# DATA STRUCTURES NOTES FOR THE FINAL EXAM SUMMER 2002

Michael Knopf

mknopf@ufl.edu

DISCLAIMER: Mr. Michael Knopf prepared these notes. Neither the course instructor nor the teaching assistants have reviewed them for accuracy or completeness. In particular, note that the syllabus for your exam may be different from what it was when Mr. Knopf took his second exam. Use at your own risk. Please report any errors to the instructor or teaching assistants.

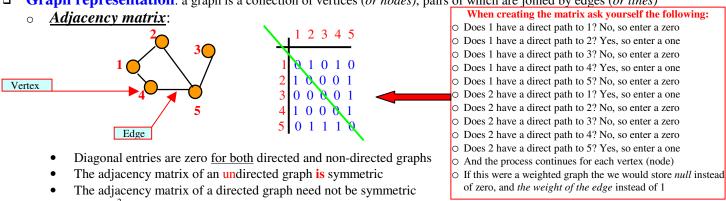
# **Graph operations and representation:**

- Path problems: Since a graph may have more than one path between two vertices, we may be interested in finding a path with a particular property. For example, find a path with the minimum length from the root to a given vertex (node)
- □ Definitions and Connectedness problems: (see page 9 for diagrams)
  - O Simple path: a path in which all vertices, except possibly the first and last, are different
  - O <u>Undirected graph</u>: a graph whose vertices do not specify a specific direction
  - o **Directed graph:** a graph whose vertices do specify a specific direction
  - o **Connected graph:** there is at least one path between **every** pair of vertices
  - o <u>Bipartite graphs</u>: graphs that have vertexes that are partitioned into 2 subsets A and B, where every edge has one endpoint in subset A and the other endpoint in subset B
  - o A complete graph: an n-vertex undirected graph with n(n-1)/2 edges is a complete graph
  - o <u>A complete digraph</u>: (denoted as  $K_n$ ) for n-vertices a complete digraph contains exactly n(n-1) directed edges
  - o <u>Incident:</u> the edge (i, j) is incident on the vertices i and j (there is a path between i and j)
  - $\circ$  **In-degree:** the in-degree d of vertex i is the # of edges incident to i (the # of edges coming into this vertex)
  - o <u>The out-degree:</u> the out-degree d of vertex i is the # of edges incident from vertex i (the # of edges leaving vertex i)
  - o The degree of a vertex: the degree d of vertex i of an undirected graph is the number of edges incident on vertex i
  - o <u>Connected component</u>: a maximal *sub-graph* that is connected, but you cannot add vertices and edges from the original graph and retain connectedness. A connected graph has EXACTLY **one** component
  - o <u>Communication network</u>: Each edge is a feasible link that can be constructed. Find the components and create a small number of feasible links so that the resulting network is connected
  - <u>Cycles</u>: the removal of an edge that is on a cycle **does not** affect the networks connectedness (a cycle creates a sort of loop between certain vertices, for **example** there is a path that links vertex a to b to c and then back to a)

## ■ Spanning problems:

- o <u>A spanning tree</u>: is a sub-graph that includes <u>all vertices</u> of the original graph <u>without cycles</u>
  - Start a breadth-first search at any vertex of the graph
  - If the graph is connected, the n-1 edges are used to get to the unvisited vertices define the spanning tree (breadth-first spanning tree)

Graph representation: a graph is a collection of vertices (or nodes), pairs of which are joined by edges (or lines)



- Needs n<sup>2</sup> bits of space, for an undirected graph we only need to store the upper or lower triangle (since they are symmetric)
- Time to find the vertex degree and/or vertices adjacent to is O(n)
- Adjacency lists: an adjacency list for vertex i is a linear list of vertices adjacent from vertex i. These are much more time efficient then an adjacency matrix.

