size. It is always the same.
2. Linear: $O(n)$ The running time exactly as the input size grows. Very common.
3. Quadratic: $O(n\\^2)$ Often sorting is quadratic.
4. *En log en*: $O(n \log n)$
5. Logarithmic: $O(\log n)$
6. Cubic: $O(n\\^3)$
7. Exponential: $O(k\\^n)$
8. Brute force

Chapter 4

Section 4.5 - Linked Lists

- A linked list overcomes a limitation of arrays in that additional space can be made on demand.
- A linked list is a linear structure consisting of nodes. Nodes have data and a pointer, which point to other nodes.
- The links can be thought of as "together", but are in actuality disparate.

Insertion

- Adding to the beginning:

1. Create a new node;
2. Make the new node's next link refer to the first entry;
3. Make the reference to the node of list refer to the new node.

- Adding in between, at some item *P*

1. Create a new node
2. Make new node's link refer to the node following *P*, compare through enumeration
3. Update the next link to which *P* refers to the new node

Deletion

- Arbitrary and last node: set current.next to old.next

- First node: front = front.next

Access

- Also known as traversing
- While next does not equal null, set current to next.

Chapter 7 - Stack

Section 1 - Properties

- Last in, first out. As in, the latest thing that went in will be the first thing to come off.
- A stack is a linear collection of entries in which for every entry *y* that enters the stack after another entry *x*, *y* leaves the stack before *x*.
- Operations:
- Push adds an entry to the top.
  - Pop deletes the entry from the top.
  - Enumerate lists all entries.
  - isEmpty returns if it has at least one entry
  - Size returns the number of entries

Section 2.2 - Postfix

- *a + b * (c - f) / d* is an example of infix, where *a b c f - * d/ +* is the corresponding postfix
- A general postfix expression is best defined by describing how it is evaluated using a stack. (!!!)
- When an operator is encountered, there must exist a most recent pair of operands or temporary results on the stack for application.

Chapter 10 - Comparison Trees

- A comparison tree or binary search on an array is a binary tree that depicts all possible search paths. It shows all the different sequences of comparison undertaken by binary search when searching for keys that may or may not be present in the array.
- Failure nodes are represented underneath the "leaves" of the tree, and represent the comparison that occurs when the program is searching through and discovers the end of the tree.
- There are the best case scenarios and worst case scenarios.
- There are the success scenarios and the failure scenarios.
- There are then probabilities and average cases.
- Synthesizing these, there exists a calculation for:
  - Best case for success
  - Worst case for success (*2h - 1*)
  - Best case for failure
  - Worst case for failure (*2h*)

Section 3 - Binary Search Tree Operations

- Search
  - Searching in a binary search tree is identical to tracing.
    - Each node has a basic operation of equality test, and each branch has a basic operation of less-than/greater-than performed on it.
      - If it is equal, the search terminates successfully.
      - If not, the search continues on the left or right branches.
- Insert
  - In order to insert a value, the search process is employed to *force a failure*, and the new value is inserted at the place where the

search failed.
    - A newly inserted node always becomes a leaf node in the search tree.
- Delete
    - X is a leaf node
        - This is the simplest case.
        - Simply delete X, detach this node from its parent.
    - X has one child
        - Make X's parent point to X's child.
    - X has two children
        - Switch X with the right most leaf of X's right child.

Videos - Binary Search Tree

Video 1

- Description of algorithm for binary search in an array:
    - Make high variable equal to number of elements minus one or end of relevant search.
    - Make low variable equal to zero or start of relevant search.
    - Make mid equal to high plus low divided by two in integer division.
    - Compare the item at mid to the sought after item.
    - When high equals low, the loop quits, the final call.

Video 2

- I can use a binary search comparison tree to find the number of comparisons for successful and failure on arrays of specific length.
- I am going to learn how to find the average for success and failure.
- The number of comparisons for failure is always one more than the number for success.

Average case success

- For an array of length seven, where k = 7, and it is flat and even, failure is one more than success.
- For successes, you first locate all places where a match could occur, success nodes.
- Get comparisons for the positions, and add them up.
- Divide this by the number of success positions. Add up all possible cases
- (Sum of all possible comparisons divided by the number of elements)

Average case failure

This is tricky because a failure can catch multiple failures, range matters. You need a domain. Without a domain, it is indeterminate, but it will be between a set of values.

1. Multiply the number of failure values for a node with the number of comparisons to get to that node. Add up all these values for all nodes.
2. Divide by the total number of failure values. (Subtract number of elements from extent of array)

- Evidently, you never need to compute the average for failure for any tree. All failures are at the last level.

Probabilistic search

- Rethink the average as a sum of products.
- Multiply the number of cases which each node covers by the probability of "landing" on it.
- This is akin to "distributing" the division at the end of average cases

Video 3 - Worst case for arbitrary length n and big o

- Number of nodes at a level *k* increases as 2\\^*k,* geometric series, relies on "ideal structure"

- A tree of height *k* has *k+1* levels and 2\\^*k* nodes.
- Every descent uses 2 more comparisons.
- The formula for an ideal tree is *2*log_2(n + 1) - 1*
- To make it apply generally, apply ceiling to the logarithm.
- The worst case for failure is just one more than the one for success.

Midterm

1. Sorted Linked Lists Difference (20 pts, 15+5) Suppose you are given two sorted lists of integers. The difference of the first list with respect to the second is the set of all entries that are in the first list but not in the second. For instance, given these two sorted lists:
   L1: 3 --> 5 --> 9 --> 15 --> 19 --> 25
   L2: 5 --> 8 --> 15 --> 18
   The difference (L1 minus L2) is:
   L1: 3 --> 9 --> 19 --> 25
   Complete the following method to implement the difference of two sorted lists. Assume neither list has any duplicate items. You can implement helper methods if needed, and you have the option of using recursion.

```
 public LLNode difference(LLNode list1, LLNode list2) {
      LLNode a = list1;
      LLNode c = new LLNode(0, null);
      LLNode d = c;
                  while (a != null) {
                       LLNode b = list2;
                      boolean shouldAdd = true;
                      while (b != null) {
               if (a.info == b.info) {
                     shouldAdd = false;
                             }
              b = b.next;
                        }
          if (shouldAdd) {
     c.info = a.info;
     if (a.next != null) {
          c.next = newLLNode(0, null);
          c = c.next;
     }
}
a = a.next;
      }
     return d;
 }
```

