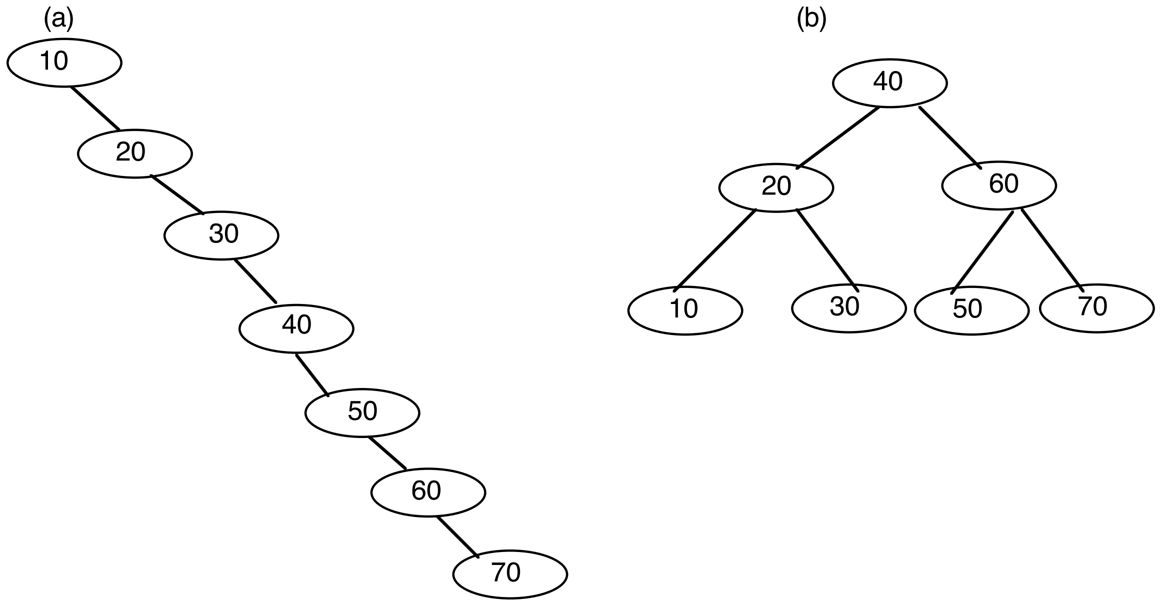
Chapter 12-1, Lecture Notes

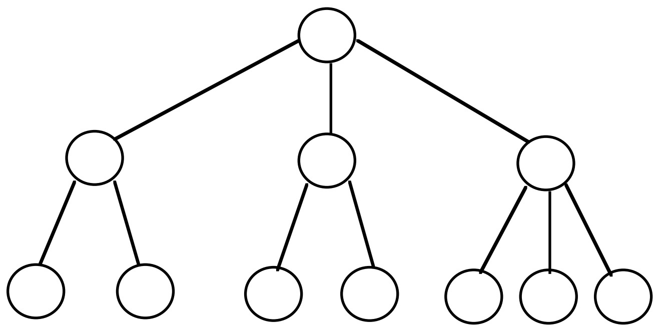
Balanced Search Trees

1. In searching a tree, the height needs to be minimum height.
   1.1 For instance, Figure 12-1:



   1.2 Searching the tree with maximum tree of (a) takes the longest while the minimum height is the most efficient (b).

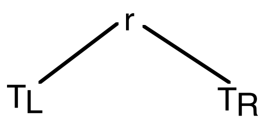
2. There are other search trees rather than the binary search tree.
   2.1 The 2-3 tree is a tree with the internal nodes having either 2 or 3 children.
      2.1.1 The two-node would have two children or three-node would have three children.
      2.1.2 The 2-3 tree is not a binary tree but has similarity to a full binary tree.
      2.1.3 Figure 12-2 has an example of a 2-3 tree:



   2.2 A 2-3 tree with N nodes never has height greater than $\lceil \log_2(N + 1) \rceil$.

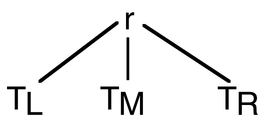
2.2.1 A 2-3 node does not need to be a search tree.
2.2.2 However, the 2-3 tree is generally considered a 2-3 search tree.

2.3 The definition of a 2-3 tree is:
2.3.1 T is empty (a 2-3 tree of height 0)
2.3.2 T is of the form



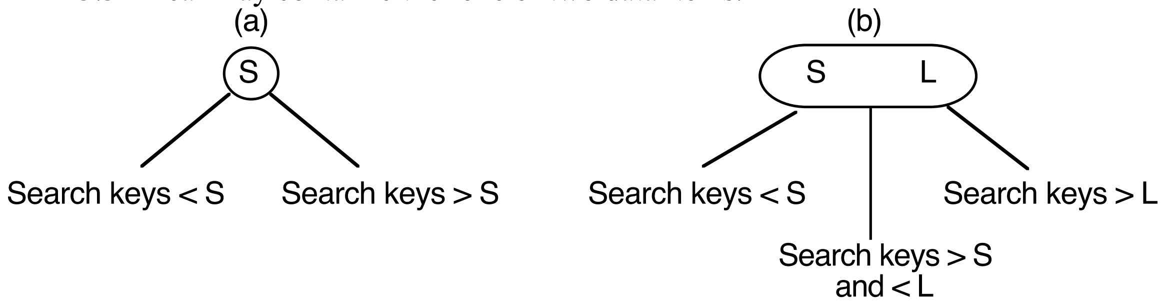
2.4 The search key in r must be greater than the right node and less than the left node.

2.5 T is of the form:

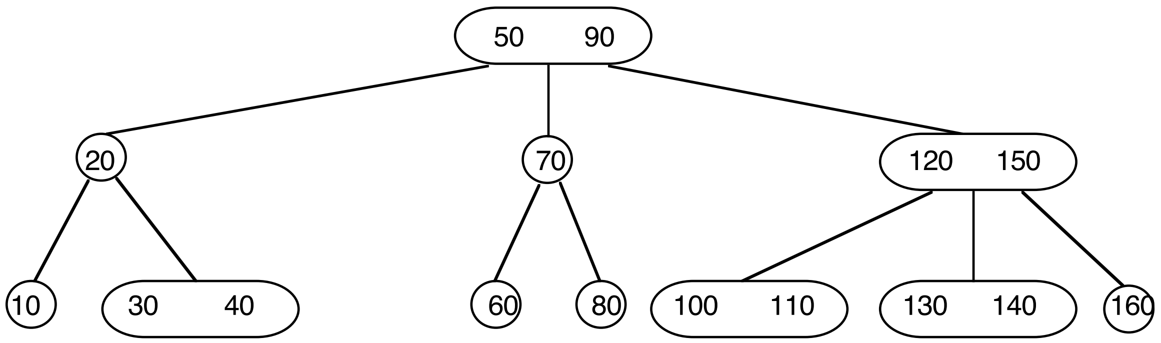


2.6 There would be two search keys.
2.6.1 The smaller search key must be greater than the left but less than the middle.
2.6.2 The greater search key would be greater than the middle but less than the right.

3 Rules for placing data items in the nodes of a 2-3 tree
3.1 A 2-node, which has two children, must contain a single data item whose search key is greater than the left child's search key(s) and less than the right child's search key(s), as figure 12-3a illustrates.
3.2 A 3-node, which has three children, must contain two data items whose search keys S and L satisfy the following relationships, as figure 12-3b illustrates: S is greater than the left child's search key(s) and less than the middle child's search key(s); L is greater than the middle child's search key(s) and less than the right child's search key(s).
3.3 A leaf may contain either one or two data items.



3.4 The items in a 2-3 tree are ordered by their search keys.
3.4.1 Figure 12-4 shows a 2-3 tree:



3.4.2 The following C++ statement would describe any node with a 2-3 tree:

```
class TreeNode
{
private:
        TreeItemType smallItem, largeItem;
        TreeNode *leftChildPtr, *midChildPtr, * rightChildPtr;

        // friend class-can access private class members
        friend class TwoThreeTree;
}; // end TreeNode
```

3.5 When the node contains only one data item, you place it in smallItem and use leftChildPtr and midChildPtr to point to the node's children.

