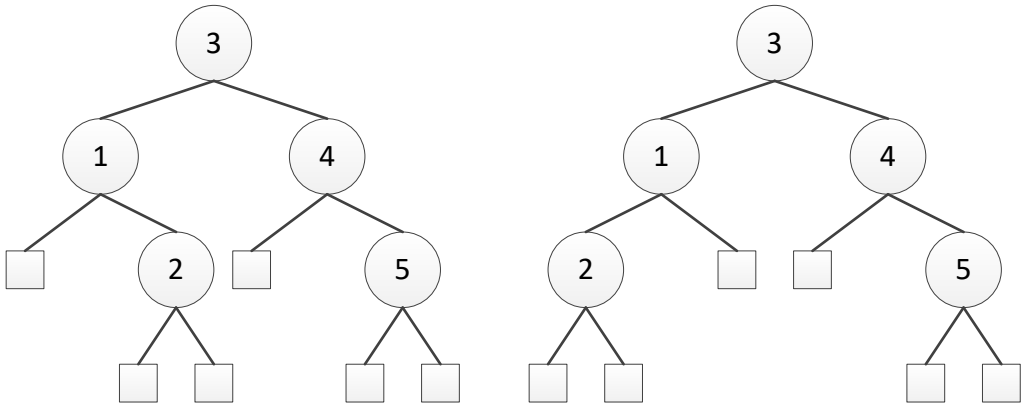
R-3.3

Professor Amongus claims he has a "patch" to his claim from the previous exercise, namely, that the order in which a fixed set of elements is inserted into an AVL tree does not matter – the same AVL tree results every time. Give a small example that proves that Professor Amongus is still wrong.

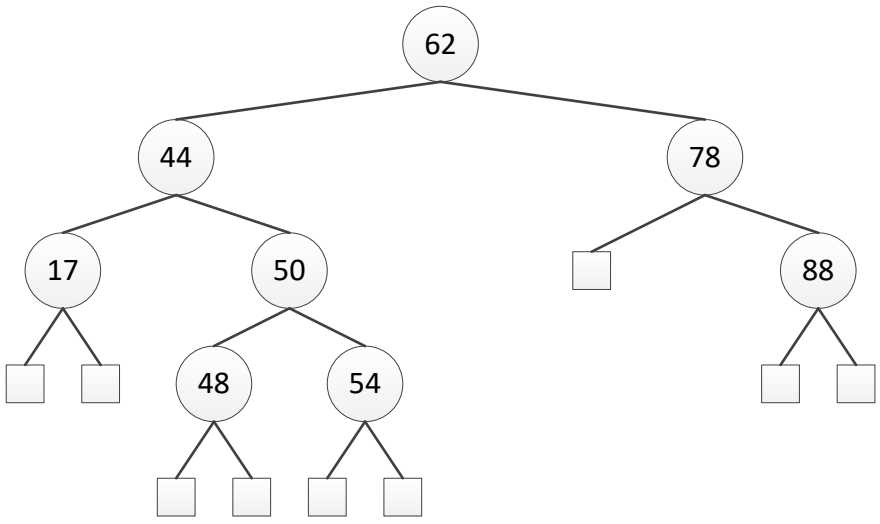
One solution is to draw an AVL tree created by 3, 1, 4, 2, and 5, and to draw another AVL tree created by 3, 2, 4, 1, and 5. These two trees are:



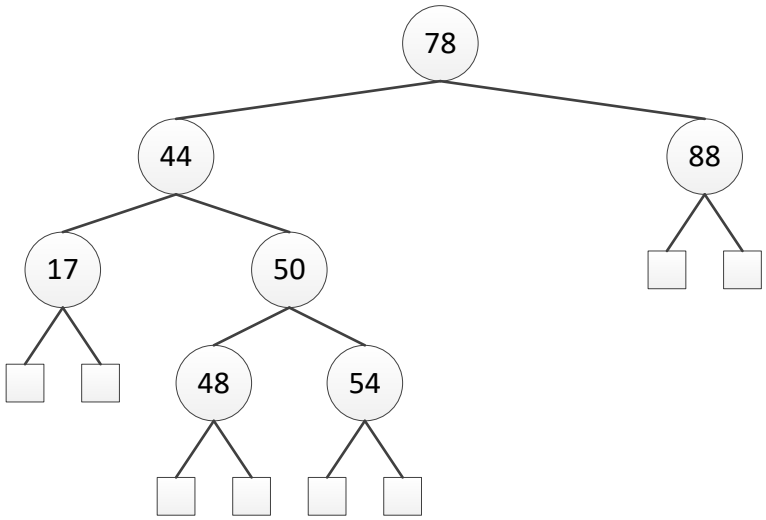
Obviously, these two trees are different. As a result, the professor is wrong.

R-3.6

Draw the AVL tree resulting from the removal of the item with key 62 from the AVL tree of Figure 3.15b.

