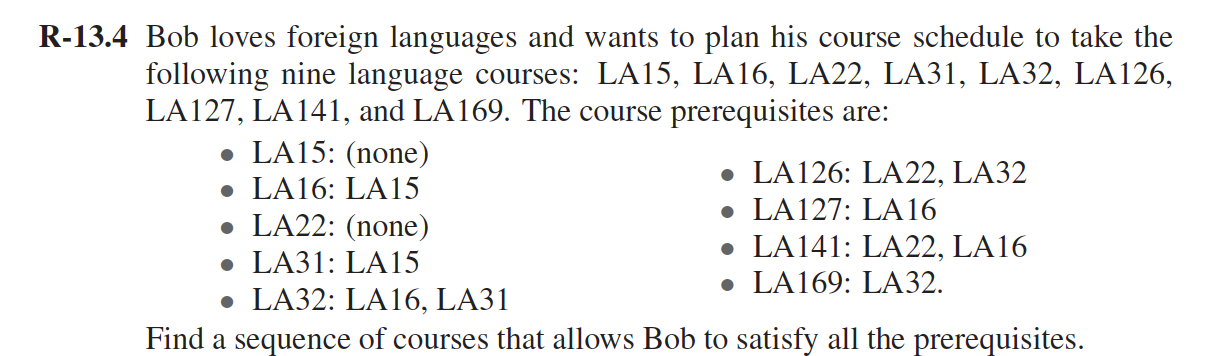
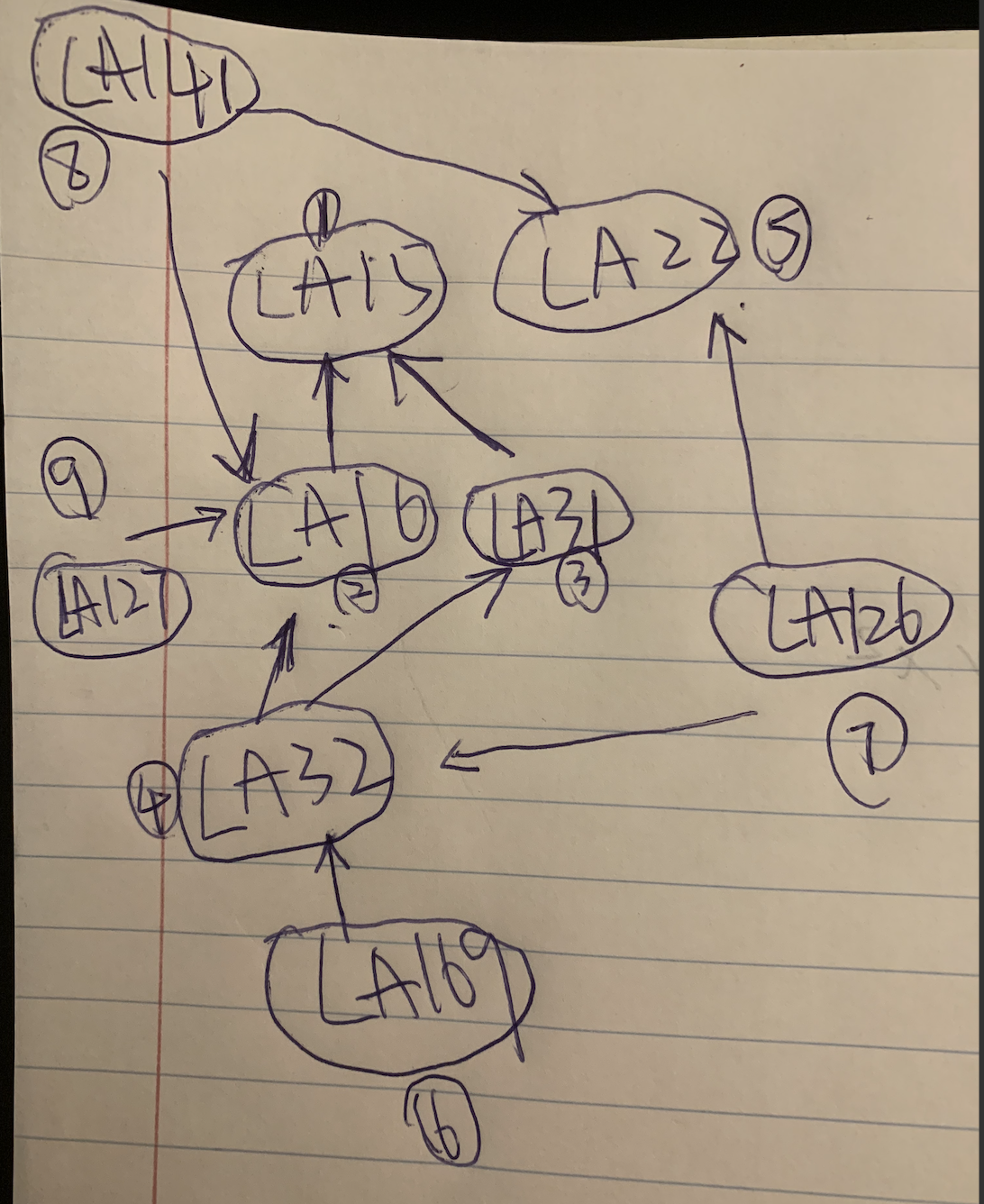
Chapter 13 Exercises: R-13.4,