   (b) What is the worst case running time of your implementation, if the length of the two lists are m and n. Identify the basic unit time basic operations, count the number of times they are done, total the counts, and convert to big O. You will not get any credit without an adequate derivation, even if your answer is correct.
   - The worst case running time of my implementation is o(m*n).
   - This is because the program loops through every element in list 1 by the number of elements in list 2 times.
   - The basic unit operations are the comparison of the elements in the first list to the elements in the second.
   - There is no information about the number of elements this code is likely to get. The worst case, though, is that m = n, because this would result in the most possible comparisons.
   - This results in a Big-O of O(n\\^2)
2. Sorted Array Insertion (20 pts, 15+5)
   - Implement the fastest possible algorithm to insert a new entry into a sorted (in ascending order) array of items. Duplicates are NOT allowed (throw an exception if a duplicate is attempted to be inserted.) After insertion, the array should still be in sorted order. You will get at most half the credit if your algorithm is not the fastest possible. (Fastest here refers to the real clock time,

not big O).

```
public static void sortedInsert(int[] A, int n, int item) {
         int imax = n - 1;
         int imin = 0;
         while (imax \>= imin) {
         int imid = (imin + imax) / 2;
        if (A[imid] \<  item) {
              imin = imid + 1;
         } elseif (A[imid] \> item) {
             imax = imid - 1;
        } if (imax == imin){
             for (int j = n; j \> imid; j--) {
                 A[j + 1] = A[j];
             }
            A[imid] = item;
            return;
          }
  }
  return;
}\
```

- What is the worst case big O running time for your implementation? Identify the basic operations and show how they add up to the running time. (For any of the search algorithms done in class, you may assume its known running time without derivaton.) You will not get any credit without an adequate derivation, even if your answer is correct.

3. BST Ranking (17 pts, 10+7) You are given a Binary Search Tree (BST) with nodes defined as follows:

```
 public class BSTNode\<T extends Comparable\<T\>\> { T data;
    BSTNode\<T\> left, right;
... }
```

- Describe an algorithm to find the k-th smallest item in the BST. (k=1 means smallest, k=2 means second smallest, etc.) Your description must be precise enough to be able to translate into Java code. You can use additional data structure(s) if you like, but you may NOT change the BSTNode class itself. Credit depends both on correctness (50%) and efficiency (50%).

AVL Tree

Lecture

- An AVL tree is liked a BST but it's "balanced." Thus, more efficient.
- AVL tree is a BST I. Which at every node X, the heights of the sub trees of x differ at most by 1.
- Height is the number of branches from root to the farthest leaf.
- Height of empty tree is -1, single node is zero.
- Balance factor at every node: '-' (equal high), '/' (left high), or '\' (right high)
- Adelson-Velski and Landis
- Debalance at X

Problem Set 6

1. Each node of a BST can be filled with a height value, which is the height of the subtree rooted at that node. The height of a node is the maximum of the height of its children, plus one. The height of an empty tree is -1. Here's an example, with the value in parentheses indicating the height of the corresponding node:

```
        P(3)
      /   \
     M(1)  V(2)
    /     /  \
  A(0)  R(1) X(0)
         \
          S(0)
```

Complete the following recursive method to fill each node of a BST with its height value.

```
public class BSTNode<T extends Comparable> {
    T data;
    BSTNode left, right;
    int height;
    ...
}

// Recursively fills height values at all nodes of a binary tree
public static <T extends Comparable>
void fillHeights(BSTNode root) {
    if (root == null)
                return;
            height = -1;
            if (root.left != null)
                height = root.left.height;
            if (root.right != null)
                height = Math.max(root.height, root.right.height);
            height++;
}
```

2. In the AVL tree shown below, the leaf "nodes" are actually subtrees whose heights are marked in parentheses:

```
             ------ D -------
            /                 \
           B                   F
       /     \             /       \
      A         C          E         G
    /   \     /   \      /   \     /   \
  T1    T2   T3   T4    T5   T6   T7   T8
(h-1) (h) (h-1) (h-1) (h+1) (h) (h)   (h)
```

   Mark the heights of the subtrees at every node in the tree. What is the height of the tree? Mark the balance factor of each node.
3. Starting with an empty AVL tree, the following sequence of keys are inserted one at a time: 1, 2, 5, 3, 4 Assume that the tree allows the insertion of duplicate keys. What is the total units of work performed to get to the final AVL tree, counting only key-to-key comparisons and pointer assignments? Assume each comparison is a unit of work and each pointer assignment is a unit of work. (Do not count pointer assignments used in traversing the tree. Count only assignments used in changing the tree structure.)

Huffman Coding

Lecture

- Each character is represented in some binary format, they are encoded.
  - For every character, there is a character encoding scheme.
  - The present standard is UTF-8
    - This is part of the Unicode program.
    - UTF-8 defines defines greater than a million chars in 1 to 4 bytes.

        - It is of variable length, then.
    - The old way is ASCII
        - Each character in ASCII has 7 bits
        - This means that you could have 2\\^7 different characters, 128.
        - Being as the smallest unit of memory is the byte, this left an extra bit to play with.
        - If you don't mind your program not being internationalized, you can still use ASCII and keep everything smaller.
- More frequent characters get smaller code.
- Variable length encoding is Huffman coding.
- Given a set of characters, Huffman came up with an algorithm to determine the encoding needed.
- With ASCII, the average is always 8.
- An attempt at drawing this:
    - **Queue** `S:` `a w u e t $`
    - **Queue** `T:`
        - Failed.
- After a Huffman tree is built, to create the encoding, for every character you go left for, add a zero. Every time you go right, assign a one.

Problem Set 7

1. Answer the following questions in terms of h, the height of a binary tree:
    - What is the minimum possible number of nodes in a binary tree of height h?
        - The case is where it's a stalk, and the height is `h + 1`
    - A strictly binary tree is one in which every node has either no children or two children; in other words, there is no node that has exactly one child. What is the minimum possible number of nodes in a strictly binary tree of height h?
    - A complete binary tree is one in which every level but the last has the maximum number of nodes possible at that level; the last level may have any number of nodes. What is the minimum possible number of nodes in a complete binary tree of height h?
2. Two binary trees are *isomorphic* if they have the same shape (i.e. they have identical structures.) Implement the following recursive method:
3. The *radix tree* data structure shown below stores the bit strings `0,1,01,010,100,` and `1011` in such a way that each left branch represents a 0 and each right branch represents a 1.
4. You are given the following preorder and inorder traversals of a (boolean) expression tree:
    - Preorder traversal:

      ```
      || && || a > b c < d e && == != c a f < p q
      ```

    - Inorder traversal:

      ```
      a || b > c && d < e || c != a == f && p < q
      ```

    Build the expression tree from these traverals. (The first || symbol in the preorder traversal matches with the later || in the inorder.) Assume the traversals are stored in string arrays. Assume a recursive build tree method that takes five parameters: the two arrays, the preorder index, the inorder low index and the inorder high index (current inorder subarray limits).
    In your tree building process, specify, for every subtree, the index into the preorder traveral array and the index limits of the inorder array.