3.5.1 To be safe, place a NULL in rightChildPtr.

3.6 You can traverse a 2-3 tree in sorted search-key order by doing the following inorder:

```
inorder(in ttTree:TwoThreeTree)
// Traverses the nonempty 2-3 tree ttTree in sorted search-key order

        if (ttTree's root node r is a leaf)
                Visit the data item(s)

        else if (r has two data items)
        {       inorder(left subtree of ttTree's root)
                Visit the first data item
                inOrder(middle subtree of ttTree's root)
                Visit the second data item
                inOrder(right subtree of ttTree's root)
        }

        else    // r has one data item
        {       inorder(left subtree of ttTree's root)
                Visit the data item
                Inorder(right subtree of ttTree's root)
} // end if
```

3.7 Searching 2-3 tree is similar to a binary search tree, like so:

```
retrieveItem(in ttTree:TwoThreeTree, in searchKey:KeyType,
        out treeItem:TreeItemType):boolean
// Retrieves into treeItem from a nonempty 2-3 tree ttTree the item whose search key
// equals searchKey. The operation fails if no such item exists. The function returns
// true if the item is found, false otherwise.

    if (searchKey is in ttTree's root node r)
    {       // the item has been found
            treeItem = the data portion of r
            return true
    }

    else if (r is a leaf)
            return false    // failure

    // else search the appropriate subtree
    else if (r has two data items)
    {       if (searchKey < smaller search key of r)
                    return retrieveItem(r's left subtree, searchKey, treeItem)
            else if (searchKey < larger search key or r)
                    return retrieveItem(r's middle subtree, searchKey, treeItem)
            else
                    return retrieveItem(r's right subtree, searchKey, treeItem)
    }

    else   // r has one data item
    {       if (searchKey < r's search key)
                    return retieveItem(r's left subtree, searchKey, treeItem)

            else
                    return retrieveItem(r's right subtree, searchKey, treeItem)
    } // end if
```

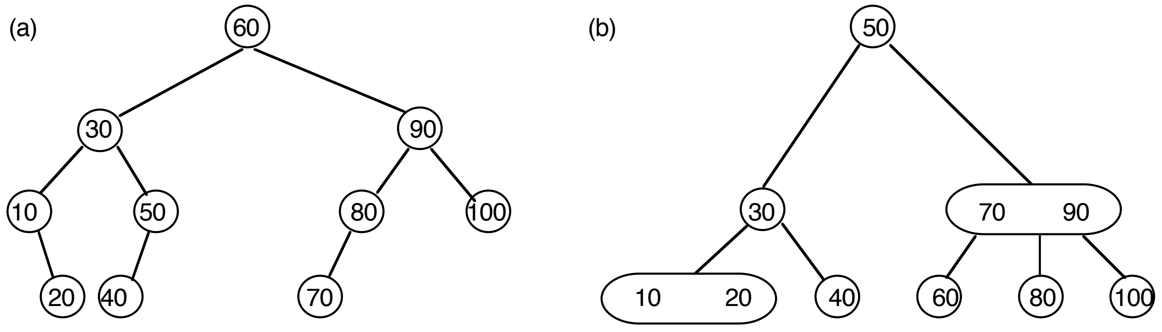
4 You can search a binary search tree and a 2-3 tree with about the same efficiency because:

- A binary search tree with N nodes cannot be shorter than $\lceil \log_2(N + 1) \rceil$
- A 2-3 tree with N nodes cannot be taller than $\lceil \log_2(N + 1) \rceil$.
- A node in a 2-3 tree has at most two items

5 However, searching a 2-3 tree is not more efficient than searching a binary search tree.

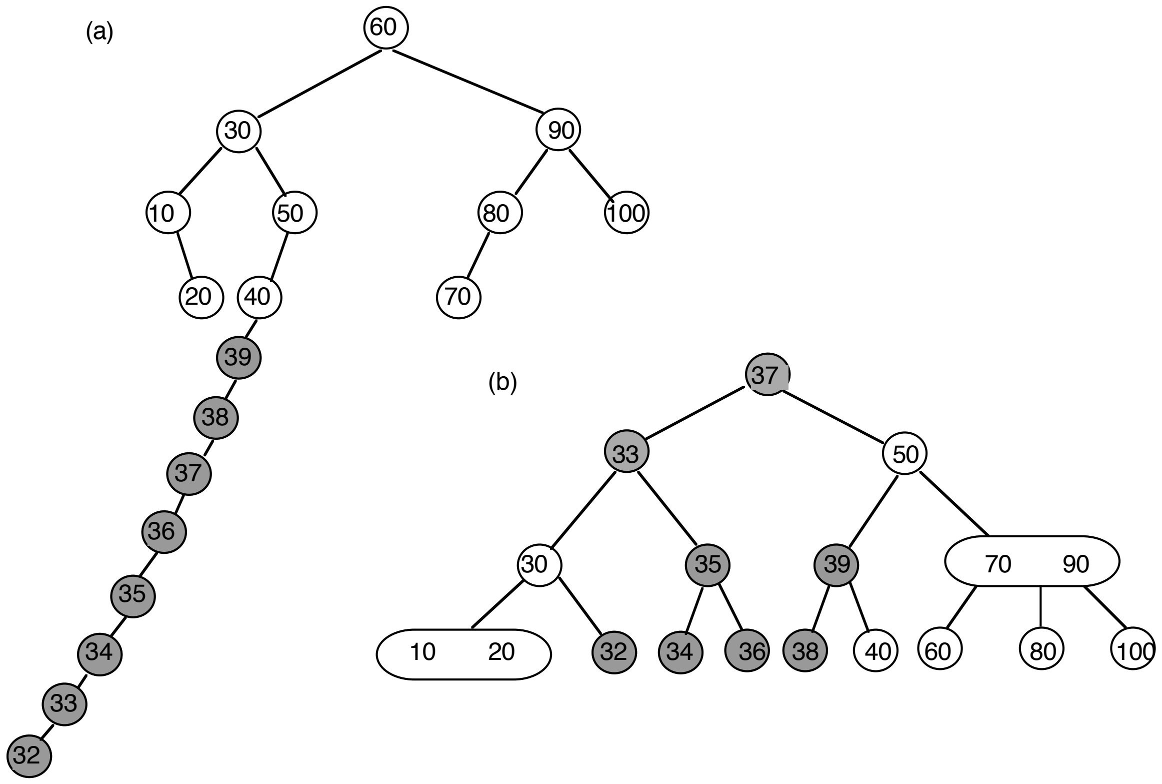
5.1 The problem is that while the 2-3 is at a minimum height, the advantage is offset by the extra comparisons needed to be done.

5.2 As figure 12-5 shows:



5.3 With multiple insertions, the binary search tree would have difficulty retaining its shape.

  5.3.1 However, the 2-3 tree would not have that difficulty as figure 12-6 shows:

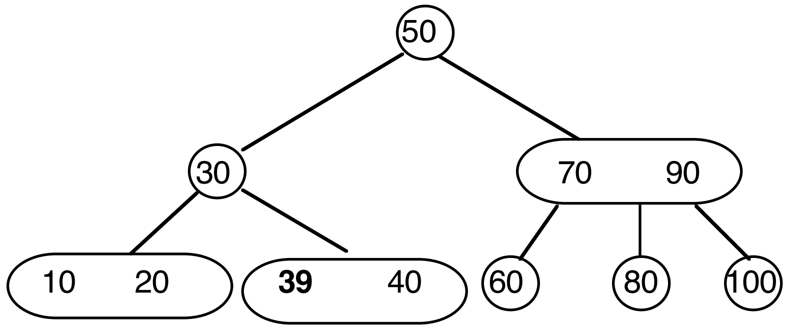


  5.3.2 The binary search tree in 12-6a has the new values keep increasing the height.

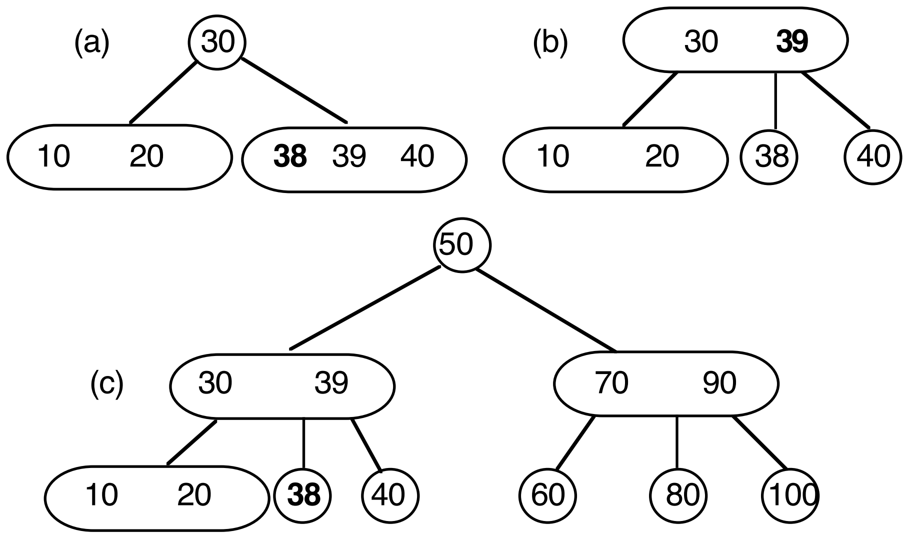
  5.3.3 The 2-3 tree would spread out the values as in 12-6b since each node can have either 2 or 3 children.