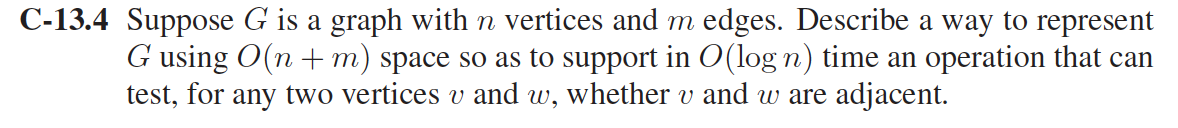


Arrow from A -> B means A requires a prerequisite B. So A->B means you have to do B before doing A.

Therefore, the point with the in degree without out degree should be first done, and sort with the result of ( in degree – out degree) by descending order:

The sequence can be LA15, LA16,LA31,LA32,LA22,LA169,LA126,LA141,AL127.

C-13.4,



We can use Adjacency list representation which takes O(n+m) space to represent a graph G.

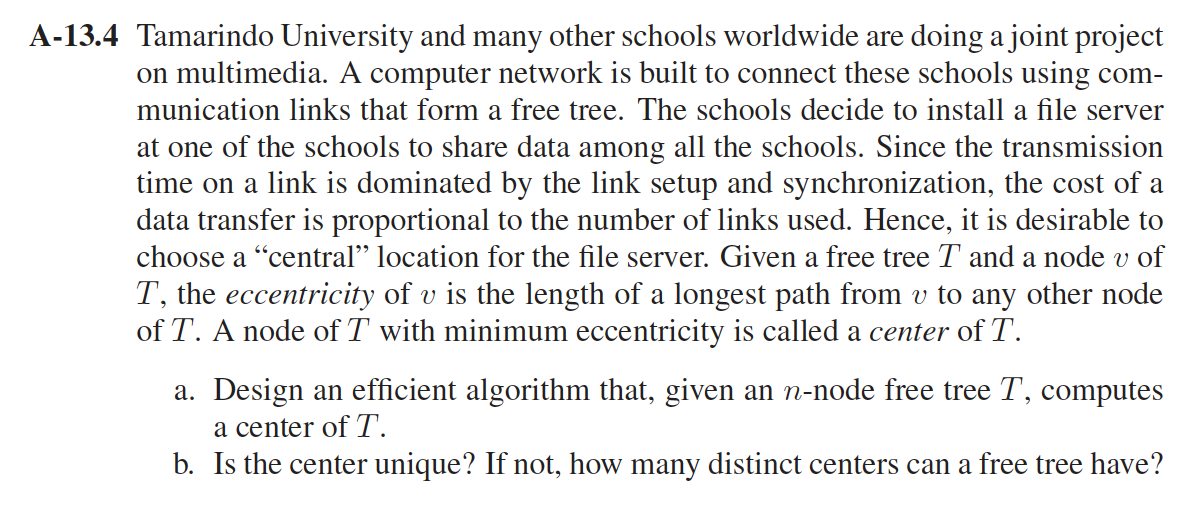
For each vertex v, Adjacency list maintains a list of vertices which are reachable from v.

So we have n list in Adjacency list. Max length of such list is n.

Further note that for vertex v we will keep the list of vertices which are reachable from v in sorted order.

And for checking whether v and v are adjacent, we will use binary search which takes O(logn) time.

A-13.4



1. Since T is a tree, there will be no cycles in the tree T. Every time we mark the children nodes of the tree T and conceptually remove them from the tree T until only one or two nodes are left which are exactly the centers.

The procedure：

1. remove all the children of three T. Let the result tree be T1.

2.remove all the children of the tree T1, let the result tree be T2.

