EXERCISE 3. Suppose the edges of a graph are presented to you only once (for example over a network connection) and you do not have enough memory to store all of them. The edges do not necessarily arrive in sorted order.

(i) Outline an algorithm that nevertheless computes a minimum spanning tree using space in $O(|V|)$.

total points: 12

EXERCISE 4. Finding the quickest routes in public transportation systems can be modelled as a shortest-path problem for an acyclic graph. Consider a bus, tram or train leaving a place p at time t and reaching its next stop p' at time t'. This connection is viewed as an edge connecting nodes (p, t) and (p', t'). Further, for each stop p and subsequent events (arrival and/or departure) at p, say at times t and t' with $t < t'$, we have the waiting link from (p, t) to (p', t').

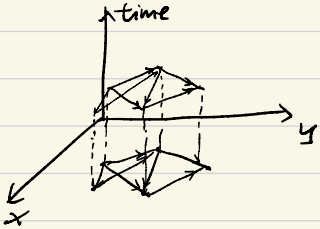
(i) Show that the graph obtained in this way is a directed acyclic graph.

(ii) Suppose you have computed the shortest-path tree from your starting node in space and time to all nodes in the public transportation graph reachable from it. How do you actually find the route you are interested in?

total points: 14

Exercise 4 :

a) Since the graph obtains has two characters : position and time. We can model the graph like below.



the vertices exists in the 3D space.

Then it's obvious to find two basic properties.

① A walk through the graph is a path which only goes upward in the 3D graph.

② Since walking from one position to another takes time, so for each map at certain time t', no edges exist.

Then suppose we have a path from (p, t) to (p', t'), with $t' > t$. It's impossible to find a way back since time flows in one direction, No cycles would exist in such graph.

b) Since we already get the shortest path tree, we can use depth first search and back-tracking to find the path.

Algorithm :

```
dfs(root, target)
    stk = Stack()
    marked = {}
    for i in vertexList
            marked[i] = 0
    Path = DirectedGraph
    search(root, target, stk)
    While stk is not empty:
            path.insertEdge(stk.pop())
    return path
```

#see next page

```
search (root, target, stack):
    if root == target:
        return True
    for edge in outgoingEdges (root):
        next_vertex = edge[1]
        if marked[next_vertex] == 0:
            marked[next_vertex] = 1
            if search (root, target):
                stack.push([root, edge])
            return True
    return False.
```