  5.3.4 Let's trace the inserts for 2-3 tree:

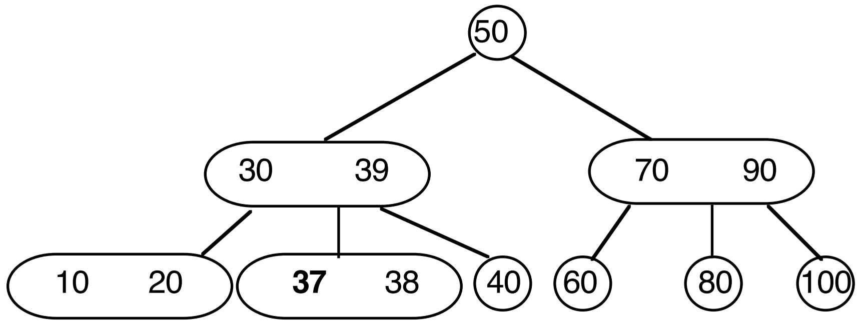
5.3.4.1 Insert 39: Find the proper place for inserting the number.
5.3.4.2 In this case, it is a leaf node and the value of 39 goes there like figure 12-7:



5.3.5 Insert 38: find a place for the 38 that is also a leaf.
5.3.5.1 But putting in three key values in one node is a problem.
5.3.5.2 You would take the middle value and move it up to the parent like figure 12-8 shows:



5.3.6 Insert 37: That value is fairly easy since the leaf node only have one key value -- Figure 12-9 shows it:



5.3.7 Insert 36: This is involved.

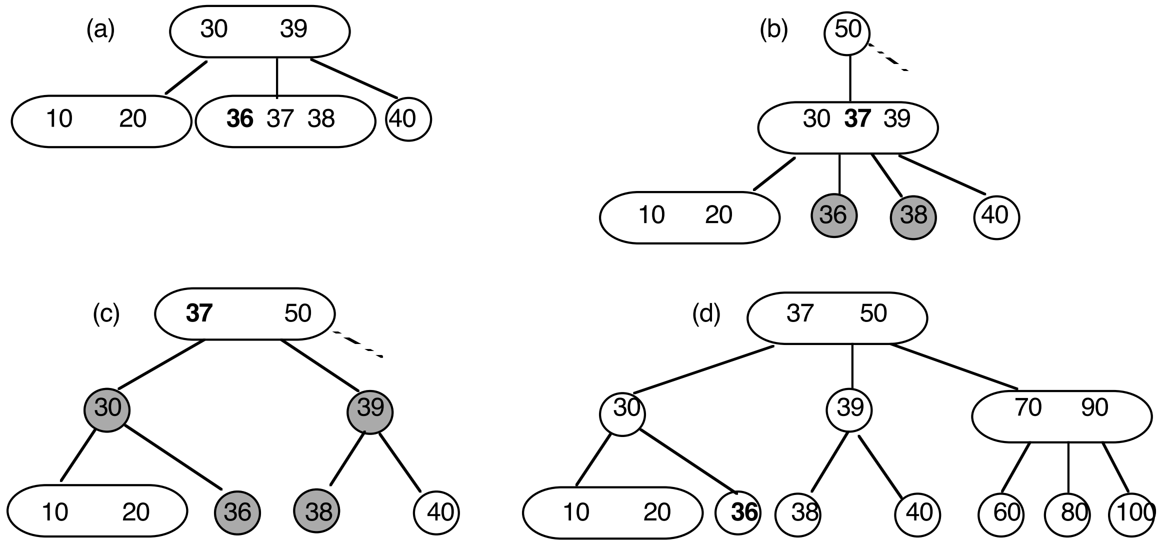
5.3.7.1 Conceptually we start with 3 values at a leaf node then expand it. Figure 12-10 shows it.

5.3.7.2 12-10a has the 3 values.

5.3.7.3 Then we break the node into two 2-nodes and move the 37 value up one level at 12-10b.

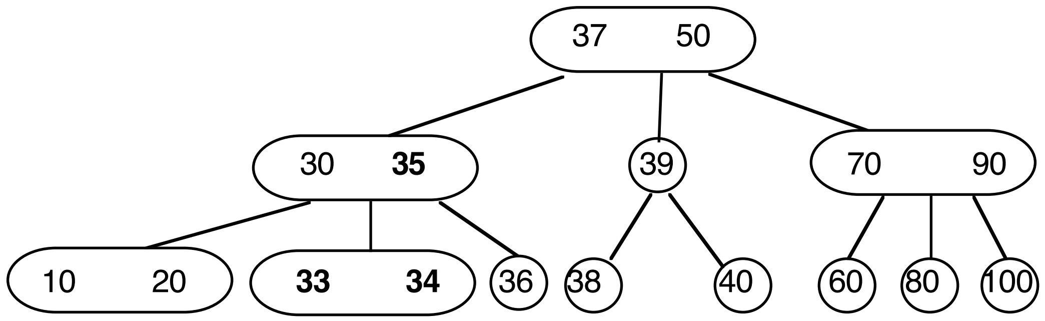
5.3.7.4 Then the node with 37 is broken into two at 12-10c and 37 moves up to the root.

5.3.7.5 12-10d shows the final 2-3 tree.



5.3.8 Insert 35, 34, and 33. It is similar to the previous inserts.

5.3.9 Figure 12-11 on page 660 shows the final result:



6 The insertion algorithm: To insert item I into a 2-3 tree, first locate the leaf at which the search for i would terminate.

6.1 If you insert at the leaf and there are only 2 values, you are done.

6.2 But if you have 3 values, you need to split the node into nodes n1 and n2.

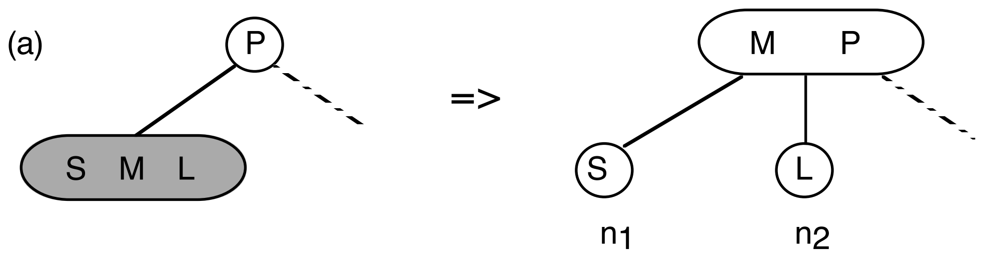
6.3 Figure 12-12a shows the split.

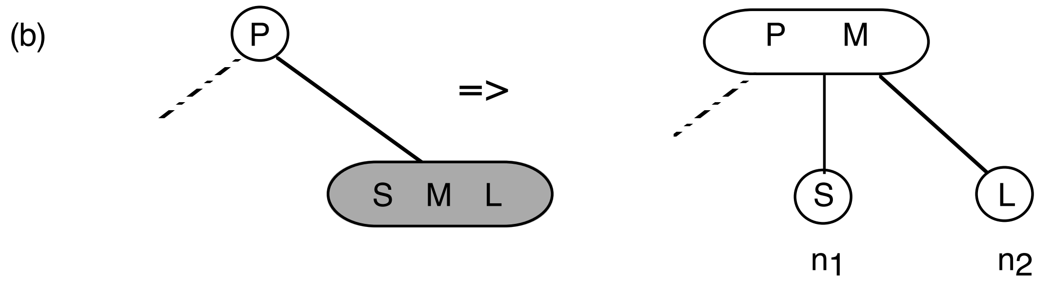
6.4 You place the small item S into n1, place the largest item L into n2, and move the middle item M up to the original leaf's parent.