    ```
    a || b > c && d < e
    c != a == f && p < q
    ```

5. Exercise 9.4, page 295 of the textbook. Build a Huffman tree for the following set of characters, given their frequencies:

    ```
    R  C  L  B  H  A  E
    6  6  6 10 15 20 37
    ```

Using this Huffman tree, encode the following text:

```
CLEARHEARBARE
```

What is the average code length? If it takes 7 bits to represent a character without encoding, then for the above text, what is the ratio of the encoded length to the unencoded? Decode the following (the string has been broken up into 7-bit chunks for readability): 1111011 1010111 1101110 0010011 111000

Problem Set 8 - Hash table

1. You are given the following keys to be hashed into a hash table of size 11: `{96, 43, 72, 68, 63, 28}`. Assume the following hash function is used: `H(key) = key mod 11`, and chaining (array of linked lists) is used to resolve collisions.
   1. Show the hash table that results after all the keys are inserted.
      - `96 % 11 = 8`
      - `43 % 11 = 10`
      - `72 % 11 = 6`
      - `68 % 11 = 2`
      - `63 % 11 = 8`
      - `28 % 11 = 6`

      ```
       1:
       2: 68
       3:
       4:
       5:
       6: 28 -> 72
       7:
       8: 63 -> 96
       9:
      10: 43
      11:
      ```
   2. Compute the average number of comparisons for successful search.
      - Access should be `O(1)`. Two elements require iterating through a linked list, each with two elements. Two elements are found instantly. So, half the time, it will be `O(1)`, and then the rest of the time, it will take two comparisons, which is still `O(1)`.
2. Using chaining to resolve collisions, give the worst-case running time (big O) for inserting *n* keys into an initially empty hash table for each of the following kinds of chains:
   - **Chain is an unordered list**: If the list is simply unordered, you should be able to just put it on the front and call it a day. `O(1)`
     - The worst case is where all the elements are hashed into the same list. So `O(n)`.
   - **Chain is an ordered list**: Assuming this is a linked list, insertion would be constant time plus whatever it takes to add this element, which would depend on the length of the linked list. `O(n)`
     - `O(n^2)`. The worst case is when all the items are in one spot.
   - **Chain is an AVL tree**: Insertion into an AVL tree take that amount of time, and that's the logic I've used throughout. `O(log(n))`
     - The keyword is balanced, the maximum possible difference in subtrees is one.
     - If you have `n` items, the height is `log(n)`. If you want to insert an item into the tree, you're going to have to search through the tree. For searching, the worst case is the height, `log(n)`.
     - If I know how many items there are in the tree, I can get the height of the tree.
3. Using the following class definitions:

```
class Node {
          int key;
          String value;
          Node next;
     }

 class Hashtable {
          Node[] entries;
          int numvalues;
          float max_load_factor;
          public Hashtable(float max_load_factor) { ... }
     }
```

Complete the following methods of the Hashtable class, using the hash function `h(key) = key table_size`.

```
public void insert(int key, String value) {
        int index = key % entries.length();//can use h(key)
        Node e = new Node();
        e.key = key;// Add key to node
        e.value = value; // Add value to node
        e.next = entries[index]; // Adding to unordered list
        entries[index] = e; // Flipping the spots
        numvalues++; // Accomidating for the new value
        float load_factor = (double)numvalues /entries.length;
        if (load_factor > max_load_factor)
          rehash(); // rehashing if we need
}
private void rehash() {
        Node oldEntries[] = entries;
 int oldCapacity = oldEntries.length();
 int newCapacity = 2*oldCapacity;
 entries = new Node[newCapacity];
 numvalues=0;
 for (int i = 0 ; i < oldCapacity ; i++) {
     for (Node e = oldEntries[i] ; e != null ; e = e.next) {
         insert(e.key, e.value);
     }
 }
}
```

Note: When expanding the hash table, double its size.

4. Suppose you are asked to write a program to count the frequency of occurrence of each word in a document. Describe how you would implement your program using:
    1. **A hash table to store words and their frequencies.** This would require that each word have a unique key, there could be no overloading. In which case, I would probably convert Stringstoints.**So** `here you havem`` words to and you need to hash the words and store the frequency of the words. I'd resolve collision by chaining with an unordered list or ordered list, depending on the amount of lookup that needed to be done.
    2. **An AVL tree to store words and their frequencies. For each of these implementations:** The way I would implement this is by creating a node which has a value called "frequency". Other than that, this would be typical implementation of an AVL tree. I'd likely decided to use "compareTo" to decide whether to go left or right in the tree. I think the question may indicate that you're supposed to use an AVL tree to resolve collisions. Which is cool too. That would mean that you'd have some hash function instead of a big tree, which would likely be more efficient. So you use your hash function, and then insert into the tree. If there is no node with your current word, create one. If there is a node with your current word, increment its frequency value. This is the best implementation because it would be very fast.
        1. **What would be the worst case time to populate the data structure with all the words and their frequencies?**
            - Unordered list
                - The worst case is that they're all the different words that hash to the same location. This would result in one long

unordered chain of single frequency words.
    - So you get a big chain.
    - The worst case running time would be that simply be `O(n)`, because insertion is always one unit of work, and your doing insertion `n` times for `n` keys.
  - Ordered list
    - The worst case is when all words are unique and hash to the same spot, and then you need to go all the way to the end for it to be in order.
  - AVL tree
    - Insertion into an AVL tree is `O(log(n))` for the worst case.
    - Every word needs to be unique and hash to the same location.
    - So, you're doing `m` insertions of `log(n)` efficiency where `m = n`.
    - `O(n * log(n))`
2. **What would be the worst case time to look up the frequency of a word?**
3. **What would be the worst case time to print all words and their frequencies, in alphabetical order of the words? Assume there are *n* distinct words in the document, and a total of *m* words.**

Permalink (http://www.cs.rutgers.edu/courses/112/classes/fall_2012_venugopal/assignments/prog3/prog3.html)

Programming Assignment 3 - DOM Tree

In this assignment you will implement an HTML Document Object Model (DOM) Tree.

Summary

You will write an application to build a Document Object Model (DOM) for a given HTML file, and process it with a set of given input commands that will transform the tree.

Document Object Model

The Document Object Model (DOM) is a *platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. * (This is quoted from the **W3C Document Object Model (http://www.w3.org/DOM)** specification web page.)

When you access a web page, the browser builds a DOM tree structure for the HTML content in the page. Consider the following HTML example:

```
The quick brown fox


 which is rendered by a browser as:

The *quick* brown fox
 The DOM tree for this document would look like this:

html
          |
         body
          |
        center
          |
    --------------
    |     |      |
  The     em     brown fox
          |
        quick
 (Note: there is an invisible space character after "The", and before "brown")
```

You can see the DOM tree for any HTML document by using the DOM Inspector in Firefox - you can add on **DOM (https://addons.mozilla.org/en-US/firefox/addon/6622)** to Firefox. Then, you can see the DOM tree for the currently active document via Tools --> Web Developer --> DOM Inspector. You will notice that under the root html node, in addition to the body node, there is also a head node (with nothing under it), which corresponds to the head tag. The head tag is not used in the example above, and is automatically supplied by the browser. For this assignment you can ignore the head tag.

The DOM tree helps developers build "smart" HTML pages by embedding, say, Javascript code in HTML, and making queries on the tree to extract information about the structure of the document, or transform it by changing its shape. For example, you may have a table in an HTML page, whose individual columns can be shown or hidden upon user request via Javascript code.