- **Linked adjacency lists**: each adjacency list is a chain, there are 2e node in an undirected graph and e node in a directed graph (**please see page 10 for an example**)
- Array adjacency lists: each adjacency list is an array, the number of elements is 2e in an undirected graph and e in a directed graph
- $\Box$  **Graph search methods**: a vertex *u* is **reachable** from vertex *b* iff there is a path from *u* to *b* 
  - A search method starts at a given vertex v and visits/labels/marks every vertex that is reachable from v
  - o Many graph problems are solved using search methods
    - Breadth-first search: (Pseudocode and the actual code are found on page 8)
      - > Visit the start vertex and use a FIFO queue
      - > Repeatedly remove a vertex from the queue, visit its <u>unvisited</u> adjacent vertices putting the newly visited vertices into the queue, when the queue is empty the search terminates
      - > All vertices that are reachable from the start vertex (including the start vertex) are visited

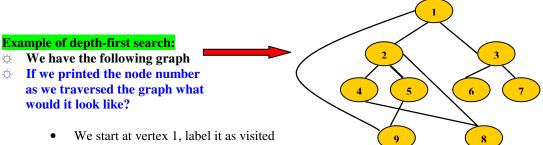
### Page 2

# • Depth-first search:

- > Has the same complexity as breadth-first search
- > Has the same properties with respect to path finding, connected components, and spanning trees
- > Edges used to reach unvisited vertices define a depth-first spanning tree when the graph is connected
- There are problems were each method performs better than the other
- > The code for depth-first search is given below:

```
public int newDFS (int theV, int theK)
       if( theK <= 0 || theK > vertices())
       {
               //no vertex to return
               return 0;
       }
       else
       {
               //initialize static variables
               count = 0;
               targetCount = theK:
               label = new boolean[vertices() + 1];  //the default value is false
               return newDFS(theV); //making the call to the static method newDFS(theV)
}
/** the private method that does the actual depth-first traversal
private static int newDFS(int v)
{
       //return vertex if found
       //reached a new vertex
       count++;
       if (count == targetCount)
       {
               return v: //we found the vertex we were looking for
       }
                  //we need to reach more vertices
       else
               label[v] = true;
               Iterator iv = iterator(v);
               while(iv.hasNext())
                       //visit an adjacent vertex of v
                       int u = ((EdgeNode.iv.next()).vertex;
                      if(!label[u])
                       {
                              //u is a new vertex
                              int w = newDFS(u);
                              if(w>0) //the target vertex has been found
                              {
                                 return w:
```

```
}
    return 0;  //the target vertex was not found
} //ends while-loop
} //ends if-else statement
} //ends method newDFS(int v)
```



Page 3

- Do a DFS on the unvisited neighbors of 1 = nodes 2, 3, 9
- Visit 2, label it as visited, and do a DFS on its unvisited neighbors = 4, 5, 8
- Visit 4, label it as visited, and do a DFS on its unvisited neighbors = 8
- Visit 8, label it as visited, and do a DFS on its unvisited neighbors = nobody
- Since 8 has no unvisited neighbors we return to its parent = 4 and visit any unvisited neighbors = nobody
- Since 4 has no unvisited neighbors we return to its parent = 2 and visit any unvisited neighbors = 5
- Visit 5, label it as visited, and do a DFS on its unvisited neighbors = 9
- Visit 9, label it as visited, and do a DFS on its unvisited neighbors = nobody
- Since 9 has no unvisited neighbors we return to its parent = 5 and visit any unvisited neighbors = nobody
- Since 5 has no unvisited neighbors we return to its parent = 2 and visit any unvisited neighbors = nobody
- Since 2 has no unvisited neighbors we return to its parent = 1 and visit any unvisited neighbors = 3
- Visit 3, label it as visited, and do a DFS on its unvisited neighbors = 6, 7
- Visit 6, label it as visited, and do a DFS on its unvisited neighbors = nobody
- Since 6 has no unvisited neighbors we return to its parent = 3 and visit any unvisited neighbors = 7
- Visit 7, label it as visited, and do a DFS on its unvisited neighbors = nobody
- We are done, the print out would appear as follows: 1, 2, 4, 8, 5, 9, 3, 6, 7
- A <u>Breadth-first-search</u> would yield the following print out: 1, 2, 3, 9, 4, 5, 8, 6, 7 (<u>please see page 9 for a detailed explanation of Breadth-first-search on this graph</u>)