6.5 Nodes n1 and n2 then become children of the parent.

6.6 If the parent now has only three children (and contains two items) you are done.

6.7 However, if the parent has four children (and has three items), you must split it.



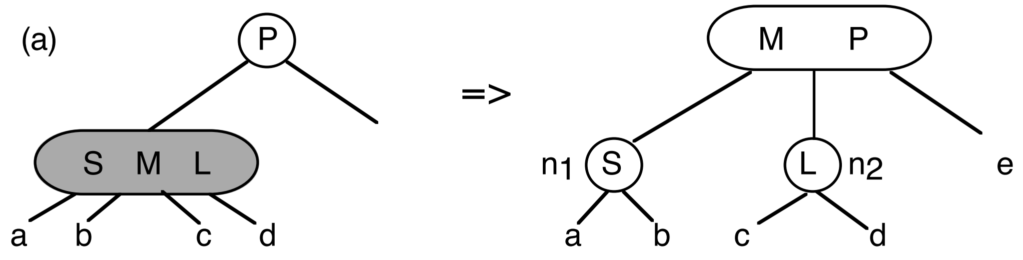


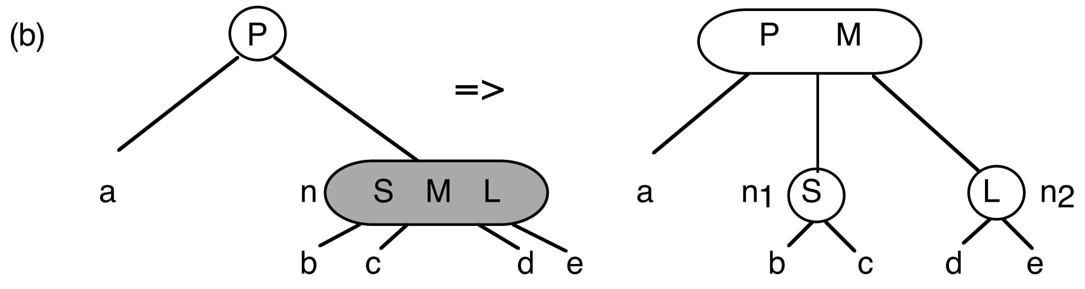
7 To split an internal node n that has three items by using the process just described with a leaf but you must also take care of n's four children.

7.1 Figure 12-13 shows splitting n into n1 and n2, place n's smallest item S into n1, attach n's two leftmost children to n1, place n's largest item L into n2, attach n's two rightmost children to n2, and move n's middle item M up to n's parent.

7.2 After this, the process of splitting the node and moving an item up is done recursively.

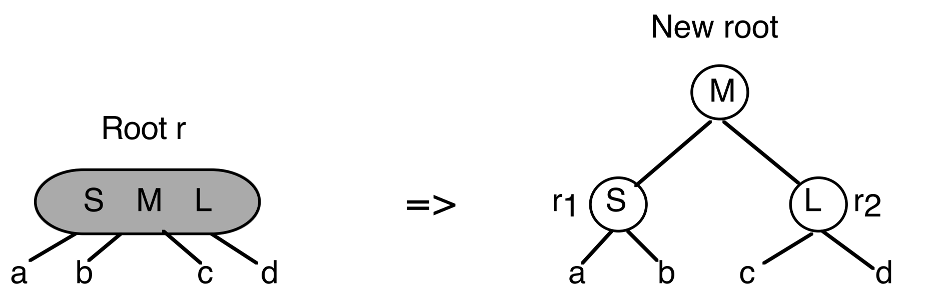
7.3 The tree's height is not increased as long as there is a node in the path that has only one value.





7.4 If the root needs to be split, you would split it into r1 and r2 like any other internal node.

7.4.1 Figure 12-14 shows it:



7.4.2 The algorithm is:

```
insertItemItem(in ttTree:TwoThreeTree, in newItem:TreeItemType)
// Inserts newItem into a 2-3 tree ttTree whose items have distinct search keys
// that differ from newItem's search key

        Let sKey be the search key of newItem
        Locate the leaf leafNode in which sKay belongs
        Add newItem to leafNode

        if (leafNode now has three items)
                split(leafNode)

        split(inout n:TreeNode)
        // Splits node n, which contains 3 items. Note: if n is not a leaf, it has
        // 4 children. Throws TreeException if node allocation fails
        if (n is the root)
                Create a new node p (refine later to throw exception if
                                        allocation fails)
        else
                Let p be the parent of n
```

7.4.3 Replace node n with two nodes, n1 and n2, so that p is their parent

7.4.3.1 Give n1 the item in n with the smallest search-key value
7.4.3.2 Give n2 the item in n with the largest search-key value

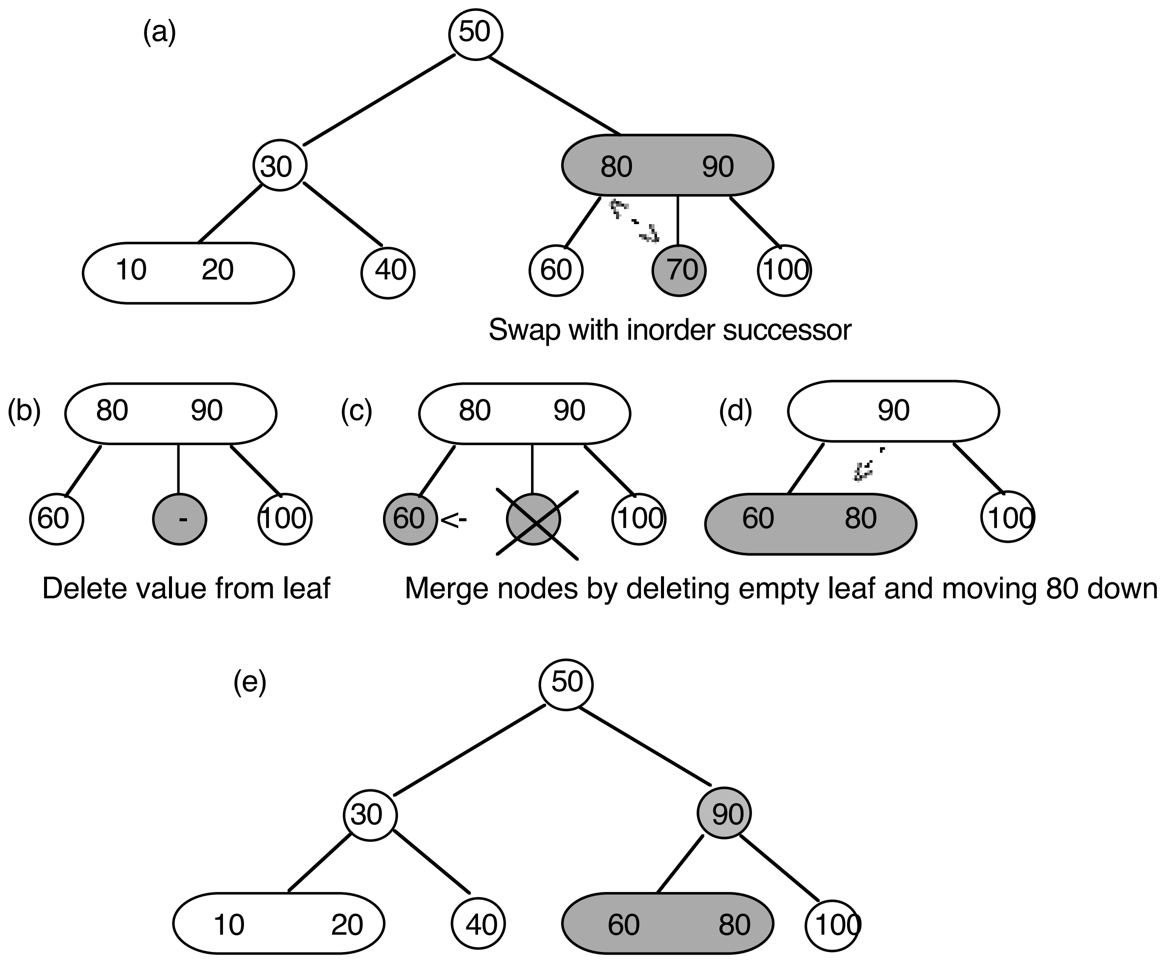
```
        if (n is not a leaf)
        {       n1 becomes the parent of n's two leftmost children
                n2 becomes the parent of n's two rightmost children
        } // end if
```

