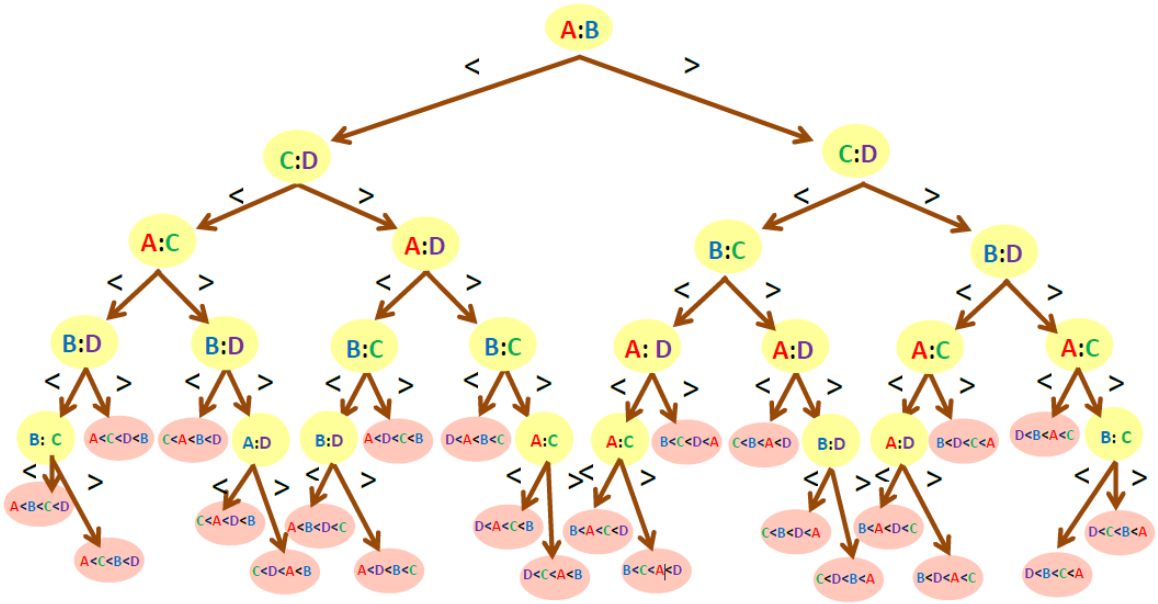
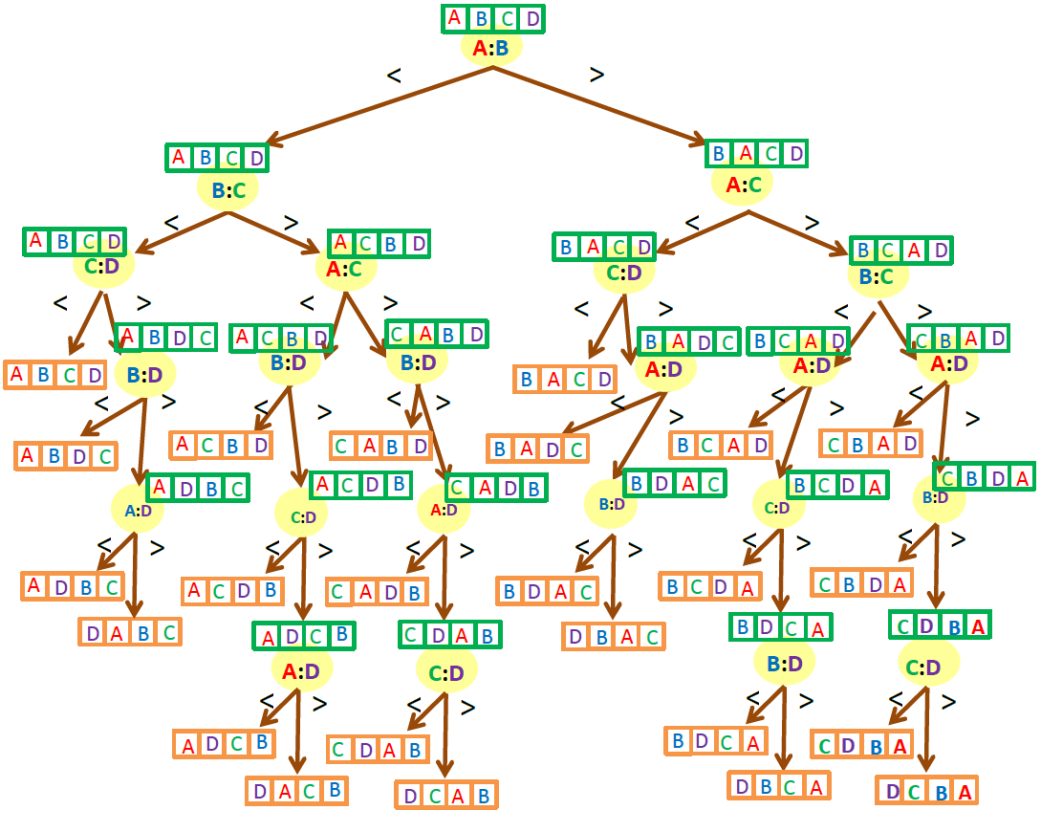
Problem 8-1 (a) Please give an optimal decision tree with four elements a, b, c, and d. (b) Please give a decision tree for insertion sort operating on four elements a, b, c, and d.