### □ Algorithm design methods:

- o <u>The greedy method</u>: examples are Least Processing Time for machine scheduling, 0/1 knapsack, etc.
  - *Knapsack greedy heuristics*: select a subset with < = k items
    - $\triangleright$  If the weight of this subset is > c (capacity) then discard the subset
    - If the subsets weight it < = c, fill as much of the remaining capacity as possible by being greedy on profit density
    - $\rightarrow$  Try all subset with < = k items and select the one that yields maximum profit (this is common sense)
    - First sort into decreasing order of profit density (profit density = profit / weight)
    - There are  $O(n^k)$  subsets with at most k items
    - > Trying a subset takes O(n) time
    - The total time is  $O(n^{k+1})$  when k > 0
    - $\rightarrow$  (Best value greedy value) / (best value) < = 1 / (k + 1) so the results are inversely proportional to k

### • Shortest path problem:

- Directed weighted graph
- The path length is the sum of the edges on the path
- > The vertex at which the path begins is the source vertex
- The vertex at which the path ends is the destination vertex
- There are several <u>source-to-destination algorithms</u> which include the following:

- The greedy single source single destination algorithm:
  - o Leave the source vertex using the cheapest/shortest edge.
  - o Leave the new vertex using the cheapest edge subject to the constraint that a new vertex is reached
  - Continue until the destination is reached.

### The *greedy single source all destination* algorithm:

- o Let d(i) (distanceFromSource(i)) be the length of the shortest one edge extension of an already generated shortest path, the one edge extension ends at i.
- o The next shortest path is to an as yet un-reached vertex for which d() is the least.
- o Let p(i) (predecessor(i)) be the vertex just before vertex i on the shortest one edge extension of i
- o Terminate the algorithm as soon as the shortest path to the desired vertex has been generated.

Page 4

### • Dijkstra's algorithm:

- > The greedy single source all destination algorithm described above is known as Dijkstra's algorithm
- $\triangleright$  Implement d(i) and p(i) as a 1D array
- $\triangleright$  Keep a linear list L of reachable vertices for which the shortest path is yet to be generated.
- $\triangleright$  Select and remove the vertex v in L that has the smallest d(i) value
- Update the d(i) and p(i) values of vertices that are adjacent to v.
- $\rightarrow$  Total time  $O(n^2)$  using a linear list
- Total time O(n log n) using a min heap in place of a linear list

### • Minimum-cost spanning tree:

- > Weighted connected undirected graph
- > Spanning tree
- Cost of spanning tree is sum of edge costs
- > Find spanning tree that has minimum cost
- > Edge <u>selection\_greedy</u> strategy: (all result in a min-cost spanning tree)
  - Kruskal's method: (see page 10 for an example) Start with an n vertex forest. Consider edges in ascending order of cost; select an edge if it does not form a cycle with an already selected edge. A min-cost spanning tree is unique when all the edge costs are different. This uses union-find trees to run in O(n + e log e) time.

# Psuedo code for Kruskal's method:

```
Start with an empty set T of edges.
while (E is not empty && |T| != n-1)
{
    Let (u,v) be a least-cost edge in E.
    E = E - {(u,v)}. // Delete edge from E
    if ((u,v) does not create a cycle in T)
    Add edge (u,v) to T.
}
if (|T| == n-1) T is a min-cost spanning tree.
else Network has no spanning tree.
```

