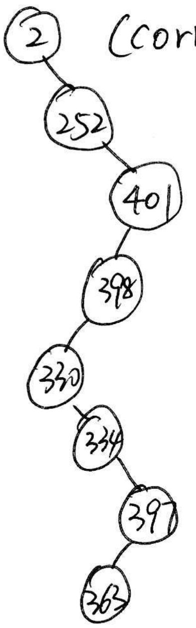
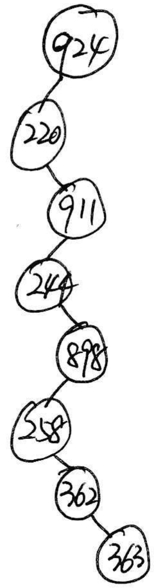
1. (1)



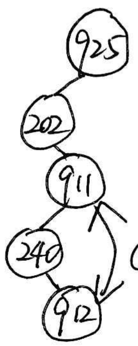
(correct)

(2)



(correct)

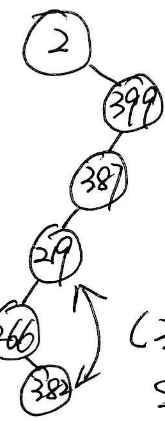
(3)



(wrong)

(912 cannot be greater than 911)

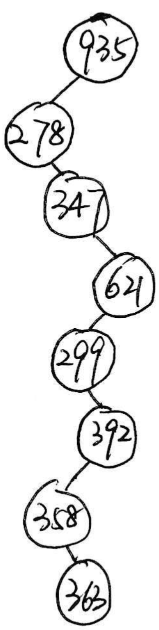
(4)



(wrong)

(382 should be smaller than 219)

(5)



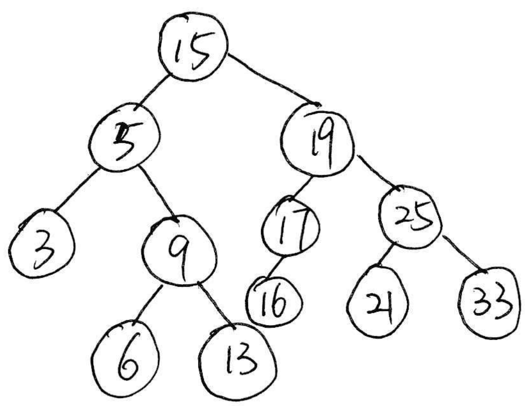
(correct)

2. Since if the successor has one left child, it will smaller than the node, so ~~s~~ the successor will be that left child instead of that node, thus the successor does not have any left child. The same thing we can proof that its predecessor does not have any right child.

Thus a, b, e can be search trace, while c, d are not.

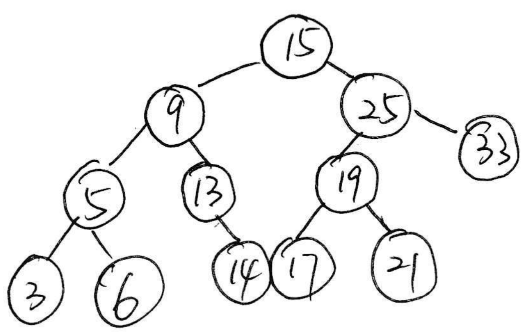
~~P~~3. The running time depends on the structure of the BST. If BST is balanced or nearly balanced, which means the height of the tree is around $\lg n$ whenever you insert an element, the running time will be $\Theta(n\lg n)$ including post-tranverse which costs $\Theta(n)$ time. If the BST is nearly ~~seo~~ in sequence. the running time will be $\Theta(n^2)$ Thus worst-case running time is $\Theta(n^2)$, and the best-case running time is $\Theta(n\lg n)$

4. ~~a)~~ left substree of left child.
1)
a) insert 16: ~~a)~~ b) one rotation c)



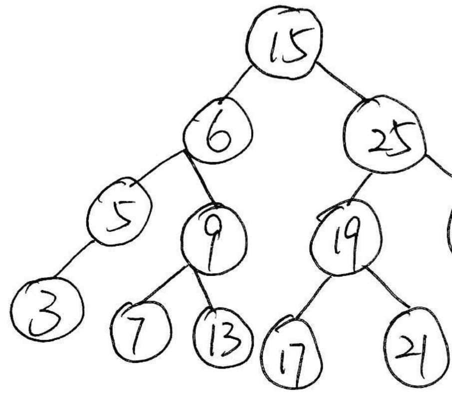
right subtree of right child

2) a) insert 14 b) one rotation c)



~~left~~ right subtree of ~~right~~ left child

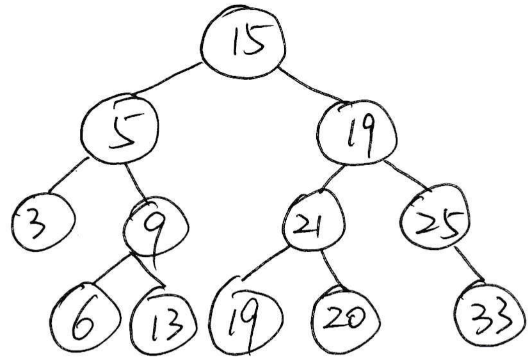
3) a) insert 7 b) double rotation c)



4) left subtree of right child

a) insert 20 b) double rotation

c)