Solution:

(a)



(b)



Problems 8-2 Sorting in place in linear time

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.
b. Give an algorithm that satisfies criteria 1 and 3 above.
c. Give an algorithm that satisfies criteria 2 and 3 above.
d. Can any of your sorting algorithms from parts (a)-(c) be used to sort n records with b-bit keys using radix sort in $O(bn)$ time? Explain how or why not.
e. Suppose that the n records have keys in the range from 1 to k. Show how to

modify counting sort so that the records can be sorted in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (Hint: How would you do it for $k = 3$?)

Solution:

a. Counting sort runs in $O(n)$ time and is a stable sorting algorithm.

b. We can perform one pass of a Quicksort PARTITION algorithm around the pivot $x = 0$. If it is a Lomuto PARTITION, it will place all elements ≤ 0 (all 0's) on the left and all elements > 0(all 1's) on the right. Effectively, this sorts the array. It is in place, and it has a $\Theta(n)$ running time. The only thing is that we have to work with a fictitious pivot that is not necessarily an element of the array, and hence we do not have to worry about correctly placing that pivot.

$i \leftarrow 0$

for $j \leftarrow 1$ to n

do if $A[j] \leq 0$

then $i \leftarrow i + 1$

exchange $A[i] \leftrightarrow A[j]$

c. Insertion sort is an in place sorting algorithm. It is also stable. To see this consider $A[i] = A[j]$ and $i < j$. Since $i < j$, $A[i]$ will be considered first. $A[i]$ will be added at the correct position (by shifting) into the sorted array $A[1..i-1]$. This will result in a sorted array $A[1..i]$ containing the original $A[i]$ at some position $k \leq i$. So $A[i]$ is now $A[k]$. When $A[j]$ is considered, $A[j]$ has to be shifted down into $A[1..j-1]$ of which $A[1..i]$ is a subarray containing $A[k]$ (originally $A[i]$). $A[j]$ cannot bypass $A[k]$ in the shifting process because $A[k] = A[j]$. Therefore, the original $A[i]$ and the original $A[j]$ will preserve their relative order.

d. part (a) – Counting sort. Counting sort runs in $O(n)$ times, and for b-bit keys with each bit value varies from 0 to 1, we can sort in $O(b(n + 2)) = O(bn)$ time.

e. The modified counting sort is not stable:

1. Initialize $c[0], c[1], \ldots, c[k]$ to 0
2. /* First, set $c[j]$ = # elements with value j */
   For $x = 1, 2, \ldots, n$; increase $c[A[x]]$ by 1
3. /* Set $c[j]$ = location to place next element whose value is j (iteratively)*/
   For $y = 1, 2, \ldots, k$; $c[y] = c[y\text{-}1] + c[y]$
4. /* Set $p[j] = c[j]$ to record correct position for placing elements*/
   For $y = 0, 1, \ldots, k$; $p[y] = c[y]$

```
5.  /* Process A*/
6.  x = 1;
    While x <= n
    {
        /* If A[x] is correctly placed */
        if(p[A[x]-1] < x && x <= p[A[x]] )
            /* incrementing x to check next position */
            x = x + 1
        /* else */
        else
            /* put A[x] in place, exchanging with the element there */
            Exchange A[x] with A[c[A[x]]]
            /* Update counter */
            Decrease c[A[x]] by 1;
    }
```