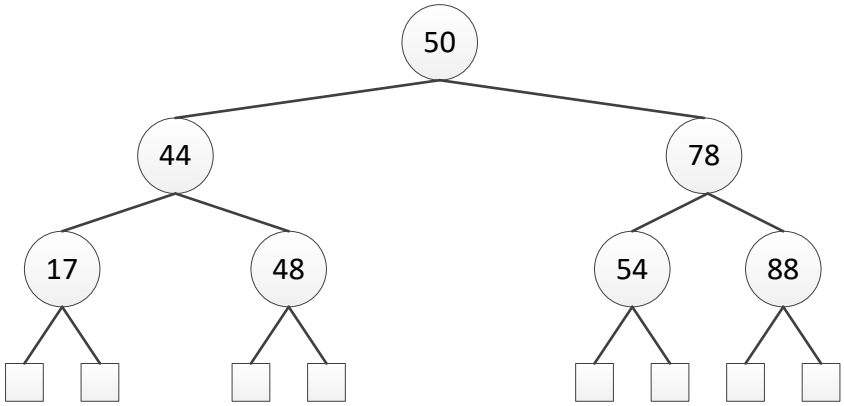


After removal:



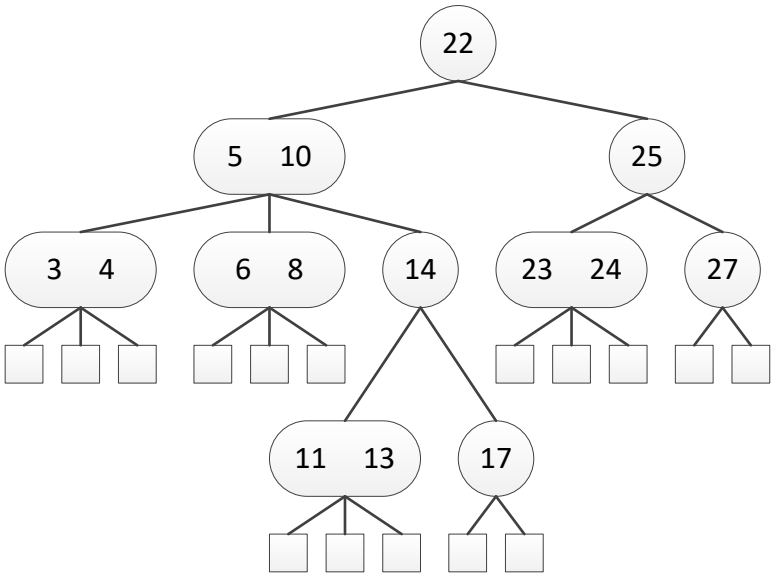
Obviously, this tree is unbalanced. We need to rebalance this tree.

After double rotation:



R-3.8

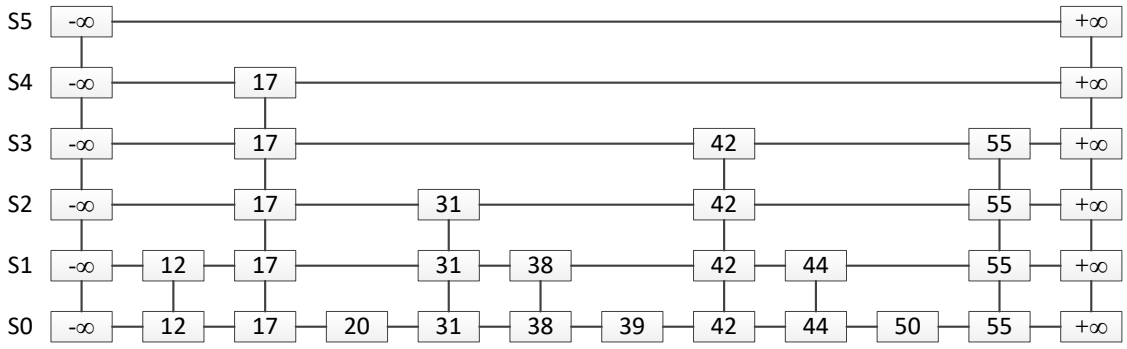
Is the multi-way search tree of Figure3.17a a (2, 4) tree? Justify your answer.



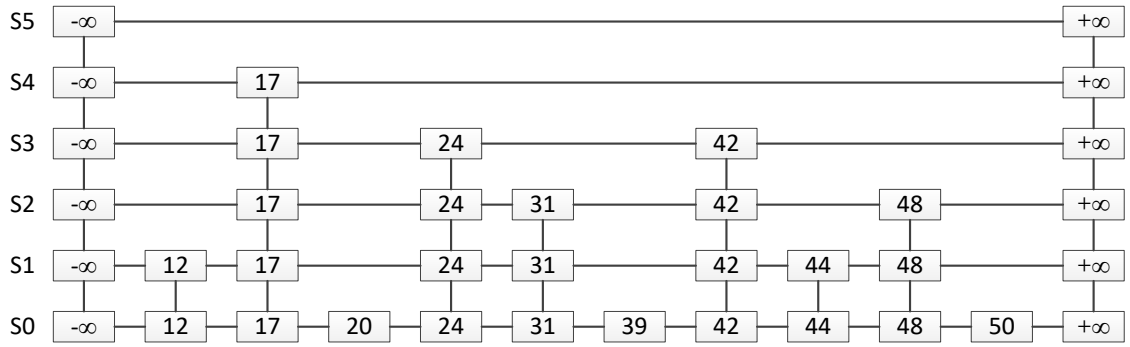
No. The depth property of (2, 4) tree is that all external nodes have the same depth. Obviously, some nodes in this tree do not have the same depth. So this tree is not a (2, 4) tree.