In the example above, you could replace the em tag with the b tag, and remove the center tag, transforming the DOM tree to this:

```
html
          |
         body
          |
    --------------
    |     |      |
  The     b      brown fox
          |
        quick
 which would be rendered as:

The **quick** brown fox
```

Bring up this sample **try.html (http://www.cs.rutgers.edu/try.html)** in your browser. This page has javascript code embedded in HTML, that replaces em with b when you click a button. It does this by changing the DOM tree representation of the page. (Compare the DOM trees using the DOM Inspector before and after the transformation.)

Restricted HTML

For this assignment, you will work with a restricted set of HTML tags. These are:

html

top level

body

second level

p

paragraph

em

emphasis (italics)

b

boldface

table

table

tr

row (within table)

td

column (within tr)

ol

ordered (numbered) list

ul

unnumbered list

li

list item (within ol or ul)

Moreover, the format of the HTML file itself will be restricted to the following:

Examples

Following are some sample HTML pages. Click on a link to see the page, and view the page source to see the underlying HTML:

Tree Structure

Since the nodes in the DOM tree have varying numbers of children, the structure is built using linked lists in which each node has three fields: the

tag (which can be an HTML tag or plain text), the **first child** (which is null if the tag is plain text), and the **sibling**, which is a pointer to the next sibling. As an example, for the following input HTML:

```
The

quick

 brown fox


 the DOM tree would look like this:

----------
    |html| ||
    ------|---
          |
          V
       ----------
       |body| ||
       ------|---
             |
             V
          ----------   -------   ----------------
          |The ||-|-&gt; |b| |-|-&gt; | brown fox|||
          ----------   ---|---   ----------------
                          |
                          V
                      -----------
                      |quick|||
                      -----------
```

Note: Tree nodes containing tags do NOT include angle brackets with the tags. So, if a node stores em, it stores the string "em", NOT the string "". **Also, closing tags ("/em") are NOT be stored in the tree.**

Implementation

Download the attached domtree_project.zip file to your computer. DO NOT unzip it. Instead, follow the instructions on the Eclipse page under the section "Importing a Zipped Project into Eclipse" to get the entire project into your Eclipse workspace.

You will see a project called DOM Tree Assignment with the following classes:

- structures.TagNode
- structures.Tree
- structures.Stack
- apps.DOM

There are also a number of sample test files directly under the project folder (see the **Examples** section that follows.)

Note: You are not required to use the included Stack class. But if you do use a stack, you MUST use this class (not your own). When we grade your program, we will keep this Stack class in the project in case you are using it.

You will implement the following methods in the Tree class:

(Note: 5 points have been added to every method to count toward 100 points instead of the previous 75 points.)

Examples of Transformations

Here are examples of applying transformations to some HTML pages, and the resulting HTML:

Observe the following rules while working on Tree.java:

- You may NOT add any import statements to the file.
- You may NOT add any fields to the Tree class.

- You may NOT modify the headers of any of the given methods.
- You may NOT delete any methods.
- You MAY add helper methods if needed, as long as you make them private.

Grading

Since every method you will implement results in building or modifying the DOM tree structure, we will grade by examing the tree structure that results at the end of each of your methods, and comparing it with the expected structure.

We will NOT be looking at the HTML printout of the tree. (It is possible to get the same printout with an incorrect tree structure.) The HTML printout is only for your convenience, but you cannot rely on it to make sure the tree has been constructed correctly--for that, you may want to use the Eclipse debugger. There is comprehensive information on how to use the debugger in Eclipse under Help -> Help Contents -> Java development user guide -> Tasks -> Running and Debugging, and Help -> Help Contents -> Java development user guide -> References -> Views -> Variables View. Essentially, you need to know how to set breakpoints and examine variable values when the program stops at a breakpoint.

Midterm 2

- array
- linked list
- array list
- stack
- queue
- sequential search
- binary search
- binary search tree
  - Worst Case number of comparisons for success: `2 * log2 (n + 1) - 1`
  - Worst Case number of comparisons for failure: `2 * log2 (n + 1)`
  - Big O: `O(log(n))`
- AVL tree
  - An AVL tree is a binary search tree in which the heights of the left and right substrees differ in height by at most 1. It can also be recursively defined as a binary search tree in which the left and right subtrees of the root are AVL subtrees that differ at most by 1.
  - Worst case Insertion, Delete, Search: `O(log(n))`
  - Average number of comparisons for failure: `~ 2 log (n)`
  - AVL tree operations
    - Rotation - Locally rearranging the structure of an AVL tree, done to rebalance an AVL tree. - Takes O(1) time. - Done once if the off-balance balance factors are pointing in the same direction, done twice if the balance factors are pointing in opposite directions
    - Insertion - Insert the new value in like a normal BST - Backtrack up the tree and change balance factors accordingly
      - If one becomes imbalanced in the direction of its balance factor, a rebalance is necessary - check two parent nodes up to see if the balance factors are the same or not - Rotate the first two nodes, then if the balance factors were different, rotate the first node with the top node, placing the first node at the top of observed AVL tree - After one rebalance is done, or one backtracks to the top of the tree, Insertion is complete!
- binary tree
  - Inorder Traversal of Tree T: First recursively traverse the Left subtree of T, then Visit the root of T, then recursively traverse the Right subtree of T. (LVR) ¯
  - Preorder Traversal of Tree T: First Visits the root of T, then recursively traverse the Left subtree of T, then recursively traverse the Right subtree of T. (VLR) ¯
  - Postorder Traversal of Tree T: First recursively traverse the Left subtree of T, then recursively traverse the Right subtree of T, then Visit the root of T. (RVL)

- Huffman coding
  1. Construct a single-node binary tree for each symbol, place them all in queue S
  2. Pick the two smallest weight trees from queue S and T
  3. Construct a new tree by creating a root and attaching the two trees as the subtrees of the root
  4. Add the tree to queue T
  5. Repeat until S is empty
- hash table
  - A hash table is a storage array.
  - Hashing is the process of storing an object in a hash table by deriving a numeric hashcode for it and mapping this hashcode to a location in the table.
  - Collision Resolution is required when a new entry hashes to a place in the table that is already occupied, and it is the process then used to determine where the entry gets inserted.
  - Hashing function: h(k) = (sum of digits of k) mod (length of hash table array)
  - Requirements are `O(1)` time, and it must distribute keys uniformly over the hash table, ideally every location in the hash table would have the same probably of being filled as any other location.
  - If a collision occurs at location i of the hash table, it simply adds the colliding entry to a linked list that is built at that location.
  - When a new entry is to be inserted at an occupied location, it is inserted in the front of the chain at that location, O(1)
  - The worst case running time of a hash table using Chaining as collision resolution is `O(n)`
  - The load factor of a hash table is the ratio of the number of entries in the table to its size. Let n be the number of entries in the hash table, and N be the size (or capacity) of the table

Chapter 11 - Heap

Heap As Priority Queue

- The role of the heap structure is to have different priorities of removal.
- Generalization of the first-in, first-out (FIFO) queue.
- The FIFO queue is a special case of a priority queue in which the priory of an entry is the time of its arrival in the queue.