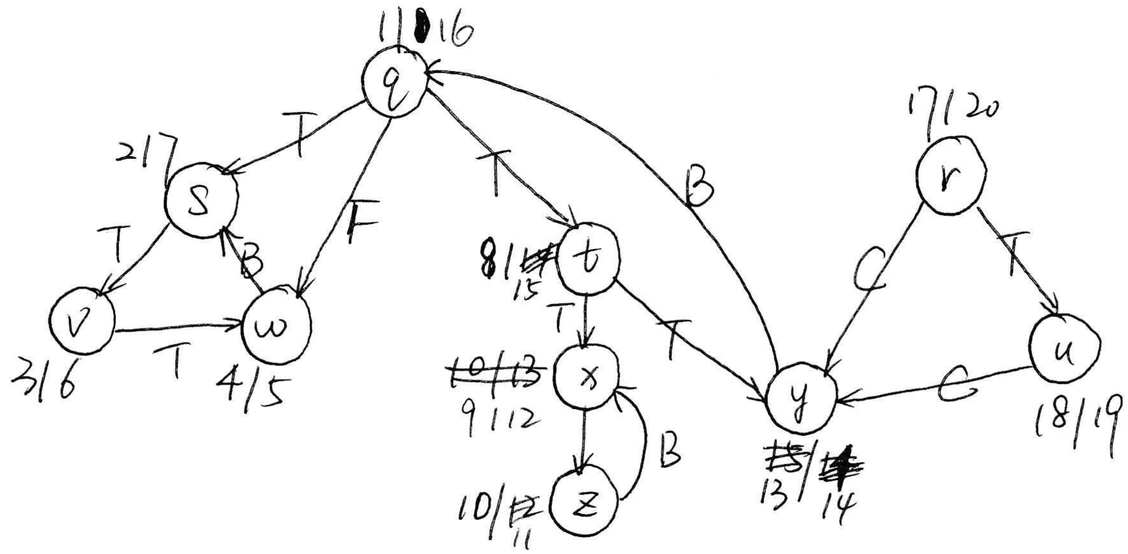
5. Suppose we look at cell $A(i,j)$. If $A(i,j)=0$, then vertex j cannot be an universal ~~st~~ sink. If $A(i,j)=1$, vertex i cannot be a universal link. The algorithm is as follows. We create a ~~p~~ set of potential universal sinks. Each time we pick two different from it. And we can claim that after $n-1$ steps, there is only one element left in that set since $A(x,y)$ is either 1 or 0. Then we can check the vertex left. Let's say vertex u. We can declare u as a universal sink if the whole row is zero and the whole column is one except for $A(u,u)$. Otherwise the matrix does not have a universal sink. The total running time is $(n-1)+(2n-1) = O(n)$

6. (1) proof: Since for each node u, we are going to traverse its adjacent array. suppose node u has distance d, then each of its adjacent vertex will have distance d+1. Also this is irrelevant to the order that these vertexes in adjacent list appear.

(2) proof: In figure 22.3 if t is prior to x in Adj[w] we can get the breath-first tree shown in the figure. But if x is prior to t in Adj[w] and u precedes y in Adj[x], we can get edge (x.u) in the breath-first tree.

7. tree ~~eg~~ edge: T Back edge: B

Forward edge: F Cross edge: C



8. revised code for DFS-VISIT(G.u)

```
time = time+1
u.d = time
u.color = GRAY
for each v ∈ G:Adj[u]
    if v.color == WHITE
        print ("(u, v) is a tree edge")
        v.π = u
        DFS-VISIT(G, v)
    elseif v.color == GRAY
        print ("(u,v) is a back edge")
    elseif v.color == BLACK
        print (" (u,v) is a forward edge or cross edge")
u.color = BLACK
time = time+1
u.f = time
```

if graph G is an undirected graph, then (u,v) is one tree edge if the color of v is white, and (u,v) is one back edge if the color of v is black or gray.

9. a) 1. if (u,v) is a back edge or forward edge, then one vertex must be predecessor of another vertex. However, if one vertex has been detected and, since it is BFS, the other vertex will be detected in (d+1), thus (u,v) should be a tree edge rather than a back edge or a forward edge.

2. ~~if $v.d \neq u.d+1$ then $v.d$ (u,v) is a~~

Since it is BFS, $v.\pi = u$ and $v.d = u.d+1$ for each $v \in G.Adj[u]$. Thus for each tree edge (u,v), $v.d = u.d+1$.

3. Consider a cross edge (u,v), u is visited before v. vertex v must be already on the queue, otherwise (u,v) will be a tree edge. Because v is on the queue, $d[v] \leq d[u]+1$ by Lemma 22.3. By Corollary 22.4, we have $d[v] \geq d[u]$. Thus either $d[v] = d[u]$ or $d[v] = d[u]+1$

b) 1. if there is a forward edge (u,v), then we should have visited it while exploring u, then it should be a tree edge rather than a forward edge.

2. according to the property of BFS, an edge (u,v) is a tree edge only if $\pi[v] \leftarrow u$, and we only do this when we set $d[v] \leftarrow d[u]+1$

3. We cannot have $d[v] > d[u]+1$ since we will visit v as soon as we discover edge (u,v). Thus $d[v] \leq d[u]+1$

4. For a back edge (u,v), v is an ancestor of u or $v = u$, thus $d[v] \leq d[u]$, and $d[v] \geq 0$