Module 10

More Efficient Search Trees

Objectives:

1. Understand the concept of using 2-nodes, 3-nodes, and 4-nodes, and how items are stored in 2-3 trees and 2-4 trees. You don't need to be familiar with the add or remove algorithms.

2. Know the rules for red-black trees.

3. Know the difference between bottom-up insertion and top-down insertion, and know which one is more efficient.

4. Know that for top-down insertions imbalances must be anticipated and corrected before they occur.

5. Know what sentinels are and why they are used.

6. Know the rules for an AA Tree, and that it's implementation is typically recursive.

7. Understand how the skew and split operations work for an AA tree.

8. Know what a B tree is, and why it is useful.

9. Be familiar with the basic strategies for adding and removing items in a B tree.

Reading:

Chapter 28 (Balanced Search Trees), sections 28.13 - 28.41 (2-3, 2-4 trees, red-black trees).

Assignment:

No assignment for Module 10.

Quiz:

Take the quiz for Module 10.

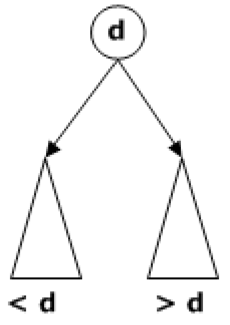
Lab:

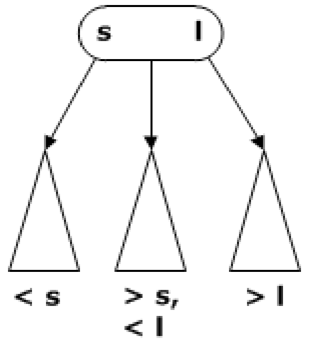
No lab for this module.

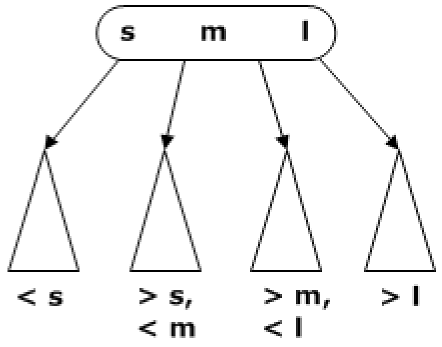
2-3 Trees and 2-4 Trees

I'm not going to cover these types of binary search trees in depth, but I do think it is important that you are aware of them. A 2-3 tree consists of two types of nodes: 1) 2-nodes, which store a single data item and references to left and right child nodes, and 2) 3-nodes, which store two data items and references to left, middle, and right child nodes. The left subtree of a 3-node contains items that are smaller than the smallest item in the 3-node. The right subtree contains items that are greater than the largest item in the 3-node. The middle subtree contains items that are greater than the smallest item in the 3-node, but less than the largest item in the 3-node.

A 2-4 tree is an extension of a 2-3 tree, and contains three types of nodes: 1) 2-nodes, 2) 3-nodes, and 3) 4-nodes. The 2-nodes and 3-nodes are the same as described above for 2-3 trees. The 4-node contains three data items, which I'll refer to as *s*, *m*, and *l* (same as in the book) and references to four child nodes, which I'll refer to as left, middle-left, middle-right, and right. The diagrams in Figure 10.1, reproduced from the book, summarize how the items are stored for the various types of nodes.







2-node	3-node	4-node

Figure 10.1. Illustrations of a 2-node, a 3-node, and a 4-node.

The advantage of using these trees is that they are always completely balanced - all the leaf nodes are always at the same level. As a result, 2-3 trees tend to be shorter than AVL trees, and 2-4 trees tend to be shorter than 2-3 trees. Recall from Module 9 that finding an item (and hence, adding and removing items) in an AVL tree has a running time proportional to the height of the tree. Therefore, the shorter the tree, the shorter the running time should be. These trees are also easier to keep balanced than AVL trees.

Red-black Trees

Recall from the last module that AVL trees require two passes for additions and removals. The first pass goes from the root to the insertion/deletion point to add or remove an item. The second pass goes from the insertion/deletion point back up the tree to the root, rebalancing the tree as needed. The two-pass mechanism is not intentional; rather it is simply a by-product of using recursion. If you begin at the root and recursively descend the tree, you always have to climb back up the tree to resolve all the suspended recursive method calls. This allows for relatively simple coding, but making two passes it is not very efficient for large trees.