R-3.18

Draw an example skip list resulting from performing the following sequence of operations on the skip list in Figure 3.51: removeElement(38), insertItem(48, x), insertItem(24, y), removeElement(55). Assume the coin flips for the first insertion yield two heads followed by tails, and those for the second insertion yields three heads followed by tails.



After remove 38, insert 48(x=3), insert 24(y=4) and remove 55, the skip list is:



C-3.3

Design an algorithm for performing the operation findAllElements(k) in an ordered dictionary implanted with a binary search tree T, and show that it runs in time $O(h + s)$, where h is the height of T and s is the number of items returned.

We need this operation to return all the elements in the dictionary with key equal to k. So we should prepare for a container to save all these elements, and finally we can return this container. And we need to assume that duplicates are placed in the right subtree.

First, we need to traverse this tree from root node. If k is equal to the key of this node, we will add this value of this node to the container, and we will traverse its right subtree to continue. If k is smaller than the key of this node, we will traverse its left subtree to continue. If k is bigger than the key of this node, we will traverse its right subtree to continue. If the node is an external node, we will finish the traverse and return the container. The above process need us to traverse the tree from root to external node, and the height of this tree is h, so it need $O(h)$ time to traverse.

Finally, we need to output the container including s numbers, so it need $O(s)$ time to output. As a result, this algorithm need $O(h + s)$ time to finish.

Algorithm findAllElements(k, v, c):

Input: k(search key), v(node), c(container of elements)

Output: a container of elements

if v is an external node **then**

return c

if k=key(v) **then**

AddElement v into c

return findAllElements(k, T.rightchild(v), c)

else if k<key(v) **then**

return findAllElements(k, T.leftchild(v), c)

else if k>key(v) **then**

return findAllElements(k, T.rightchild(v), c)

C-3.14

Let T and U be (2, 4) trees storing n and m items, respectively, such that all the items in T have keys less than the keys of all the items in U. Describe an $O(\log n + \log m)$ time method for joining T and U into a single tree that stores all the items in T and U(destroying the old versions of T and U).

First, we need to go down the right-most side of tree T, finding the height of tree T, $h(t)$, which takes $O(\log n)$ time. And we also need to go down the right-most side of tree U, finding the height of tree U, $h(u)$, which takes $O(\log m)$ time.

Second, we need to choose a node as w. This node can be either the biggest node in tree T or the smallest node in tree U. This node can be found during step 1 and that external node is the result. In this algorithm, we choose the first method.

Third, if $h(t) > h(u)$, we move the node w to right-most node of tree T at the level which is $h(t) - h(u) - 1$, and link this node to the root of tree U, which takes $O(\log n)$ time to traverse to that level. If $h(t) < h(u)$, we move the node w to left-most node of tree U at the level which is $h(u) - h(t) - 1$, and link this node to the root of tree T, which takes $O(\log m)$ time to traverse to that level. If $h(t) = h(u)$, we can use the node w as the root of a new tree, and add tree T to the left child and tree U to the right child, which takes $O(1)$ time.

If $h(t) > h(u)$, the time is $O(\log n) + O(\log m) + O(\log n) = O(\log n + \log m)$.

If $h(t) < h(u)$, the time is $O(\log n) + O(\log m) + O(\log m) = O(\log n + \log m)$.

If $h(t) = h(u)$, the time is $O(\log n) + O(\log m) + O(1) = O(\log n + \log m)$.

As a result, the total time for this algorithm is $O(\log n + \log m)$.

Algorithm join(T, U):

h(t)=1, h(u)=1

w<-FindHeight(T, T.root, h(t))

r<-FindHeight(U, U.root, h(u))

if h(t)>h(u) **then**

```
    s<-FindLevel(T, T.root, h(t)-h(u)-1, 1)
    T.insert(s, w.value)
    T.child(s, T.type(s)+1)<-U.root
    return T
  if h(t)<h(u) then
    s<-FindLevel(U, U.root, h(u)-h(t)-1, 0)
    U.insert(s, w.value)
    U.child(s, 1)<-T.root
    return U
  if h(t)=h(u) then
    S.insert(S.root, w.value)
    S.child(S.root, 1)<-T.root
    S.child(S.root, 2)<-U.root
    return S

Algorithm FindHeight(T, v, h)
  if v is an external node then
    return v
  else
    return FindHeight(T, T.child(v, T.type(v)), h+1, flag)

Algorithm FindLevel(T, v, h, flag)
  if h=0 then
    return v
  else if flag=0 then
    return FindLevel(T, T.child(v, 1), h-1, flag)
  else if flag=1 then
    return FindNode(T, T.child(v, T.type(v)), h-1, flag)
```