## Data structures for Kruskal's method:

- o <u>Use an array</u> (not an ArrayLinearList) to store a set T of selected edges
- Questions to ask yourself about the set *T*:
  - <u>Does *T* have n-1 edges where n is the number of nodes in the tree?</u> Check the size of the array to determine this.
  - Does the addition of edge (u, v) to *T* result in a cycle? This can be hard to determine.
  - If not then add the edge to *T* at the end of the array.
- Each component of T is defined by the vertices in the component
- $\circ$  We will represent each component as a set of vertices:  $\{1, 2, 3, 4\}$   $\{5, 6\}$   $\{7, 8\}$
- O Two vertices are in the same component iff they are in the same set of vertices, so 2 and 3 are in the same component but 4 and 6 are not so the addition of an edge between 4 and 6 will not result in a cycle.
- O When an edge (u, v) is added to T, the two components that have vertices (u, v) are combined to form a <u>single component</u>. So with the addition of the edge between 4 and 6 above will result in the components that originally contained  $\{1, 2, 3, 4\} + \{5, 6\} ==> \{1, 2, 3, 4, 5, 6\}$
- o <u>Use min-heap operations to get the edges into increasing order of cost</u>. Time complexity O(e log e)
  - The criteria for determining if (u, v) is in the same component is as follows: S1 = find(u); //use FastUnionFind

```
S2 = find(v);
```

#### If (S1 != S2) union(S1, S2);

- Prim's method: Start with a 1-vertex tree and grow it into an n-vertex tree by repeatedly adding a vertex and an edge (if we have a choice then add the <u>least costly edge</u>) (See page 10 for an example). This method is the fastest
- Sollin's method: Start with an n-vertex forest. Each component selects a <u>least-cost edge</u> to connect to another component. Eliminate duplicate selections and possible cycles. This process is repeated until there is only 1 component left. Example given below

  This one is chosen first: because it has the least cost

This one is last

This one is last

This one is 3rd

This one is 3rd

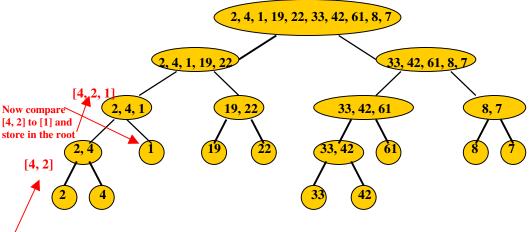
This one is 3rd

This one is 3rd

Page 5

- > Edge <u>rejection</u> greedy strategy:
  - Starting with a connected graph, repeatedly find a cycle and eliminate the highest cost edge on this cycle, stop when no cycle remains
  - Considering the edges in descending order of cost, eliminate an edge provided this leaves behind a connected graph.
- o **Divide and conquer**: a large instance is solved as follows
  - > Divide the large instance into smaller instances (performance is best when the smaller instances that result from the divide step are approximately the same size)
  - Solve the smaller instances somehow, such as continuing to break them into smaller instances until there is only one element per instance. For **Example:** Please see the diagram below: imagine a tree where the root is the original problem and consists of 10 integers in random order, we wish to sort these integers into decreasing order. We first break up the root into sub-trees that contain portions of the root, now we break up these sub-trees into even smaller groups of integers until we finally reach the leaves, which have only single integer. At this time we compare two leaves and sort them accordingly.

Array [2, 4, 1, 19, 22, 33, 42, 61, 8, 7]



Now that we have broken the problem down into much smaller problems we compare the leaves and place them into their root in the proper order. After the divide and conquer merge sort terminates the array would be as follows:

Array [61, 42, 33, 22, 19, 8, 7, 2, 2, 1]

- > Combine the results of the smaller instances to obtain the results of the larger instance
- The working of a recursive divide-and-conquer algorithm can be described as a tree a recursion tree
  - The algorithm moves down the recursion tree dividing large instances into smaller ones
  - The leaves represent *small* instances
  - The recursive algorithm moves back up the tree combining the results from the sub-trees

### Natural merge sort:

4 is larger than 2 so we store [4, 2] in the root

The initially sorted segments are the naturally occurring sorted segments in the input