Remember, though, that any problem that can be solved recursively can also be solved iteratively. Thus, it should be possible to write a balanced binary search tree using loops instead of recursion. Using an iterative solution, it is also possible to add or remove an item and maintain the balance using a single pass down the tree from the root to the insertion/deletion point.

The red-black tree is an example of a binary search tree that uses iterative algorithms for item additions and removals. In the most efficient implementation, it uses a single, top-down pass to add or remove an item, and correct any imbalances that occur. The trade-off is that the code for a red-black tree is very complex and difficult to write. Red-black trees consist solely of 2-nodes - each node stores a single data item and references to left and right child nodes.

All red-black trees must obey the following rules:

1. Every node is colored either red or black.
2. The root node is always black.
3. Red nodes may only have black child nodes (i.e., there cannot be two consecutive red nodes in any path).
4. Every path from a given node to a null link must contain the same number of black nodes.
5. By convention, null child nodes are considered to be black.

Top-down Insertion

The top-down insertion algorithm ensures that an item is added to the proper location in the tree in a single pass from the root down to the insertion point. On the way down, potential imbalances are anticipated and avoided by using a combination of rotations and color flips. The rotations are essentially the same as those used for the AVL tree. Color flips involve swapping the colors of a parent node and its child nodes.

The algorithm for top-down insertion is as follows:

1. Begin searching for the insertion point at the root. On the way, if a parent node is encountered that has 2 red child nodes, which is a violation of rule #3, do one of the following:

    a. If the parent's parent is black, do a color flip (change the color of the parent node to red and the colors of the child nodes to black. If the parent happens to be the root of the tree, change its color back to black to avoid violating rule #1.

    b. If the parent's parent is red, do a color flip, but then perform either a single rotation (if the node is inserted on the outside) or a double rotation (if the node is inserted on the inside) to avoid violating rule #3.

2. Once the new node is inserted, set the color of the new node to red. If the new node's parent is black nothing further needs to be done. If the parent of the new node is red perform either a single or a double rotation to avoid violating rule #3.

The algorithm itself sounds very simple, but the code needed to implement it is not. We'll look at the code a bit later. The algorithm runs in O(log N) time, as does the insertion algorithm for an AVL tree, but it is more efficient because it only uses a single pass down the tree. Below is an example that covers all the important points.

Top-down Insertion Example

Figure 10.2 (starting on the next page) illustrates an example of top-down insertion. In this example, a new red-black tree is built beginning with an empty tree.