7.4.4 Move the item in n that has the middle search-key value up to p

```
if (p now has three items)
        split(p)
```

7.5 Deleting from a 2-3 tree.

- 7.5.1 The deletion strategy is the inverse of its insertion strategy.
- 7.5.2 Figure 12-15 on page 664 shows deleting 70:



7.6 Delete 100 is fairly easy since it is a leaf node.

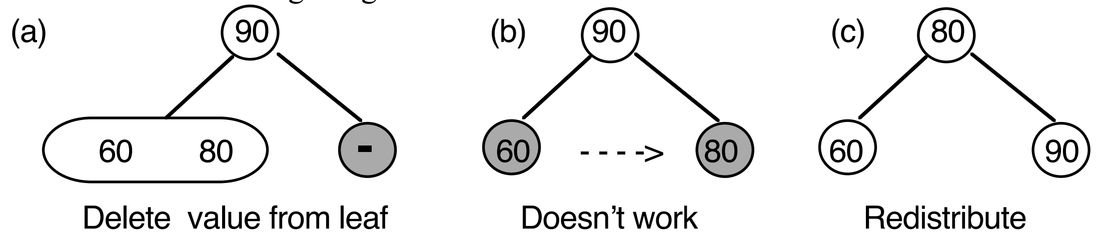
- 7.6.1 However, you need to redistribute the values.
- 7.6.2 However, if you move the 80 over to the node that had the 100 then 2-3 tree would not be valid – the 80 would be to the right of the parent of 90.
- 7.6.3 You need to redistribute the values – the 90 would move down to the right and the 80 replaces the parent value.
- 7.6.4 Figure 12-16 shows the redistribute.

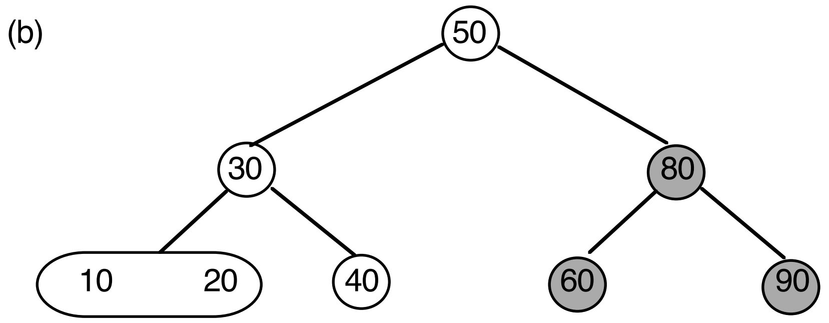
7.7 If you delete 80, then you need to merge the 60 and 90.

- 7.7.1 However, the parent becomes empty and you need to merge the root node with the left node the move the 60 and 90 node over.
- 7.7.2 The root becomes empty and it can be deleted and the node with the node with 30 and 50.

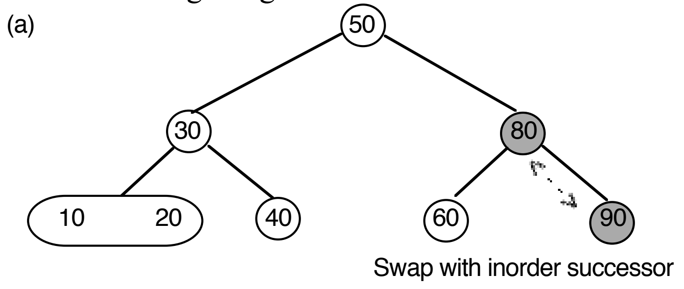
7.7.3 Figure 12-17 shows it.

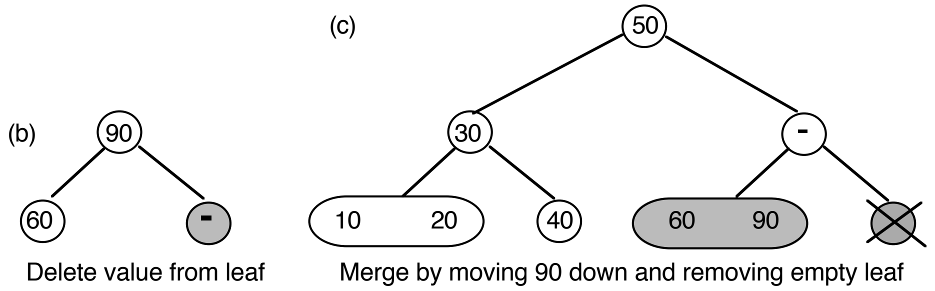
7.8 The following is figure 12-16.

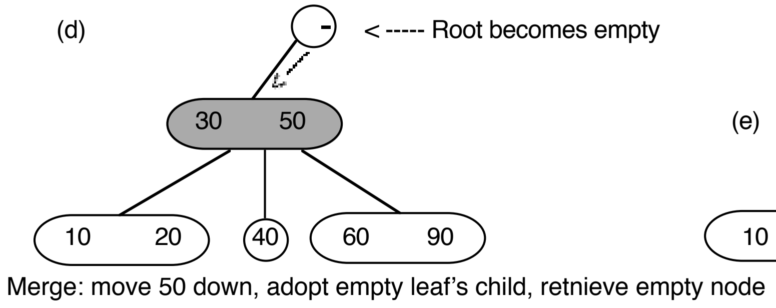


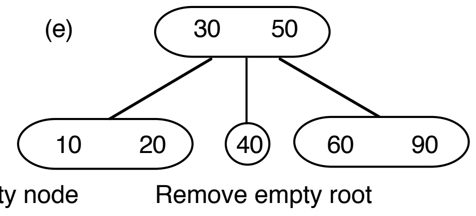


7.9 The following is figure 12-17






(d)
-
< ----- Root becomes empty
30 50
10 20
40
60 90
Merge: move 50 down, adopt empty leaf's child, retnieve empty node



8 In deleting from a 2-3 tree, the tree would remain as a complete tree.

8.1 When you delete the node that has i, node, you must first find the n node that has it.

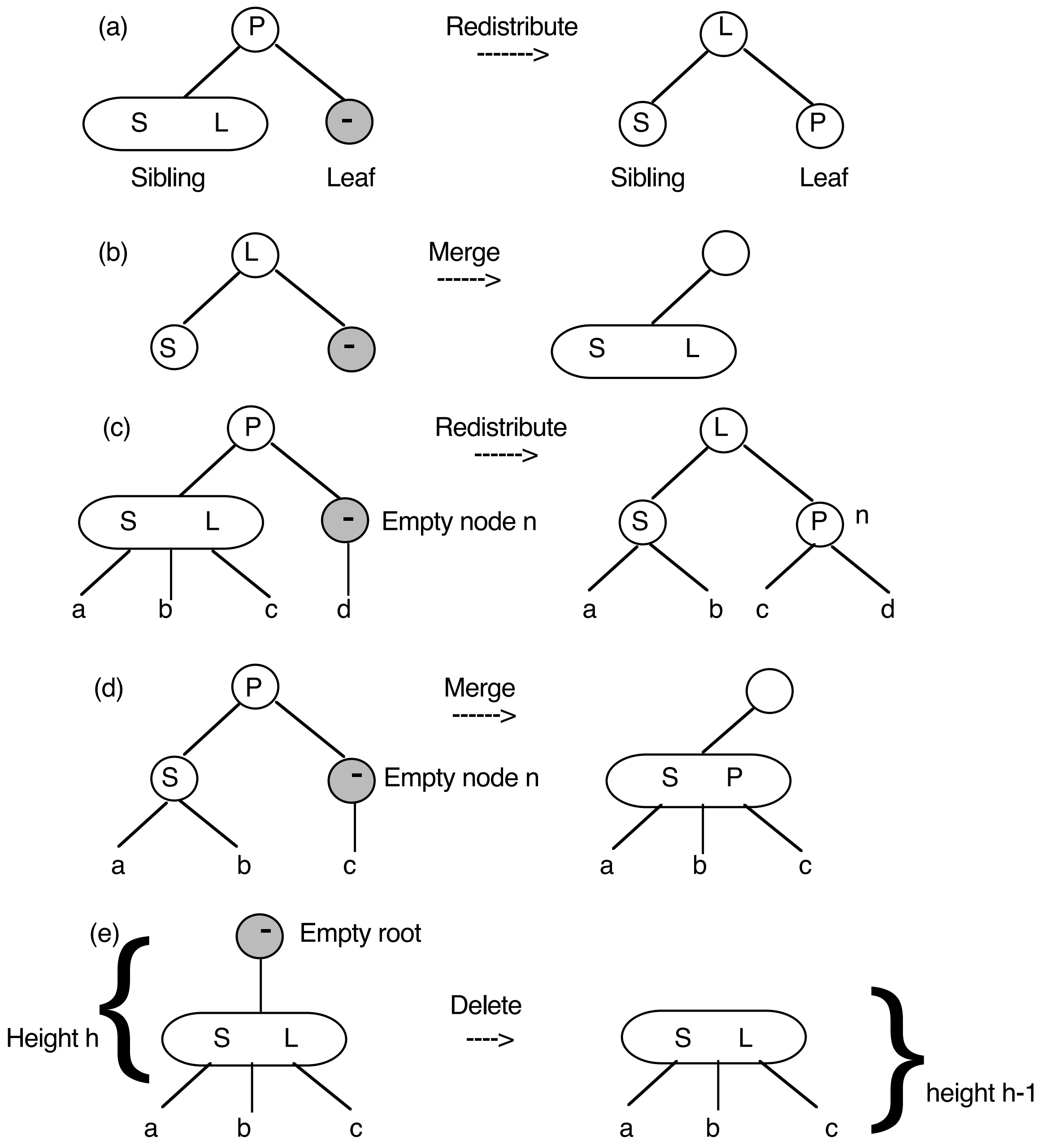
8.2 If n node is not a leaf, you find i's inorder successor.