- The segment boundaries are found by evaluating the items in the input with the criteria of a[i] > a[i+1] for **Example:** If the input array had the following values: [8, 9, 10, 2, 5, 7, 9, 11, 13, 15, 6, 12, 14] then we would have 3 natural segments which are {8, 9, 10} because 10 > 2 but 2 is not > 5 {2, 5, 7, 9, 11, 13, 15} again 15 > 6 but 6 is not > 12 {6, 12, 14}
- The benefit of the natural merge sort is that it can reduce the number of passes that need to be taken with other methods, in the above situation we would only have to passes; the first to merge {8, 9, 10} with {2, 5, 7, 9, 11, 13, 15} which results in {2, 5, 7, 8, 9, 10, 11, 13, 15} and the second pass to merge the result with the last segment of {6, 12, 14} which results in the sorted array [2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

Page 6

# • Quick sort:

- Small instances are when n<=1 where n is an element (remember that every instance this small is considered already sorted)
- To sort the large instances select a **pivot element** from the n elements
- Partition the n elements into 3 groups (left, middle, and right) where the middle group contains only the pivot elements, all the left groups elements are smaller than or equal to the middle groups elements, and the right groups elements are larger than or equal to the middle groups elements. The middle group only contains one element, the pivot point.
- Use recursion to sort the left and right groups
- The answer is the sorted left group, followed by the middle group, followed by the right group
- **REMEMBER** the pivot is the <u>leftmost element</u> in the list to be sorted, so it would be position zero in an array, <u>or we could</u> <u>randomly select a pivot point</u>, or the <u>median-of-three rule</u>. If we randomly choose our pivot point or we use the median-of-three rule we do not need to change our quick sort code, we simply swap the positions of the pivot point and position zero in the array.
- The median-of-three rule: for this rule the pivot is chosen to be the median of the three elements {a[leftEnd], a[(leftEnd+rightEnd)/2], a[rightEnd]}. For **Example** if these elements have keys {5, 9, 7}, then a[rightEnd] is used as the value of pivot because 5 <= 7 <=9 (in other words ask yourself "what is the key in a[0], a[a.length-1], and a[(a.length-1)/2]". Which ever key is the middle value is chosen as the pivot point). Once the pivot has been chosen simply swap its position with the first position in the array and then proceed from this point as if we were using the *leftmost element* rule (see example below).
- **Example:** using a leftmost element pivot point strategy: we have an array containing the following elements: [6, 2, 8, 5, 11, 10, 4, 1, 9, 7, 3] we chose the first element = 6 as the pivot point. We then compare all other array positions with 6, if they are < = 6 then they are placed into the left segment, if they are > = 6 they are placed in the right segment. The left and right segments are then sorted recursively in exactly the same manner. So the array [2, 5, 4, 1, 3] = left segment [8, 11, 10, 9, 7] = right segment and [6] = the middle segment. Next we sort both the left and right segments recursively, so sorting the left segment results in [2] being chosen as the pivot point, [1] = left segment, [5, 4, 3] = right segment and the process continues. This can be done using 2 arrays (array A and B) that are of the same size, where the left segment of array A is stored in the left half of the array B and the right segment of array A is stored in the right half of array B, and the pivot point being stored in the very middle. When the recursion finally terminates we have a sorted array done in O(n log n) which is just awesome compared to O(n<sup>2</sup>).
- We can also do **in-place swaping** instead of using 2 arrays (see the example below). After determining the pivot point we find the <u>left</u>most element (called bigElement) that is <u>larger</u> then the pivot, and the smallest <u>right</u>most element (called smallElement) that is <u>smaller</u> then the pivot, then swap there array positions (<u>provided that the larger element is to the left of</u> the smaller one). When the recursion terminates we swap the pivot with the last smallElement
- Example: in-place swaping: let the color = the pivot point, the color = the bigElement, and the color = the smallElement. We have the following array: [3, 4, 9, 1, 16, 11, 23, 8] 3 = pivot, 4 = bigElement, and 1 = smallElement. We perform the swap and the array looks as follows: [3, 1, 9, 4, 16, 11, 23, 8] Now we do it again: 9 = bigElement, 4 = smallElement, and the pivot doesn't change, so the swap yields the following: [3, 1, 4, 9, 16, 11, 23, 8] Now we do it again: 9 = bigElement, 1 = smallElement, whoop's! bigElement is NOT to the left of smallElement so the recursion terminates, we swap the pivot = 3 and smallElement = 1 which yields the following: [1, 3, 4, 9, 16, 11, 23, 8] and then we return the (somewhat) sorted array to its caller. As you can see once the final array is returned it will be sorted (in increasing order, but could just as easily be in decreasing order).