Heap Properties

- A **heap** is a complete binary tree with the property that the key of the item at any node `x` is greater than or equal to the keys of the item at all the nodes in the subtree rooted at `x`.
  - A **max heap** is one which has the greatest element at the top of the tree.
  - A **min heap** is a complete binary tree with the property that the key of the item at any node `x` is less than or equal to the keys of the items at all the nodes in the subtree rooted at `x`.
    - This implies that minimum key is at the top of the heap.

Heap Operations

- **create-heap**: create an empty heap
- **heapify**: create a heap out of given array of elements
- **find-max** or **find-min**: find the maximum item of a max-heap or a minimum item of a min-heap, respectively (aka, peek)
- **delete-max** or **delete-min**: removing the root node of a max- or min-heap, respectively
- **increase-key** or **decrease-key**: updating a key within a max- or min-heap, respectively
- **insert**: adding a new key to the heap
- **merge**: joining two heaps to form a valid new heap containing all the elements of both.

Insert

- Insert a new key in a heap must ensure that after insertion, both the heap structure and the heap ordering properties are satisfied.
- First, insert the new key so that the heap structure property is satisfied, meaning that the new tree after insertion is also complete.

- Second, makes sure that the heap ordering property is satisfied by sifting up the newly inserted key.
- Steps
  1. Add the element to the bottom level of the heap.
  2. Compare the added element with its parent; if they are in the correct order, stop.
  3. If not, swap the element with its parent and return to the previous step.

SIFTING UP

- Sifting up consists of comparing the new key with its parent, and exchanging them is the parent is less than the new key.
- In the best case, no exchanged are done.
- In the worst case, exchangers may have to be done repeatedly until the new key reaches the root.

CODE

```
public void insert(int value) {
    heapSize++;
    data[heapSize - 1] = value;
    siftUp(heapSize - 1);
}

private void siftUp(int nodeIndex) {
    int parentIndex, tmp;
    if (nodeIndex != 0) {
        parentIndex = getParentIndex(nodeIndex);
        if (data[parentIndex] > data[nodeIndex]) {
            tmp = data[parentIndex];
            data[parentIndex] = data[nodeIndex];
            data[nodeIndex] = tmp;
            siftUp(parentIndex);
        }
    }
}
```

RUNTIME ANALYSIS

- Sifting up during insertion takes one comparison per level between the new key and its parent.
- The worst case is that the new key needs to be sifted to the root, resulting in `h` comparison.
- This means that the worst case takes the same amount of comparisons as the number of nodes, which results in:
- **Big *O***: `log(n)`

Delete

- The entry at the top of the heap is the one with the maximum key. Deletion removes this entry from the heap. This leaves a vacant spot at the root, and the has to be restored.
- Finding an item in a heap is an `O(n)` operation (it is not a search structure), but if you already know where it is in the heap, removing it is `O(log n)`.
- Steps:
  1. Replace the root of the heap with the last element on the last level.
  2. Compare the new root with its children; if they are in the correct order, stop.
  3. If not, swap the element with one of its children and return to the previous step.

SIFTING DOWN

- The key `k` that is extracted from the last node and written into the root is moved as far down as necessary to ensure that placing `k` in this spot will preserve the heap ordering property.
- The children of `k` are first compared with each other to determine the larger key.
  - This larger key is then compared with `k`.
  - If `k` is smaller, it is exchanged with the larger key.

CODE

```
public void removeAt(int i) {
    data[i] = data[heapSize - 1];
    heapSize--;
    if (heapSize > 0)
        siftDown(i);
}

private void siftDown(int nodeIndex) {
    int leftChildIndex, rightChildIndex, minIndex, tmp;
    leftChildIndex = getLeftChildIndex(nodeIndex);
    rightChildIndex = getRightChildIndex(nodeIndex);
    if (rightChildIndex >= heapSize) {
        if (leftChildIndex >= heapSize)
            return;
        else
            minIndex = leftChildIndex;
    } else {
        if (data[leftChildIndex] <= data[rightChildIndex])
            minIndex = leftChildIndex;
        else
            minIndex = rightChildIndex;
    }
    if (data[nodeIndex] > data[minIndex]) {
        tmp = data[minIndex];
        data[minIndex] = data[nodeIndex];
        data[nodeIndex] = tmp;
        siftDown(minIndex);
    }
}
```

RUNTIME ANALYSIS

- Sifting down key `k` during deletion invokes two comparisons per level, one between the children of `k`, and another between the larger child and `k`.
- In the worst case, `k` may be sifted all the way down to a leaf nodes, or `h` levels, for a total of `2h` comparisons.
- In terms of `n`, this amounts to `2 * log(n + 1) - 2`, resulting in a:
- **Big *O***: `log(n)`

Runtime analysis

- Let the heigh of the heap be `h`. The maximum number of nodes `N_{max}` in a binary tree of height `h` is `2^{h + 1} - 1`. In a heap, if we assume that the last level is full, we have number of nodes `n = N_{max} = 2^{h + 1} - 1` which implies `h = log(n + 1) - 1`.
  - **Height**: `log(n + 1) - 1`
  - **Nodes**: `2^{h + 1} - 1`

Summary

- A heap is a complete binary tree with the property that value of the item at any node `x` is greater than or equal to the values of the items at all the nodes in the subtree rooted at `x`.
  - This defines a max heap, to create a min head, redefine it replacing greater than with less than.
- A heap can be used a priority queue. It can also be used to sort.
- Heaps are implemented using arrays for maximum efficiency.
- A heap is not a search structure.

Problem Set 9 - Heaps

1. Given the following sequence of integers: 12, 19, 10, 4, 23, 7, 45, 8, 15
   1. Build a heap by inserting the above set, one integer at a time, in the given sequence. Show the heap after every insertion. How many comparisons in all did it take to build the heap?
      - The way that inserting into a heap works is you place your element at the leftmost bottom open spot, and then sift up (compare) until the tree is structured.
   2. Perform successive *delete* operations on the heap constructed in the previous step, until the heap is empty. Show the heap after every deletion. How many comparisons in all did it take to perform these deletions?
      - The way that deleting works in this question is you take away the top not, and compare the two children, sifting down until you have a balanced tree. The worst case is `h` comparisons each time.
2. Suppose we have a (**max**) heap that stores integers. (By contrast, in a "min" heap the key at any node is *less than or equal to* the key at its children, so the *smallest* valued key is at the top of the heap.) Then, given an integer *k*, we would like to print all the values in this heap that are greater than k. Implement the following method to do this. *H* is the array storage for the max heap, and *n* is the number of entries in the heap. Note: The challenge is to do this efficiently. Use the heap order to reduce the number of entries of the heap to be examined.

```
public void printGreater(int[] H, int n, int k) {
   this.printGreater(H[], n, k, 0);
}

private void printGreater(int[] H, int n, int k, int i) {
    if (i > n)
        return;
    if (H[i] < k)
        return;
    System.out.println(H[i]);
    this.printGreater(H[], n, k, (2*i) + 1);
    this.printGreater(H[], n, k, (2*i) + 2);
}
```

   - You're going to need a recursive algorithm that passes in both `2k + 1` and `2k + 2` with a base case of either of those values being greater than `n`, and the program quitting on that. Additionally, if the value at `2k + 1` or `2k + 2`.
3. Consider a max heap that only supports the operations **insert**, **deleteMax**, **size**, and **isEmpty**. A client of the heap wants to update the priority of an entry in the heap. Since there is no search operation, the only way to accomplish the update is this:
   - Perform successive **deleteMax** operations until the entry is extracted
   - Update the entry's priority
   - **Insert** the entry, as well as all the other deleted entries back into the heap

- What would be the worst case running time (big *O*) of this update process on a heap with *n* entries?