8.3 If the leaf contains an item in addition to i, you just delete i and you are done.

8.4 But if there is no other and the node becomes empty, then more work needs to be done.

8.5 First check the siblings of the now-empty leaf.

   8.5.1 If a sibling has 2 items, you redistribute them among the sibling, the empty leaf, and the leaf's parent as figure 12-19a shows.



8.6 If no sibling has two items, you merge the leaf with an adjacent sibling by moving an item down from the leaf's parent into the sibling and removing the leaf as figure 12-19b.

- 8.6.1 If n has a sibling with 2 items and three children, you redistribute the items among n, the sibling, and n's parent.
- 8.6.2 You also give n one of its sibling's children as figure 12-19c would show.

8.7 If n has no sibling with two items, you merge n with a sibling as figure 12-19d (i.e. move an item down from the parent and let the sibling adopt n's one child).

8.7.1 If the merging continues with the root having no value, then remove the root and the tree's height decreases by one as figure 12-19e shows.

8.8 The following would be the high-level algorithm for deleting a node with the 2-3 tree:

```
deleteItem(in ttTree:TwoThreeTree, in searchKey:KeyType)
                throw TwoThreeTreeException
// Deletes from the 2-3 tree ttTree the item whose search key equals searchKey.
// Throws TwoThreeTreeException if no such item exits

        Attempt to locate item theItem whose search key equals searchKey

        if (theItem is present)
        {       if (theItem is not in a leaf)
                Swap item theItem with its inorder successor,
                        which will be in a leaf leafNode

                // the deletion always begins at a leaf
                Delete item theItem from leaf leafNode

                if (leafNode now has no items)
                        fix(leafNode)
        }
        else
                Throw a TwoThreeException

fix(in n:TreeNode)
// Completes the deletion when node n is empty by either removing the root,
// redistributing values, or merging nodes. Note: if n is internal, it has one child

        if (n is the root)
                Remove the root
        else
        {       Let p be the parent of n
                if (some sibling of n has two items)
                {       Distribute items appropriately among n, the sibling, and p
                        if (n is internal)
                                Move the appropriate child from sibling to n
                }

                else // merge the node
                {       Choose an adjacent sibling s of n
                        Bring the appropriate item down from p into s

                        if (n is internal)
```

```
                    Move n's child to s

                Remove node n

                if (p is now empty)
                    fix(p)
            } // end if
        } // end if
```

9 The implementation of the above algorithm can be challenging.

9.1 The binary tree that is balanced is the most efficient to search.

9.1.1 However, it is difficult to maintain as balanced with multiple deletions and insertions.

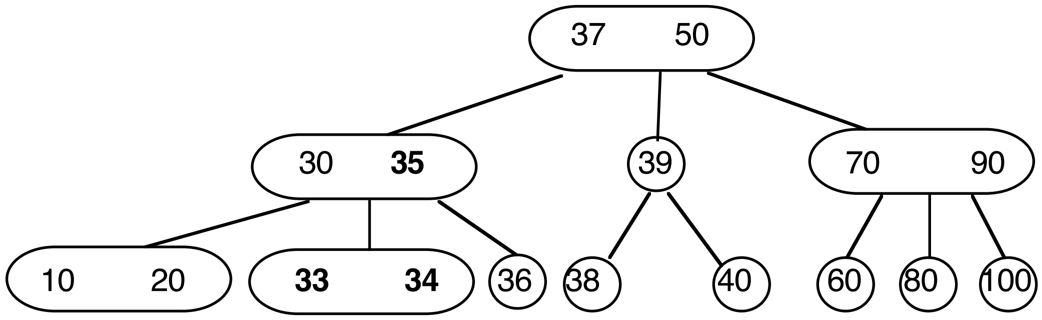
9.1.2 The 2-3 tree can be easier to maintain as a complete and thus balanced tree.

9.1.3 The 2-3 tree is a compromise.

9.2 If a 2-3 is so good, then why not have a 2-3-4 tree?

9.2.1 With a 2-3-4 tree of height 3 in figure 12-20 on page 671 has the same items as the 2-3 tree in figure 12-6b.

9.2.2 It is easier to insert and delete into a 2-3-4 tree.

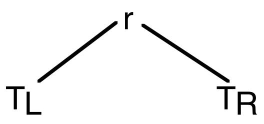


9.3 T is a 2-3-4 tree of height h if:

- T is empty (a 2-3-4 of height 0)

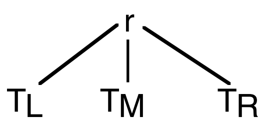
Or

- T is of the form

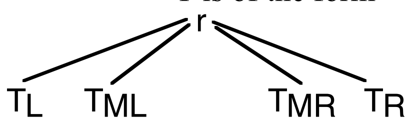


Or

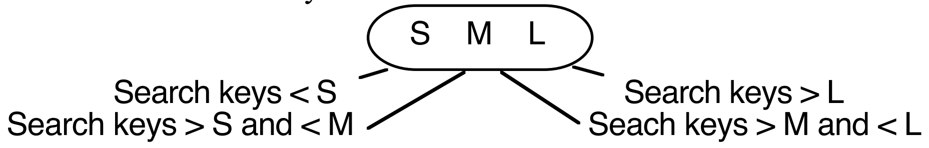
- T is of the form



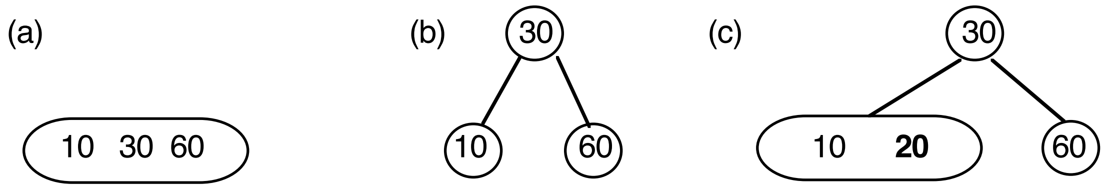
- T is of the form



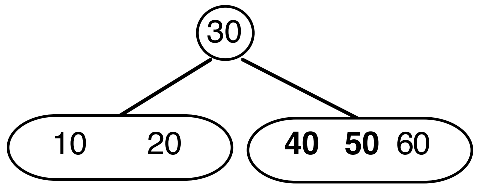
9.4 The search keys must be:



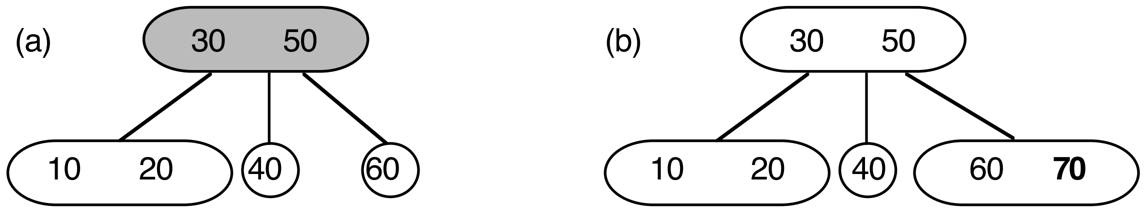
9.4.1 Figure 12-22 shows inserting a 20:



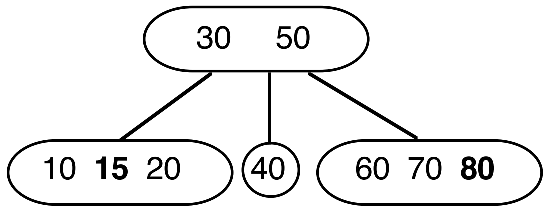
9.4.2 Figure 12-23 shows inserting 50 and 40:



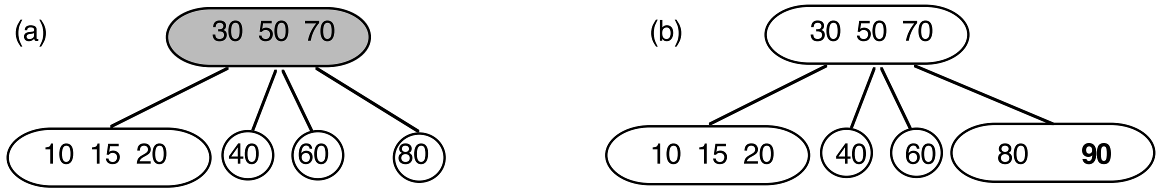
9.4.3 Figure 12-24 shows inserting 70:



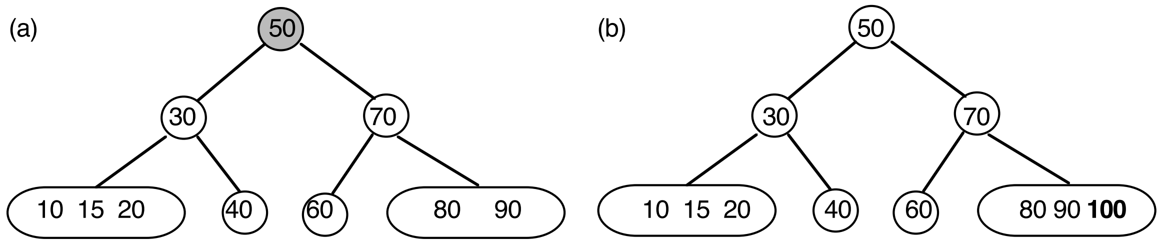
9.4.4 Figure 12-25 shows inserting 80:



9.4.5 Figure 12-26 shows inserting 90:



9.4.6 Figure 12-27 shows inserting 100:

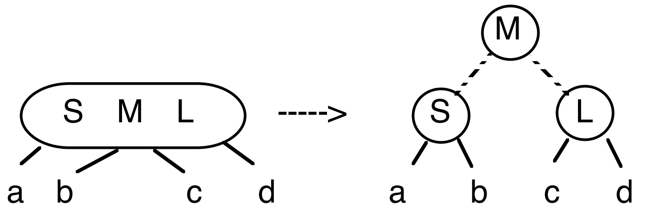


9.5 Deletion from a 2-3-4 tree is similar to a 2-3.

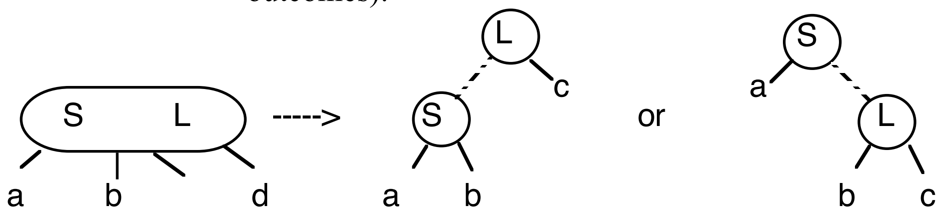
- 9.5.1 The 2-3-4 tree is attractive because it is balanced and its insertion and deletion operations use only one pass from root to leaf.
- 9.5.2 But the 2-3-4 tree takes up more storage than a binary search tree.
- 9.5.3 But using a binary search tree is problematic since it might not be balanced.
- 9.5.4 However, you can use a special binary search tree called a “red-black” tree to represent a 2-3-4 tree and thus not have the storage overhead.

9.6 The idea is to represent each 3-node and 4-node in a 2-3-4 tree by an equivalent binary tree.

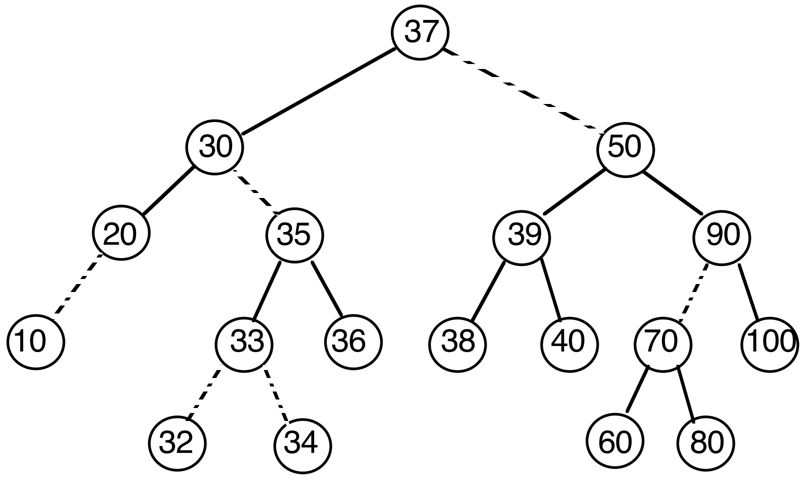
- 9.6.1 To distinguish 2-nodes, 3-nodes, and 4-nodes, you use red and black child pointers.
- 9.6.2 You start with all child pointers in a 2-3-4 tree be black then use red pointers when you split 3-nodes and 4-nodes.
- 9.6.3 Even with pointer colors, a node in a red-black tree needs less storage that a node in a 2-3-4 tree
- 9.6.4 Figure 12-31 shows how for a 4-node (dashed line is red and black is solid):



9.6.5 Figure 12-32 shows how to split a 3-node (there are two possible outcomes):



9.6.6 Figure 12-33 shows a red-black tree that represents the 2-3-4 tree in figure 12-20:



9.6.7 The C++ statements for the red-black are:

```
enum Color {RED, BLACK};

class TreeNode
{
private:
        TreeItemType Item;
        TreeNode       *leftChildPtr, *rightChildPtr;
        Color            leftColor, rightColor;

        friend class RedBlackTree;
}; // end TreeNode
```

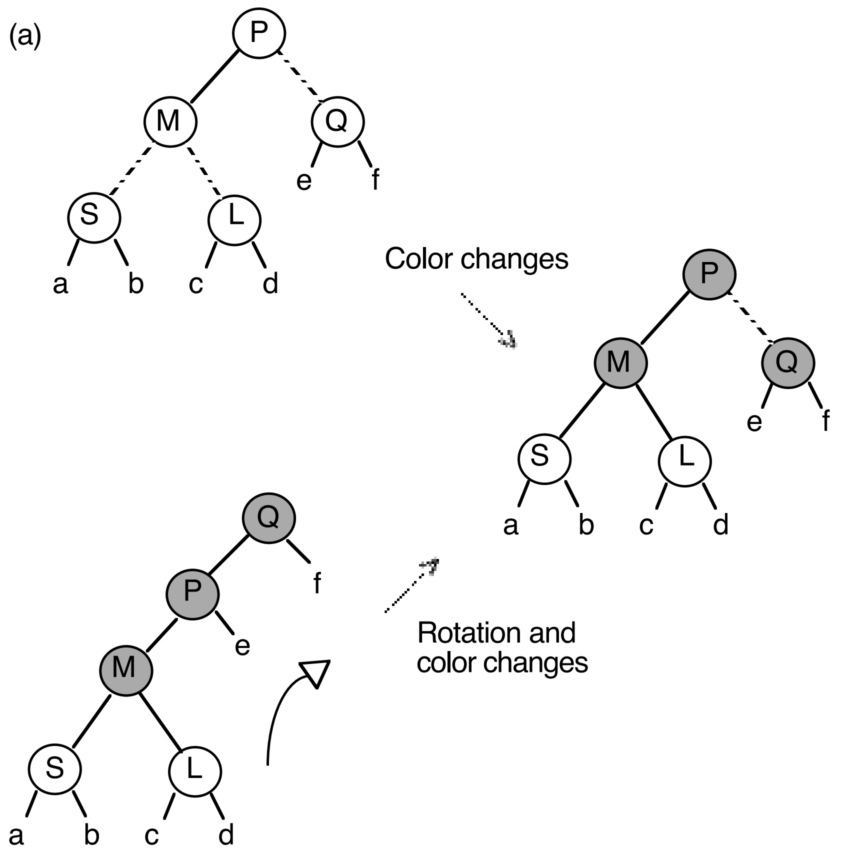
9.7 With searching a red-black tree, just ignore the color of the pointers.