# Things to study further:

- Know the code to test a graph for a bipartite structure (the code is given on the next page) a question involving bipartite graphs may be on the exam
- \* Know how to create an adjacency matrix (for both weighted, un-weighted, directed, and un-directed) from a given graph and vise versa.

- \* Know Kruskal's, Dijkstra's, Prims, and Sollin's methods, there is sure to be a question involving these (not so much the coding but focus on the concepts)
- Completely understand and be able to reproduce Depth-first-search and Breadth-first-search, problems involving these are certain to be on the exam since they can be used to traverse all graphs.
- Be able to reproduce from memory all the code for both BinaryTreeNode and LinkedBinaryTree (please see page 5 and pages 15 through 18 on "Notes for Exam II") pay special attention to the *height* method of LinkedBinaryTree
- \*\* Know everything there is to know about AVL, HBLT, Binary Search, winner, and loser trees, coding problems involving these may be on the exam (this is why knowing the height method from memory is very useful). For example: write code that tests the balance of an indexed AVL tree and can make adjustments to bring the tree into balance (look at page 605 in the text for information on insertions into an AVL tree)
- Do the sample exams found on the texts web site, time yourself to make sure that you understand the material well enough to complete it in 75 minutes, then check your answers and give yourself a grade. If you don't score well then study the stuff you messed up on (this is how I judge my knowledge of the material and works for me).

#### GOOD LUCK ON THE EXAM!

Page 7

### Code for the bipartite method

We may perform a breadth-first or depth-first search in each component of the graph. In each component, the first vertex that is visited may be assigned to either set. The assignment of the remaining vertices in the component is forced by the bipartite labeling requirement that the end points of an edge have different labels. The code below assigns the first vertex in each component the label 1. It is a modified version of the code for breadth first search.

```
/** Label the vertices such that every edge connects
  ^{\star} a vertex with label 1 to one with label 2
  * @return null iff there is no such labeling
  ^{\star} @return an array of labels when the graph has such a labeling ^{\star}/
public int [] bipartite()
   // make sure this is an undirected graph
   verifyUndirected("bipartite");
   int n = vertices();
   // create label array, default initial values are 0
   int [] label = new int [n + 1];
   // do a breadth first search in each component
   ArrayQueue q = new ArrayQueue(10);
   for (int v = 1; v \le n; v++)
      if (label[v] == 0)
      {// new component, label the vertices in this component
         label[v] = 1;
         q.put(new Integer(v));
         while (!q.isEmpty())
             // remove a labeled vertex from the queue
            int w = ((Integer) q.remove()).intValue();
            // mark all unreached vertices adjacent from w
            Iterator iw = iterator(w);
            while (iw.hasNext())
            {// visit an adjacent vertex of w
               int u = ((EdgeNode) iw.next()).vertex;
               if (label[u] == 0)
               {// u is an unreached vertex
                  q.put(new Integer(u));
                   label[u] = 3 - label[w]; // assign u the other label
                  // u already labeled
                  if (label[u] == label[w])
                     // both ends of edge assigned same label
                     return null;
         }
   // labeling successful
   return label;
```

When the graph is bipartite, each vertex of the graph is added to the queue exactly once. Each vertex is also deleted from the queue exactly once. When a vertex is deleted from the queue, all vertices adjacent to it are examined. It takes Theta(n) time to find and examine the vertices adjacent to vertex w when an adjacency matrix is used and Theta( $d_w$ ) when adjacency lists are used. If the algorithm does not terminate early because of an exception or because the graph is not bipartite, all vertices in the graph get examined. The total examination time is Theta( $n^2$ ) when adjacency matrices are used and Theta(e) when adjacency lists are used. The remainder of the algorithm takes O(n) time. Allowing for exceptions and the fact that the algorithm may terminate early because the graph is not bipartite, we see that the overall complexity is  $O(n^2)$  when adjacency matrices are used and O(n+e) when adjacency lists are used.