1. Suppose you are given two heaps, stored in arrays. Write a method to merge them into a single heap, and return this heap. The original heaps are not modified:

```
public static <T extends Comparable<T>> T[] merge(T[] heap1, T[] heap2) {
  T[] res = new int[heap1.length + heap2.length];
  for (int i=0; i < heap1.length; i++) {
      res[i] = heap1[i];
  }
  for (int i=0; i < heap2.length; i++) {
      res[i+heap1.length] = heap2[i];
  }

  // in res, sift up starting with entries copied from the second heap
  for (int s=heap1.length; s < res.length; s++) {
      int k=s;
      // sift up res[k]
      while (k > 0) {
         int p = (k-1)/2;
         if (res[k].compareTo(res[p]) > 0) { // switch
            T temp = res[k]; res[k] = res[p]; res[p] = temp;
            k = p;
         } else {
            break;
         }
      }

  }

  return res;
}
```

Programming Assignment 4 - Friendship Graph Algorithms

In this assignment, you will implement some useful algorithms that apply to friendship graphs of the Facebook kind.

Background

In this program, you will implement some useful algorithms for graphs that represent friendships, such as Facebook. A friendship graph is an undirected graph without any weights on the edges. It is a simple graph because there are no self loops (a self loop is an edge from a vertex to itself), or multiple edges (a multiple edge means more than edge between a pair of vertices).

The vertices in the graphs for this assignment represent two kinds of people: students and non-students. Each vertex will store the name of the person. If the person is a student, the name of the school will also be stored.

Here's a sample friendship graph:

```
(sam,rutgers)---(jane,rutgers)-----(bob,rutgers)   (sergei,rutgers)
                        |                 |             |
                        |                 |             |
                  (kaitlin,rutgers)   (samir)----(aparna,rutgers)
                        |                               |
                        |                               |
 (ming,penn state)----(nick,penn state)----(ricardo,penn state)
                        |
                        |
                  (heather,penn state)


                (michele,cornell)----(rachel)
                        |
                        |
    (rich,ucla)---(tom,ucla)
```

Note that the graph may not be connected, as seen in this example in which there are two "islands" or cliques that are not connected to each other by any edge. Also see that all the vertices represent students with names of schools, except for rachel and samir, who are not students.

Algorithms

1. Subgraph: Students at a school
   You want to be able to focus exclusively on students in a particular school, and all the friendships between them. To do this, you will have to extract an appropriate subgraph out of the full graph. Here is the subgraph of students at rutgers, extracted from the original sample graph:

   ```
   (sam,rutgers)---(jane,rutgers)-----(bob,rutgers)    (sergei,rutgers)
                           |                                 |
                           |                                 |
                     (kaitlin,rutgers)              (aparna,rutgers)
   ```

2. Shortest path: Intro Chain
   sam wants an intro to aparna through friends and friends of friends. There are two possible chains of intros:

   ```
   sam--jane--kaitlin--nick--ricardo--aparna
              or
     sam--jane-bob--samir--aparna
    The second chain is preferable since it is shorter.
   ```

   If sam wants to be introduced to michele through a chain of friends, he is out of luck since there is no chain that leads from sam to michele in the graph.
   Note that this algorithm does NOT have any restriction on the composition of the vertices: a vertex along the shortest chain need NOT be a student at a particular school, or even a student. So, for instance, you may have to find the shortest intro chain from nick to samir, which has the following solution:

   ```
   nick--ricardo--aparana--samir
    which consists of two penn state students, one rutgers student, and one non-student.
   ```

3. Connected Islands: Cliques
   Students tend to form cliques with their friends, which creates islands that do not connect with each other. If these cliques could be identified, particularly in the student population at a particular school, introductions could be made between people in different cliques to build larger networks of friendships at that school.
   In the sample graph, there are two island cliques for students at rutgers:

```
(sam,rutgers)---(jane,rutgers)-----(bob,rutgers)    (sergei,rutgers)
                          |                               |
                          |                               |
                    (kaitlin,rutgers)              (aparna,rutgers)
```

If we were to look at students at penn state, instead, there is a single clique:

```
(ming,penn state)----(nick,penn state)----(ricardo,penn state)
                          |
                          |
                    (heather,penn state)
```

And again, a single clique for students at ucla:

```
(rich,ucla)---(tom,ucla)
```

And one for students at cornell:

```
(michele,cornell)
```

From these examples, it should be clear that if there is at least one student in the graph that goes to a particular school, then there must be at least one island clique in the graph for students at that school.

4. Connectors: Friends who keep friends together
   If jane were to leave rutgers, sam would no longer be able to connect with anyone else--jane was the "connector" who could pull sam in to hang out with her other friends. Similarly, aparna is a connector, since without her, sergei would not be able to "reach" anyone else. (And there are more connectors in the graph...)
   On the other hand samir is not a connector. Even if he were to leave, everyone else could still "reach" whoever they could when samir was there, even though they may have to go through a longer chain.
   A precise definition of a connector in any undirected graph is a vertex, v, such that there are at least two other vertices x and w for which *every* path between x and w goes through v. For example, v=jane, x=sam, and w=bob.
   Finding all connectors in an undirected graph can be done using DFS (depth-first search), but keeping track of a couple more numbers for every vertex v. These are:
   - dfsnum(v): This is the dfs number, assigned when a vertex is visited, dealt out in increasing order.
   - back(v): This is a number that is initially assigned when a vertex is visited, and is equal to dfsnum, but can be changed later based on three conditions:
     - When the DFS backs up from a neighbor, w, to v, if dfsnum(v) > back(w), then back(v) is set to min(back(v),back(w))
     - If a neighbor, w, is already visited then back(v) is set to min(back(v),dfsnum(w))

   When the DFS backs up from a neighbor, w, to v, if dfsnum(v) ≤ back(w), then v is identified as a connector, IF v is NOT the starting point for the DFS. (If v is a starting point for DFS, it can be a connector, but another check must be made - see the examples below. The examples don't tell you how to identify such cases, that's up to you.)
   Here are some examples that show how this works.
   - Example 1: (B is a connector) A--B--C The DFS starts at A. Neighbors for a vertex are stored in REVERSE alphabetical order:

```
A: B
  B: C,A
  C: B

  dfs @ A  1/1  (dfsnum/back)
      dfs @ B 2/2
          dfs @ C 3/3
              neighbor B is already visited =&gt; C 3/2
          dfsnum(B)  B 2/1
      dfsnum(A)
```

Permalink (http://www.cs.rutgers.edu/courses/112/classes/fall_2012_venugopal/probs/ps10.html)