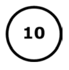
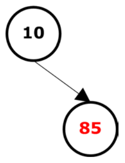
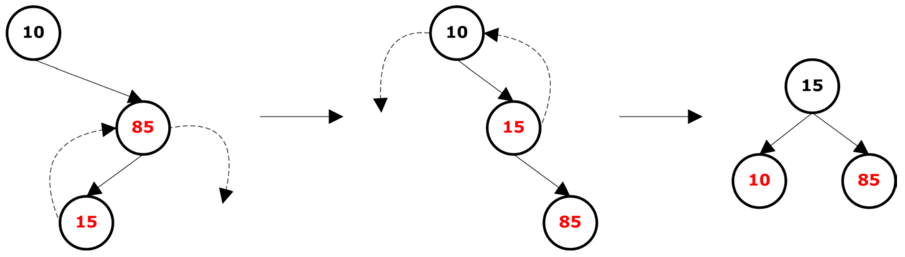
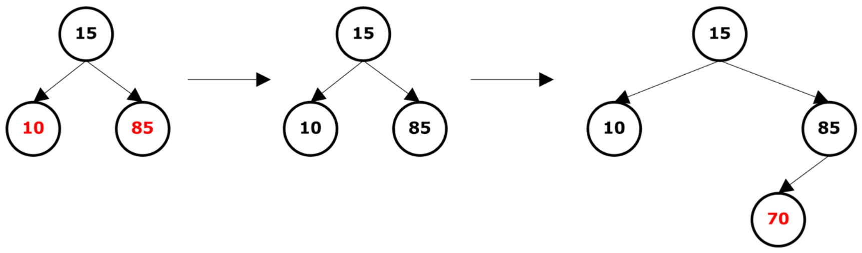
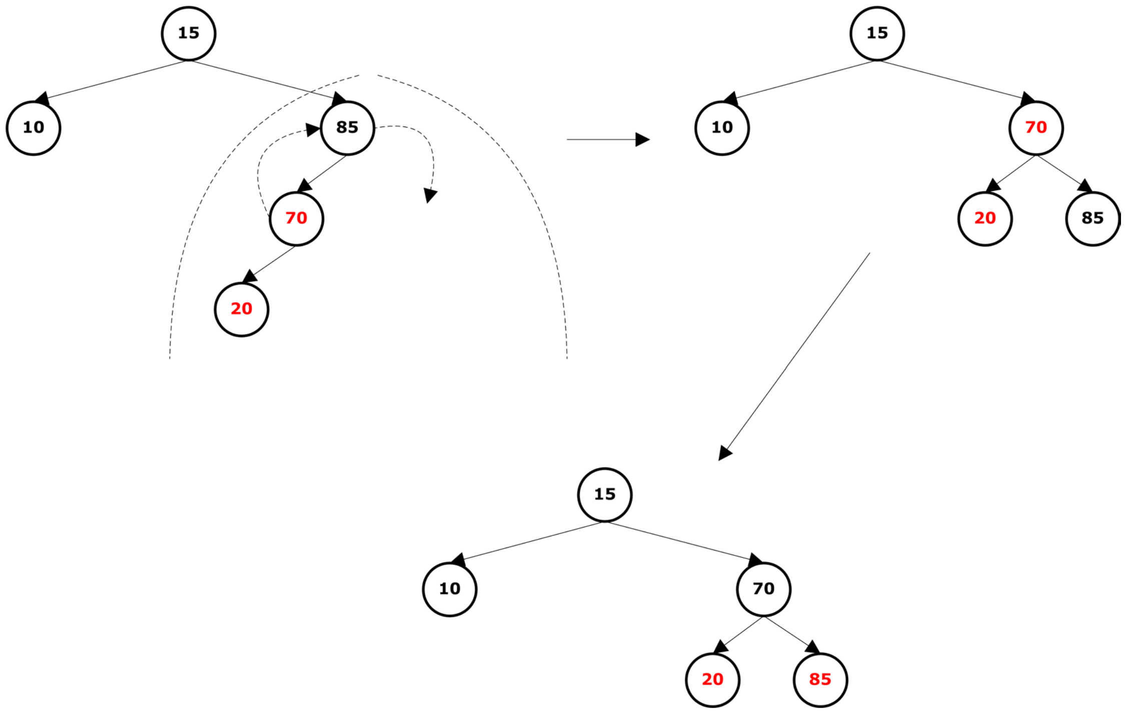
Add 10	
Nothing special here — just add the node and color it black.	
Add 85	
Adding the second item is also easy — just put it in the proper position and make sure it's red.	
Add 15	
When we add the 15, we have 2 consecutive red nodes, which is a violation of rule #3. We can't simply change colors, because we would end up violating rule #4 since the root's left subtree is empty. A single rotation will not work because we would still have 2 consecutive red nodes. So, we must do a double rotation to balance the tree. The mechanics are the same as the Case 3 double rotation for the AVL tree. The new root must be colored black, and the old root must be colored red.	
Add 70	
As we search for the insertion point, we find a parent node (the root node) that has two red children. We know that if we need to add the new item to one of the red children we will violate rule #3, so in anticipation of a possible rule violation we swap the colors of the parent node and its children. The root node must be recolored to black to prevent a violation of rule #2. Doing the color swap ensures that when we add the new red node, we will not violate rule #3. In this case, no rotations are necessary.	
Add 20	 15 10 85 70 20 15 10 70 20 85 15 10 70 20 85
	Adding a 20 results in a violation of rule #3. A single rotation around the 85 node, which is equivalent to an AVL tree Case 1 rotation, followed by a color swap, will fix the violation.

Figure 10.2. Example of top-down insertion in a red-black tree.