A test program, input, and output appear in the files TestBipartite.\*.

# Pseudocode for BFS

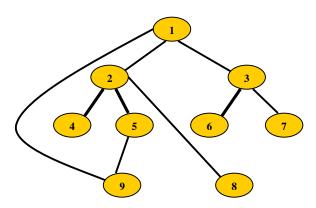
```
breadthFirstSearch(v) {
    Label vertex v as reached.
    Initialize Q to be a queue with only v in it.
    While(Q is not empty) {
        delete a vertex w from the queue.
        Let u be a vertex (if any) adjacent from w.
        While(u) {
            if(u has not been labeled)
            {
                  add u to the queue.
                  Label u as reached.
            }
            u = next vertex that is adjacent from w.
        }
}
```

# **Actual BFS code**

Page 8

```
if (reach[u] = = 0)
{
      //u is an unreached vertex
      q.put(new Integer(u));
      reach[u] = label; //mark as reached
}
```

}



Page 9

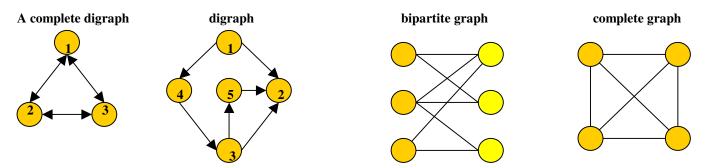
```
Example: Breadth-first-search on the graph above (the same graph that is found on page 3)
```

```
We begin at vertex 1
     Initialize a queue q with only the value 1 in it
                                                                                                                   q = 1
                                                                                                                                          reached = 1
     Delete a vertex w from the queue
                                                                                                                  q = empty
                                                                                                                                          reached = 1
                                                                                            w = 1
                                                                                                       u = none
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 1
                                                                                                       u = 2
                                                                                                                   q = empty
                                                                                                                                          reached = 1
     If u has not been labeled as reached then add it to the queue and label it
                                                                                                       u = 2
                                                                                                                   q = 2
                                                                                                                                          reached = 1, 2
                                                                                            w = 1
     Now u = the next vertex that is adjacent from w that is not labeled
                                                                                            w = 1
                                                                                                       u = 3
     If u has not been labeled as reached then add it to the queue and label it
                                                                                                       u = 3
                                                                                                                                          reached = 1, 2, 3
                                                                                                                   q = 2, 3
                                                                                            w = 1
φ
                                                                                                       u = 9
     Now u = the next vertex that is adjacent from w that is not labeled
                                                                                            w = 1
     If u has not been labeled as reached then add it to the queue and label it
                                                                                            w = 1
                                                                                                       u = 9
                                                                                                                   q = 2, 3, 9
                                                                                                                                          reached = 1, 2, 3, 9
¢
     Now u = the next vertex that is adjacent from w that is not labeled
                                                                                            w = 1
                                                                                                       u = none
     Delete a vertex w from the queue
                                                                                            w = 2
                                                                                                       u = none
                                                                                                                  q = 3, 9
                                                                                                                                          reached = 1, 2, 3, 9
φ
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 2
                                                                                                       u = 4
     If u has not been labeled as reached then add it to the queue and label it
                                                                                            w = 2
                                                                                                       u = 4
                                                                                                                   q = 3, 9, 4
                                                                                                                                          reached = 1, 2, 3, 9, 4
✡
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 2
                                                                                                       u = 5
     If u has not been labeled as reached then add it to the queue and label it
                                                                                            w = 2
                                                                                                       u = 5
                                                                                                                   q = 3, 9, 4, 5
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5
Þ
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                                       u = 8
                                                                                            w = 2
     If u has not been labeled as reached then add it to the queue and label it
                                                                                                       u = 8
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8
                                                                                            w = 2
                                                                                                                   q = 3, 9, 4, 5, 8
ф
Ф
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 2
                                                                                                       u = none
     Delete a vertex w from the queue
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8
                                                                                            w = 3
                                                                                                       u = none
                                                                                                                  q = 9, 4, 5, 8
¢
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 3
                                                                                                       u = 6
     If u has not been labeled as reached then add it to the queue and label it
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6
                                                                                            w = 3
                                                                                                       u = 6
                                                                                                                   q = 9, 4, 5, 8, 6
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 3
                                                                                                       n = 7
     If u has not been labeled as reached then add it to the queue and label it
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6, 7
                                                                                            w = 3
                                                                                                       u = 7
                                                                                                                   q = 9, 4, 5, 8, 6, 7
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 3
                                                                                                       u = none
     Delete a vertex w from the queue
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6, 7
                                                                                            w = 9
                                                                                                       u = none
                                                                                                                   q = 4, 5, 8, 6, 7
ġ
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 9
                                                                                                       u = none
     Delete a vertex w from the queue
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6, 7
                                                                                            w = 4
                                                                                                       u = none
                                                                                                                  q = 5, 8, 6, 7
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 4
                                                                                                       u = none
                                                                                                                  q = 8, 6, 7
     Delete a vertex w from the queue
                                                                                            w = 5
                                                                                                       u = none
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6, 7
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 5
                                                                                                       u = none
     Delete a vertex w from the queue
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6, 7
                                                                                            w = 8
                                                                                                       u = none
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 8
                                                                                                       11 = none
     Delete a vertex w from the queue
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6, 7
                                                                                            w = 6
                                                                                                       u = none
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 6
                                                                                                       u = none
                                                                                                                                          reached = 1, 2, 3, 9, 4, 5, 8, 6, 7
₩
     Delete a vertex w from the queue
                                                                                            w = 7
                                                                                                       u = none
₩
     Let u = a vertex that is adjacent from w that is not labeled
                                                                                            w = 7
                                                                                                       u = none
☼
```