3.repeat the remove operation ,remove all the children of Tree Ti,Let the result tree be Ti+1, until the result tree has only 1 or 2 nodes, we mark it as Tk

4.If Tk has only 1 node ,that is the center of tree T. The eccentricity of center node is k.

5.If Tk has only 2 nodes, either one of them can be the center of tree T. The eccentricity of center node is k+1.

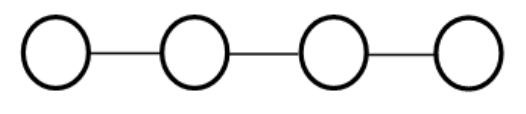
b. The center of the free-tree may not be unique.

For example:

A path with 3nodes has only 1 center:



Whereas a path with 4 nodes has 2 centers:



So a free tree can only have 1 or 2 centers.

Chapter 14 Exercises: C-14.3,

It is necessary in this changed implementation to check if the inserting vertex already exists in the new key value pair because this will increase the duplicity in the new graph and more than one path will then exists in the case when duplicity exists in the graph.

As we implement the priority queue, Q, with a heap,since each insert, extractMin, and decreaseKey operation takes *O*(lg *n*) time, and so the totel time for Dijkstra's algorithm is *O*((*n* + *m*) lg *n*).

A-14.1,

The Dijkstra’s algorithm is used to deal with the monotone requirement in a side-scrolling video game are as follows:

If( movingleft==true){

Define a “neighbor” as any nodes to the left of the current node.

}

Else{

Define a “neighbor” as any nodes to the right of the current node.

}

As per the requirement of the given scenario is to find the neighbors by using private method getNeighbors(node,movingleft)

Algorithm:

Set current\_node = starting\_node

Let current.distance=0

Set all other nodes distance to infinity

Set all other nodes to unvisited

Make a set containing all unvisited nodes( except start node)

Boolean movingLeft=TRUE

// use a while loop find the shortest path distance to end by moving\_left

While( current\_node is not equal to destination\_node && its distance < infinity)

{

For (each unvisited neighbor of current node)

{

Set neighbor.distance= the smaller of its current value && (current.distance+length of edge between current and neighbor)

}

Set current node as visited

Currentdistance= node from unvisted set with smallest distance

}

The minimum distance to end movingLeft=current.distance

Set movingLeft = FALSE;

Set all nodes as unvisted and their distances =0

Set current= start\_node

Set current as visited and its distance =0

While( current\_node != destination\_node && its distance < infinity)

{

For( each unvisited neighbor of current node)

{

Set neighbor.distance to the smaller of its current value &&( current.distance +length of edge between current and neighbor)

}

Set the minimum distance to end movingRight = current.distance

If( distance to end movingLeft =distance to end movingRight)

{

Find the overall minimum distance to the end

}

Else if ( distance to end movingLeft && distance to end movingRight (both)==infinity)

{

No way to reach the end;

}

A-14.4

The graph is a di-graph，vertices are airports and di-edges are flights with two weights, departure time, departT and arrival time, arriveT.

Distance function should be earliest arrival time, T, at the airports.

The arrival time of a sequence of flights is the arrival time at the destination airport, i.e. last flight.

use a “minimum” priority queue, Q, of airports keyed by T.

But the arrival time at the start vertex (i.e. origin airport), s, T[s] = startT, start time.

The adjacent loop must be modified:

·      Can only select flights, f, that can be caught, i.e.  f.departT ∈ T[v]

·      Also want flight with the minimum arrival time.

Make another priority queue, P:

**for** all vertices, w, adjacent to v **do**

**make** Priority Queue, P, of flights, f, with f.departT ∈ T[v] keyed by f.arriveT

Time t ++

**if** P is not empty **then** t =P.min().arriveT  // no need to remove

Relaxation is nearly the same

**if** t < T[w] **then**

T[w] = t

**update** w in Q

Algorithm：

FlightAgenda(DiGraph G, Vertex s, Time startT)

            // Input: G di-graph, not a simple graph

// vertices are airports

// di-edges are flights with two weights

//     departT is the departure time at origin airport

//      arriveT is the arrival time at the destination airport

// s is the origin airport and startT is the starting time

// initialize arrival time, T

T[s]= startT

**make** Priority Queue, Q, of vertices keyed by T

**while** Q is not empty **do**

v = Q.removeMin()

**for** all vertices, w, adjacent to v **and** in Q **do**

// determine the next flight

**make** Priority Queue, P, of fights, f, with f.departT ∈ T[v] keyed by f.arriveT

Time t ++

**if** P is not empty **then** t = P.min().arriveT  // no need to remove

// relaxation

**if** t < T[w] **then**

T[w] = t

**update** w in Q

**return** T