Chapter 14 - Graph Algorithms

Traversals

Depth-First Search for Undirected Graphs

DESCRIPTION:

- Sesh's:
  - Suppose that traversal begins at `A`
  - The algorithms visits `A` and has to chose a vertex to visit next, and `A` has neighbors `B`, `D`, `E`, and `G`. Any of these can be picked next.
  - While visiting `B`, our algorithm observes that it has neighbor `A`, which has already been visited, so it skips it. The algorithm keeps track of this by "marking" vertices.
  - Continuing this motion, this particular algorithm reaches the last element (which happens to be `G`), and finds that both of the neighbors have been visited, so this is a dead end for the algorithm.
  - When at the end, the last element, this algorithm backs out to try and visit unexamined neighbors of previously visited nodes.
- Wikipedia:
  - Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children.
  - Then the search backtracks, returning to the most recent node it hasn't finished exploring.
  - In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

ALGORITHM FOR `DFS(V)`

1. Visit `v` and mark `v` as visited
2. for each neighbor `w` of `v`
   1. if `w` is not visited, then:
      1. `DFS(w)`

Breadth-First Search for Undirected Graphs

- From book:
  - The primary characteristic of this search is that is visits the vertices following a wave like motion over the graph.
  - The traversal first visits `A`, then it visits each neighbor of `A` in turn. These visits mark the first wave.
  - The algorithm then goes to this vertex's first neighbor, and visits all of it's unvisited neighbors.
- From Wikipedia (Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.):
  1. Enqueue the root node
  2. Dequeue a node and examine it
     1. If the element sought is found in this node, quit the search and return a result.
     2. Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
  3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
  4. If the queue is not empty, repeat from Step 2.

ALGORITHM `BFS(V)`

1. visit `v` and mark `v` visited
2. add `v` to queue
3. while the queue is not empty do
   1. `w <-` vertex at the from of the queue
   2. delete `w` from the queue
   3. for each neighbor `p` of `w` do
      1. if `p` is not visited then

1. visit `p` and mark `p` as visited
2. add `p` to the queue

Topsort

Wikipedia Standard Algorithm

1. `L` ← Empty list that will contain the sorted elements
2. `S` ← Set of all nodes with no incoming edges
3. while `S` is non-empty do
    1. remove `a` node `n` from `S`
    2. insert `n` into `L`
    3. for each node `m` with an edge `e` from `n` to `m` do
        1. remove edge `e` from the graph
        2. if `m` has no other incoming edges then 1. insert `m` into `S`
4. if graph has edges then
    1. return error (graph has at least one cycle)
5. else
    1. return `L` (a topologically sorted order)

Wikipedia Depth-first Version

1. `L` ← Empty list that will contain the sorted nodes
2. `S` ← Set of all nodes with no incoming edges
3. for each node `n` in `S` do
    1. `visit(n)`
4. function `visit(node n)`
    1. if `n` has not been visited yet then
        1. mark `n` as visited
        2. for each node m with an edge from `n` to `m` do
            1. `visit(m)`
        3. add `n` to `L`

Sesh's **`DFStopsortdriver`**

1. `topnum <- n`
2. for each vertex `v` in graph do
    1. if `v` is not visited then
        1. **`DFStopsort`**`(v, topnum)`

Sesh's **`DFStopsort`**`(v, topnum)`

1. visit `v` and mark `v` as visited
2. for each neighbor `w` of `v` do
    1. if `w` is not visited then
        1. **`DFStopsort`**`(w, topnum)`
3. number `v` with `topnum`
4. `topnum <- topnum - 1`

Connected components

Sesh's DFSconndriver

1. `compnum <- 0`

2. for each vertex `v` in the graph do
    1. if `v` is not visited then
        1. `compnum <- compnum + 1`
        2. **`DFSconn`**`(v, compnum)`

Sesh's DFSconn(v, compnum)

1. visit `v` and mark `v` as visited
2. mark `v` with `compnum`
3. for each neighbor `w` of `v` do
    1. if `w` is not visited then
        1. `DFS(w, compnum)`

Running times

- Both of these algorithms scan the vertices and edges of the graph, and the only difference is the order which they are visited.
- The running time is computer by counting a unit of time for every vertex visit and a unit of time for every vertex inspection to see if it is visited.
- We assume that the graph has `e` edges and `v` vertices.
- Being as each vertex is visited exactly once, the total time for this adds `n` units of time.
- How many times are vertices inspected?
    - In general, a vertex is inspected as many times as the degree.
    - The total number of vertex inspections is there the sum of the degree of all the graph vertices.
- The total time for traversal is there for `n + 2 * e` or `O(n + e)`. (This only holds when graphs are connected.)

Summary

- Directed and undirected graphs can be used to model pair-wise relationships among entities.
- Every undirected graph is a special kind of directed graph.
- The adjacency matrix representation of a graph is useful when the graph is dense or the application makes very frequent random queries on edges, such as "is edge `(x,y)` in the graph?"
- The adjacency linked-lists representations is useful when the graph is relatively sparse, and/or the application needs to access all or most neighbors of vertices.
- Both depth-first and bread-first traversals are linear-time graph-traversal algorithms, with a running time of `O(n + e)`.
- Topological sorting is applicable only to directed or DAGs. It is a linear `((O(n + e)) algorithm that is typically used on precedence graphs.
- Dikstra's shortest-path algorithm is an instance of a greedy algorithm.
- The worst-case running time of Dijkstra's algorithm is `O(n^2)` if the fringe is a simple unordered list. This time can be improved to `O(n + e)log(N)` if the fringe is implemented using a priority queue.

Problem Set 10 - Graphs

1. Suppose a weighted undirected graph has *n* vertices and *e* edges. The weights are all integers. Assume that the space needed to store an integer is the same as the space needed to store an object reference, both equal to one unit. *What is the minimum value of e* for which the adjacency matrix representation would require less space than the adjacency linked lists representation? Ignore the space needed to store vertex labels.
    - So the storage for an adjacency matrix is always `n^2` , as every edge and vertex is represented in two-dimensional boolean values.
2. Given an **undirected** graph represented as an **adjacency matrix**, implement a method to count the number of edges in the graph. What is the worst case running time (big O) of your implementation for a graph with *n* vertices and *e* edges?

```
public class Graph {
int n;  // number of vertices
boolean[][] adjacencyMatrix;
public int numEdges() {
    int numEdges = 0;
    for (int i = 0; i < adjacencyMatrix.length; i++) {
        for (int j = 0; j < adjacencyMatrix[i].length; j++) {
            if (adjacencyMatrix[i][j]) {
                numEdges++;
            {
        }
    }
    return numEdges;
}
```

}

- Let a comparison be a unit of work and `n` be the number of vertices. Every vertex has the potential of having an edge with every other vertex, so they must all be compared against each other. This result in a big O of `O(n^2)`.

3. The complement of an **undirected** graph, **G**, is a graph **GC** such that:
   - **GC** has the same set of vertices as **G**
   - For every edge *(i,j)* in **G**, there is no edge *(i,j)* in **GC**
   - For every pair of vertices *p* and *q* in **G** for which there is no edge *(p,q)*, there is an edge *(p,q)* in **GC**.