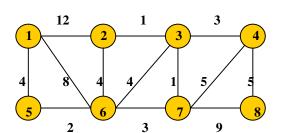
Delete a vertex w from the queue whoop's the queue is empty, so the method terminates and we are left with the reached values above.

Remember: breadth-first-search is <u>very similar to a level order traversal of a tree</u>, this is why we set u = 2, then u = 3, and then

u = 9 when w = 1 instead of some other order.



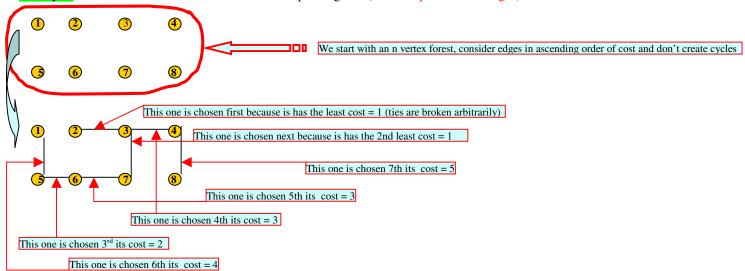
A digraph is strongly connected iff it contains a directed path from i to j and from j to i for every pair of distinct vertices i and j.



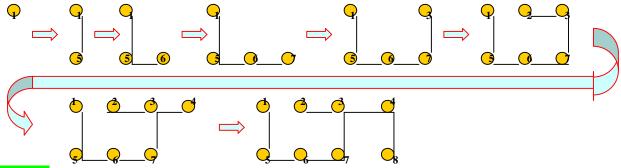
Page 10

The weighted graph above is used in the next 3 examples

Example: kruskal's method: construct a min cost spanning tree (we end up with n – 1 edges)



**Example:** Prim's method: construct a min cost spanning tree (assuming we were instructed to start at vertex # 1)



Example: linked adjacency representation: each adjacency list is a chain; there are 2e node in an undirected graph and e node in a directed graph. Each object is represented as an array of chains where each chain is of type Chain.