Top-down Removal

Removing an item from a red-black tree is quite complicated to begin with, and doing so in a single pass from the root to the removal point is even more complex. When the node to be removed has two children, we will follow the removal algorithm for the generic binary search tree - the item in the node will be re-placed with the minimum item in that node's right subtree, and the node containing the minimum item in the node's right bustree will be removed. However, an item's removal could introduce a violation of either rule #3 or rule #4, and we have to make sure neither happens. Removing a black node will introduce a violation of rule #4, since it will change the number of black nodes on one path in the tree. Removing a black node can also violate rule #3 if it happens to bring two consecutive red nodes together. Removing a red node, however, will avoid both problems. Removal will be easier, therefore, if we can ensure that the node we remove is a red node.

As we are searching for the item to be removed, let's use the following terminology:

- **Node X** - the current node we are visiting
- **Node P** - X's parent
- **Node S** - X's sibling
- **Node XL** - X's left child
- **Node XR** - X's right child
- **Node SL** - S's left child
- **Node SR** - S's right child

Our basic strategy will be to make sure that node X is always red, by doing a color swap, if necessary. The root node must be considered separately. The main case is subdivided into a number of subcases. The steps in the algorithm are outlined below:

1) X is the Root Node

If the root node has two black children, we change the color of the root to red, and branch left or right, as appropriate. Whichever child we land on becomes the new node X.

If the root node has at least one red child, the root node becomes node X and we proceed to the main case, Case 2.

2) Main Case

As we travel down the tree, our procedure for doing the color swap will ensure that X's parent, P, will always be red, because we would have made sure of that when we were visiting P. Both X and X's sibling, node S, will always be black. The first thing we will do is color X red. We may then need to perform rotations depending on the colors of the child nodes of X and S. There are two main subcases:

1. Both of X's children are black.

2. At least one of X's children is red.

Subcase 2.1 - Both of X's children are black

This subcase has five subcases of its own:

Subcase 2.1A - X's Sibling has two black children

Here we simply do a color swap. Node P becomes black, and nodes X and S both become red. We then continue moving down the tree.

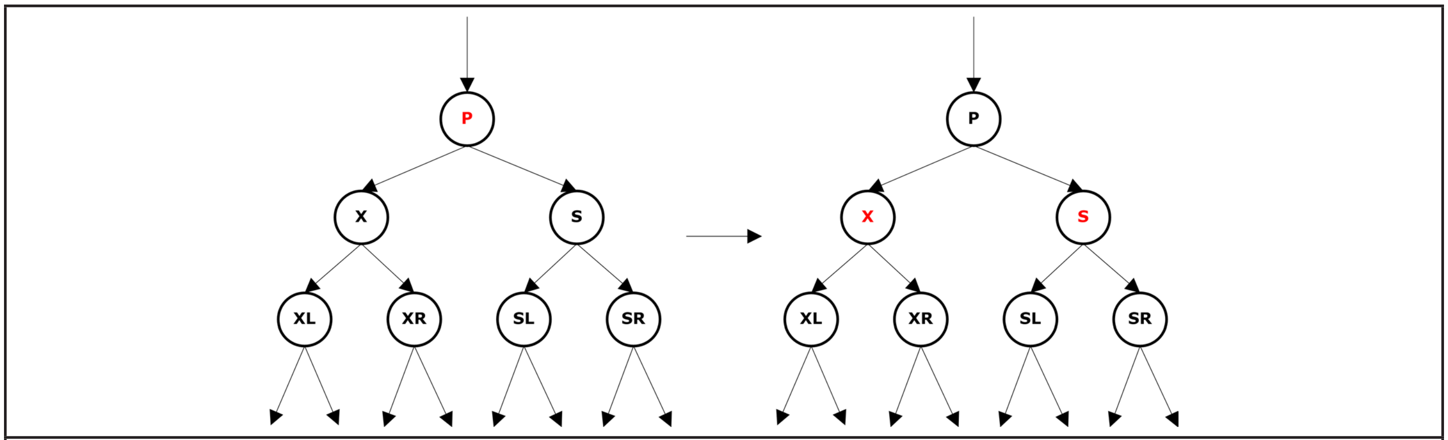


Figure 10.3. Top-down removal, subcase 2.1A, in a red-black tree.

Subcase 2.1B - X is in P's left subtree, and S's left child is red

In this case we perform a case 3 rotation, as shown below, change the color of X to red and the color of P to black, and continue.