   Implement a method that would return the complement of the **undirected** graph on which this method is applied.

```
public class Graph {
        int n;  // number of vertices
        boolean[][] aM;

        public Graph complement() {
                boolean [][] am = new boolean[n][n];
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < n; j++) {
                                if (aM[i][j]) {
                                        am[i][j] = false;
                                } else {
                                        am[i][j] = true;
                                }
                        }
                }
        }
}
```

   What is the worst case running time (big O) of your implementation for a graph with *n* vertices and *e* edges?

4. Repeat the complement exercise for an **undirected** graph that is stored in the adjacency linked lists format:

```
class Edge {
        int vnum;
        Edge next;
}

public class Graph {
        Edge[] adjlists;  // adjacency linked lists

        public Graph complement() {

        }
}
```

What would be the worst case running time (big O) of an implementation for a graph with *n* vertices and *e* edges?

5. Consider this graph: [a forkjoin]
   This graph has *n%2B2* vertices and 2*n* edges. For every vertex labeled *i*, *1 , there is an edge from *S* to *i*, and an edge from *i* to *T*.
   1. How many different depth-first search sequences are possible if the start vertex is *S*?
   2. How many different breadth-first search sequences are possible if the start vertex is *S*?

- You can use DFS to check if there is a path from one vertex to another in a directed graph. Implement the method **hasPath** in the following. Use additional class fields/helper methods as needed:
  public class Neighbor { public int vertex; public Neighbor next; ... }
  public class Graph { Neighbor[] adjLists; // adjacency linked lists for all vertices

```
// returns true if there is a path from v to w, false otherwise
public boolean hasPath(int v, int w) {
  // FILL IN THIS METHOD
  ...
}
```

}

6. You are given a directed acyclic graph. Complete the following implementation to toplogically sort the vertices using **using BFS (breadth-first search)**. Assume that the graph has already been read in. You may implement helper methods as needed. You may use the following Queue class:

```
public class Queue {
      ...
      public Queue() {...}
      public void enqueue(T item) {...}
      public T dequeue() throws NoSuchElementException {...}
      public boolean isEmpty() {...}
      ...
   }



public class Graph {
      static class Neighbor {
         int vertex;
         Neighbor next;
         ...
      }

      static class Vertex {
         String name;
         Neighbor[] neighbors; // adjacency linked lists for all vertices
      }

      Vertex[] vertices;

      // returns an array with the names of vertices in topological sequence
      public String[] topsort() {
        // FILL IN THIS METHOD
        ...
      }
      ...
   }
```

Lecture - November 27th, 2012

Dijkstra's Shortest Path Algorithm

- Question: What is the shortest path from `x` to `y` ?
- Dijkstra's algorithms is a trial and error approach.
- Steps to Dijkstra's AlgorithL:
  1. For each neighbor of the source `S`, set distance of `d(x)` to `wt(s,x)`, which is the wight of edge `s->x`. Assign to every node a tentative distance (it is zero for our initial node and to infinity for all other nodes).
  2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes except the initial node.
  3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances.
  4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again; its distance recorded now is final and minimal.
  5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal), then stop. The algorithm has finished.
  6. Set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3.

Final Review

- Graphs
  - DFS and BFS
    - Space analysis:
      1. The visited array contributed `O(n)` where `n` is the number of vertices.
    - Time analysis: The running time is computer by counting a unit of time for every vertex visit and a unit of time for every vertex inspection to see if it has been visited.
      1. We assume that the graph has `n` vertices and `e` edges.
      2. Each vertex is visited exactly once. `n` vertex visits.
      3. A vertex is visited as many times as the degree of that vertex. The sum of the vertex degrees is related to the number of edges. However, a shared edge is counted twice, resulting in a number of inspections of `2e`.
      4. `n + 2 * e` or `O(n + e)`
  - Topological Sort
    - DFS
      - Space analysis: `O(n)` because of the visited array.
      - Time analysis: `O(n + e)` because every edge is "walked" and every vertex is "hit."
  - Dijkstra's algorithm
    - Space analysis: Depends on implementation, generally, it will be `O(n)` or `O(n^2)` where `n` is the number of vertices.
    - Time analysis: Assuming a graph of `n` vertices and `e` edges.
      1. Every edge in the graph is used in at most one distance computation. The total number of distance computations is at most `e`.
      2. Starting with at most `n - 1` vertices in the fringe, the number of vertices is reduced by one after every selection. Thus, the total time for selection never exceeds: (n - 1) + (n - 2) + ... 2 + 1 = n(n - 1)/2.
      3. Therefore, the total time is `n(n - 1)/2 + e`, resulting in `O(n^2)`.
  - Adjacency matrix graph time/space analysis:
    - The adjacency matrix representation of a graph is useful when the graph is dense or the application makes very frequent random queries on edges, such as “is edge (x,y) in the graph?”
  - Adjacency linked lists graph time/space analysis:
    - The adjacency linked-lists representations is useful when the graph is relatively sparse, and/or the application needs to access all or most neighbors of vertices.
- Sorting

    - Best case big O running times of insertion sort, quicksort, and mergesort
        - Insertion: `O(n)` comparisons, `O(1)` swaps
        - Quicksort: `O(n log n)`
        - Mergesort: `O(n log n)`
    - Worst case big O running times of insertion sort, quicksort, and mergesort, heapsort
        - Insertion: `O(n^2)`
        - Quicksort: `O(n^2)`
        - Mergesort: `O(n log n)`
        - Heapsort: `O(n log n)`
    - Linear worst case running time of the build-heap algorithm with repeated sift downs.
- Other data structures (worst case big O)
    - Heap
        - Insert: The worst case is when you have to sift back up to the top, and because if the number of nodes is `n`, the height is going to be `log(n)`, so the number of sifts is `log(n)`.
        - Delete: The worst case is when you have to sift down to the bottom, because if the number of nodes is `n`, the height will be `log(n)`, so the number of sifts is `log(n)`.
    - AVL tree (assuming that the height of an is `O(log n)`)
        - Search
        - Insert
    - Hash table
        - Search: The worst case is that everything is hashed to one location and is unsorted, so you have to linearly search through the linked list, and you get the last element. Resulting in `O(n)`.
        - Insert: `O(n)` because you have to rehash in the worst case.
- Sorts
    - Insertion sort [Sec 13.1]
        - Steps:
            1. Search int eh partially sorted list to find the correct position, `p`, for `x`.
            2. Move all the entries between positions `p` and `i - 1` over to the right by one place.
            3. Write `x` into position `p`. The partially sorted list has now grown to occupy positions `0` through `i`.
        - Runtime analysis: The two operations are the comparisons between pair of entries in the list, and shifting the
    - Quicksort [Sec 13.2.1]
        - Steps:
        - Runtime analysis:
    - Mergesort [Sec 13.2.2]
        - Steps:
        - Runtime analysis:
    - Heap sort [Sec 13.3]
        - Steps:
        - Runtime analysis: