3.7)
while(S1 is not empty)
{
    // remove top element from S1
    checked_value = S1.pop();

    // store checked val into s2 in sorted stack
    while( (S2.empty() != ! and (S2.peek() < checked_value) )
    {
        // elements from s2 is popped, value smaller than checked value is pushed to s1
        S1.push( S2.pop() );
    }

    // if while condition not true, then checked val is pushed to s2
    S2.push(x);
}
Time complexity will be O(n^2)

3.9)

Adjacency matrix takes up n^2 spaces and each entry takes up 1 bit (0 or 1)

adjacency list takes up |E| spaces and each entry takes up (how many bits? let's call this amount x)
so, it's more efficient use the adjacency matrix when n^2 bits< |E|*x bits is true.

3.10)

Sufficient Condition

Let G=(V,E) be bipartite.
So, let V=A∪B such that A∩B=∅ and that all edges e∈E are such that e is of the form (a,b) where a∈A and b∈B.
(This is the definition of a bipartite graph.)
Suppose G has (at least) one odd cycle C.
Let the length of C be n.
Let C=(v1,v2,...,vn,v1).

WLOG, let v1∈A. It follows that v2∈B and hence v3∈A, and so on.

Hence we see that ∀k∈{1,2,...,n}, we have:
vk∈{A:B:k oddk even}
But as \( n \) is odd, \( v_n \in A \).

But \( v_1 \in A \), and \( v_nv_1 \in C_n \).
So \( v_nv_1 \in E \) which contradicts the assumption that \( G \) is bipartite.
Hence if \( G \) is bipartite, it has no odd cycles.

**Necessary Condition**

It is enough to consider \( G \) as being connected, as otherwise we could consider each component separately.
Suppose \( G \) has no odd cycles.
Choose any vertex \( v \in G \).
Divide \( G \) into two sets of vertices like this:
Let \( A \) be the set of vertices such that the shortest path from each element of \( A \) to \( v \) is of odd length;
Let \( B \) be the set of vertices such that the shortest path from each element of \( B \) to \( v \) is of even length.
Then \( v \in B \) and \( A \cap B = \emptyset \).
Suppose \( a_1, a_2 \in A \) are adjacent.
Then there would be a closed walk of odd length \( (v, \ldots, a_1, a_2, \ldots, v) \).
But from Graph containing Closed Walk of Odd Length also contains Odd Cycle, it follows that \( G \) would then contain an odd cycle.
This contradicts our initial supposition that \( G \) contains no odd cycles.
So no two vertices in \( A \) can be adjacent.
By the same argument, neither can any two vertices in \( B \) be adjacent.
Thus \( A \) and \( B \) satisfy the conditions for \( G = (A \cup B, E) \) to be bipartite.

**3.20)**

A binary search tree is a binary tree with a special property called the BST-property, which is given as follows:

For all nodes \( x \) and \( y \), if \( y \) belongs to the left subtree of \( x \), then the key at \( y \) is less than the key at \( x \), and if \( y \) belongs to the right subtree of \( x \), then the key at \( y \) is greater than the key at \( x \).
We will assume that the keys of a BST are pairwise distinct.

Each node has the following attributes:

- \( p \), left, and right, which are pointers to the parent, the left child, and the right child, respectively, and
- key, which is key stored at the node.

**DELETE**

Suppose we want to delete a node \( z \).
1. If \( z \) has no children, then we will just replace \( z \) by nil.
2. If \( z \) has only one child, then we will promote the unique child to \( z \)'s place.
3. If \( z \) has two children, then we will identify \( z \)'s successor. Call it \( y \). The successor \( y \) either is a leaf or has only the right child. Promote \( y \) to \( z \)'s place. Treat the loss of \( y \) using one of the above two solutions.
ALGORITHM

This algorithm deletes z from BST T. BST-Delete(T, z)
1: if left[z] = nil or right[z] = nil
2: then y ← z
3: else y ← BST-Successor(z)
4: y is the node that’s actually removed.
5: Here y does not have two children.
6: if left[y] 6= nil
7: then x ← left[y]
8: else x ← right[y]
9: x is the node that’s moving to y’s position.
10: if x 6= nil then p[x] ← p[y]
11: p[x] is reset if x isn’t NIL.
12: Resetting is unnecessary if x is NIL.
13: if p[y] = nil then root[T] ← x
14: If y is the root, then x becomes the root.
15: Otherwise, do the following.
16: else if y = left[p[y]]
17: then left[p[y]] ← x
18: If y is the left child of its parent, then
19: Set the parent’s left child to x.
20: else right[p[y]] ← x
21: If y is the right child of its parent, then
22: Set the parent’s right child to x.
23: if y 6= z then
24: { key[z] ← key[y]
25: Move other data from y to z } 27: return (y)

Time Complexity:

The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

(2p) Let R be the relation represented by the digraph in Fig 3.2 (page 105, on the right side). Please draw the graphs representing R^i for i = 2, 3, 4, and R^*.
$R'$

- $(a, b)$
- $(a, c)$
- $(b, e)$
- $(c, e)$
- $(d, b)$
- $(d, c)$
- $(d, e)$
- $(e, a)$

$R^2$

- $(a, e)$
- $(a, e)$
- $(b, a)$
- $(c, a)$
- $(d, e)$
- $(d, e)$
- $(d, a)$
- $(e, b)$
- $(e, c)$
• (2p) Please draw the sequences of 2-3 tree and 2-3-4 tree, respectively, after each insertion of the following elements (in the given order): 10, 9, 8, 7, 6, 5, 4, 3, 2, 1. The trees are assumed empty initially.