9.7.1 With inserting, you need to pay attention to the colors.

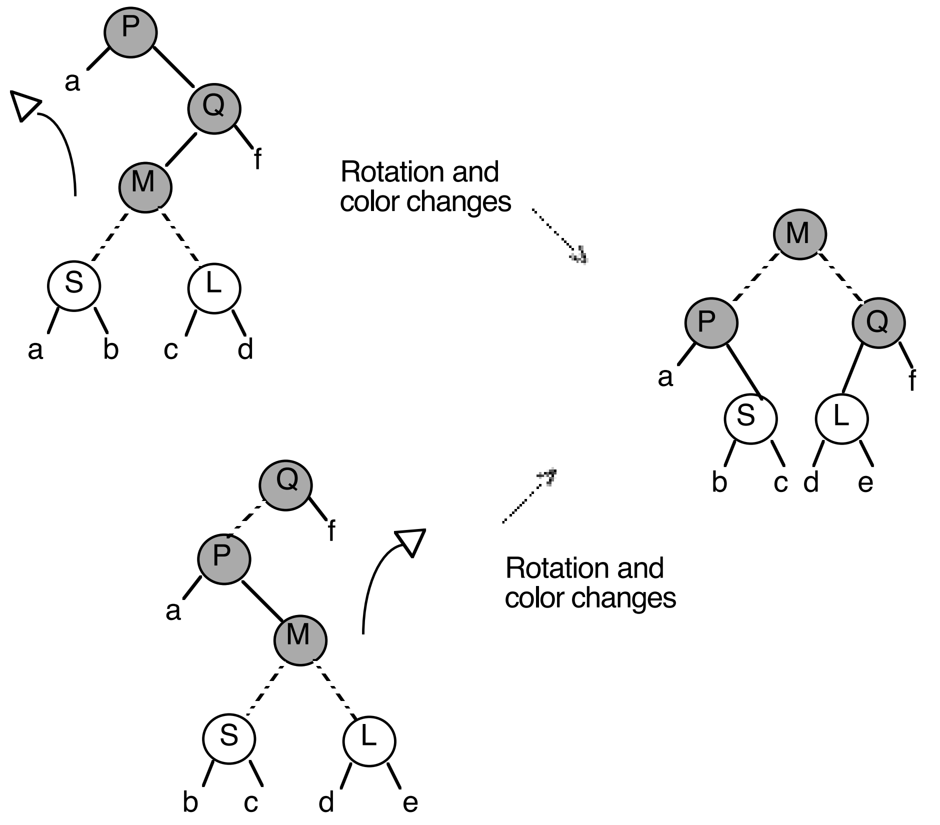
9.7.2 Though splitting and merging are simplified by the fact that you can change the color of the pointers.

9.7.3 Figure 12-36 shows the color changes and any needed rotations to maintain a binary search tree:

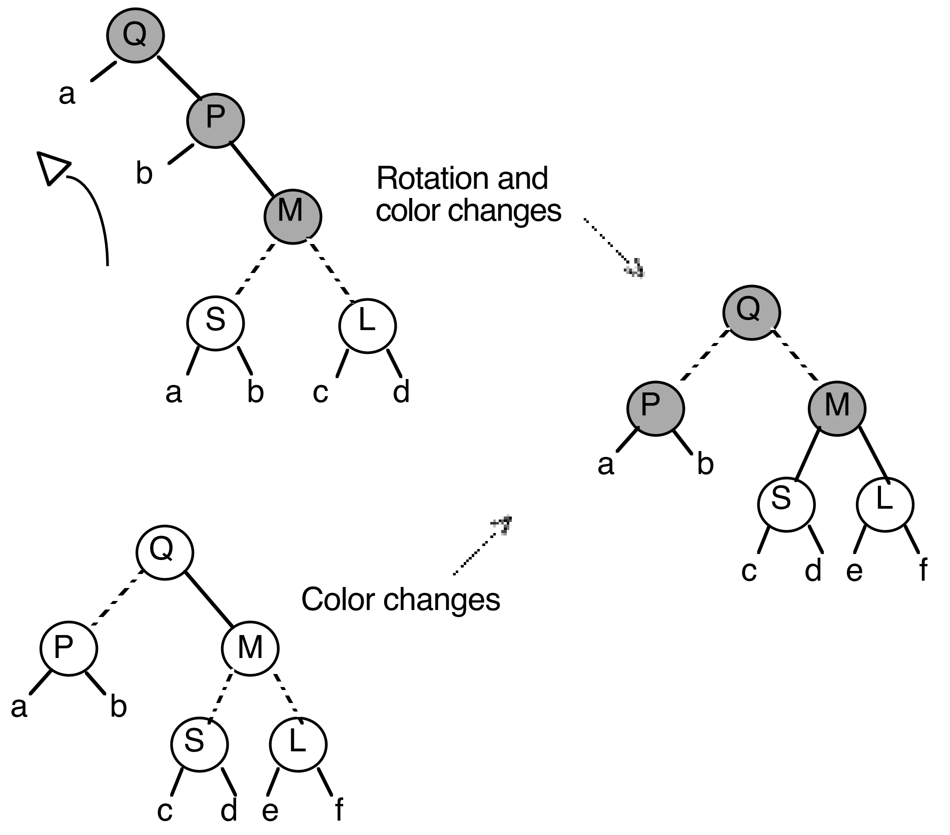
(a)


P
M
Q
S
L
a
b
c
d
e
f
Color changes
Rotation and
color changes

(b)

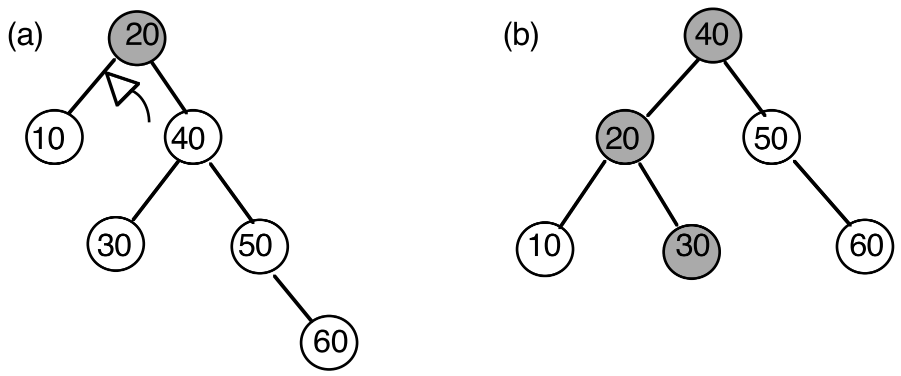

P
Q
M
S
L
a
b
c
d
e
f
Rotation and
color changes
Rotation and
color changes

(c)



9.8 Brief note about the AVL tree – it was named after its inventors: Adel'son Vel'skii and Landis.

- 9.8.1 It is a balanced binary search tree where the left and right subtrees of any node differ no more than 1.
- 9.8.2 Trying to maintain a minimum height binary search tree can take too much effort.
- 9.8.3 The AVL tree is a compromise.
- 9.8.4 After an insertion or deletion, it checks to see if the tree is balanced.
- 9.8.5 If not, then it would rotate the nodes to achieve a balanced tree (but NOT necessarily a complete tree).
- 9.8.6 An AVL tree will be close to the minimum height of a tree and thus is a good implementation.
- 9.8.7 The remaining figures show how the rotation can occur such as Figure 12-38 (note that the resulting tree is NOT a complete tree):



10 The rest you can see on your own.