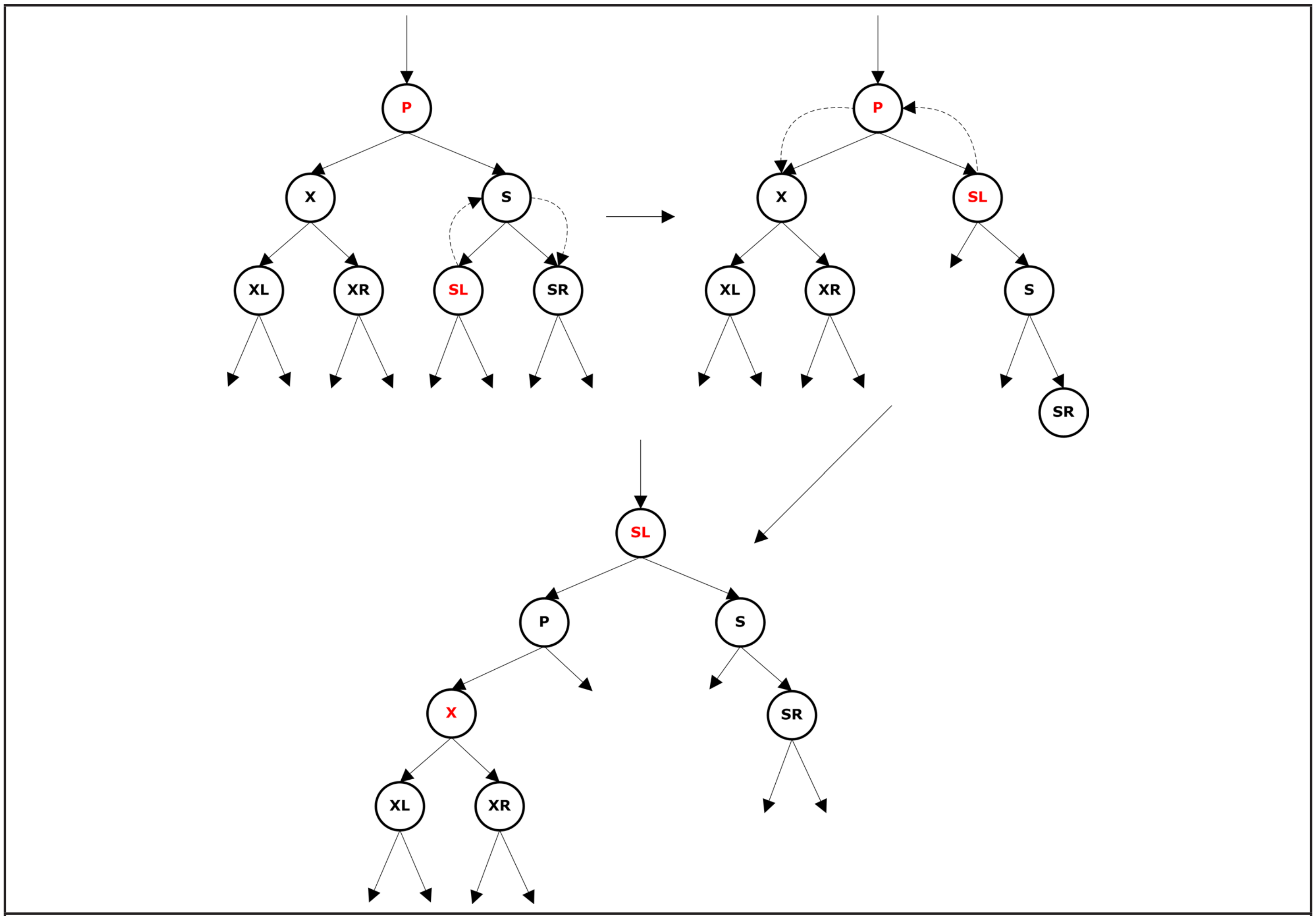


Figure 10.4. Top-down removal, subcase 2.1B, in a red-black tree.

Subcase 2.1B-M - X is in P’s right subtree and S’s right child is red

This is the mirror image of subcase 2.1B, so we perform a case 2 rotation, then change X to red and P to black, and continue.

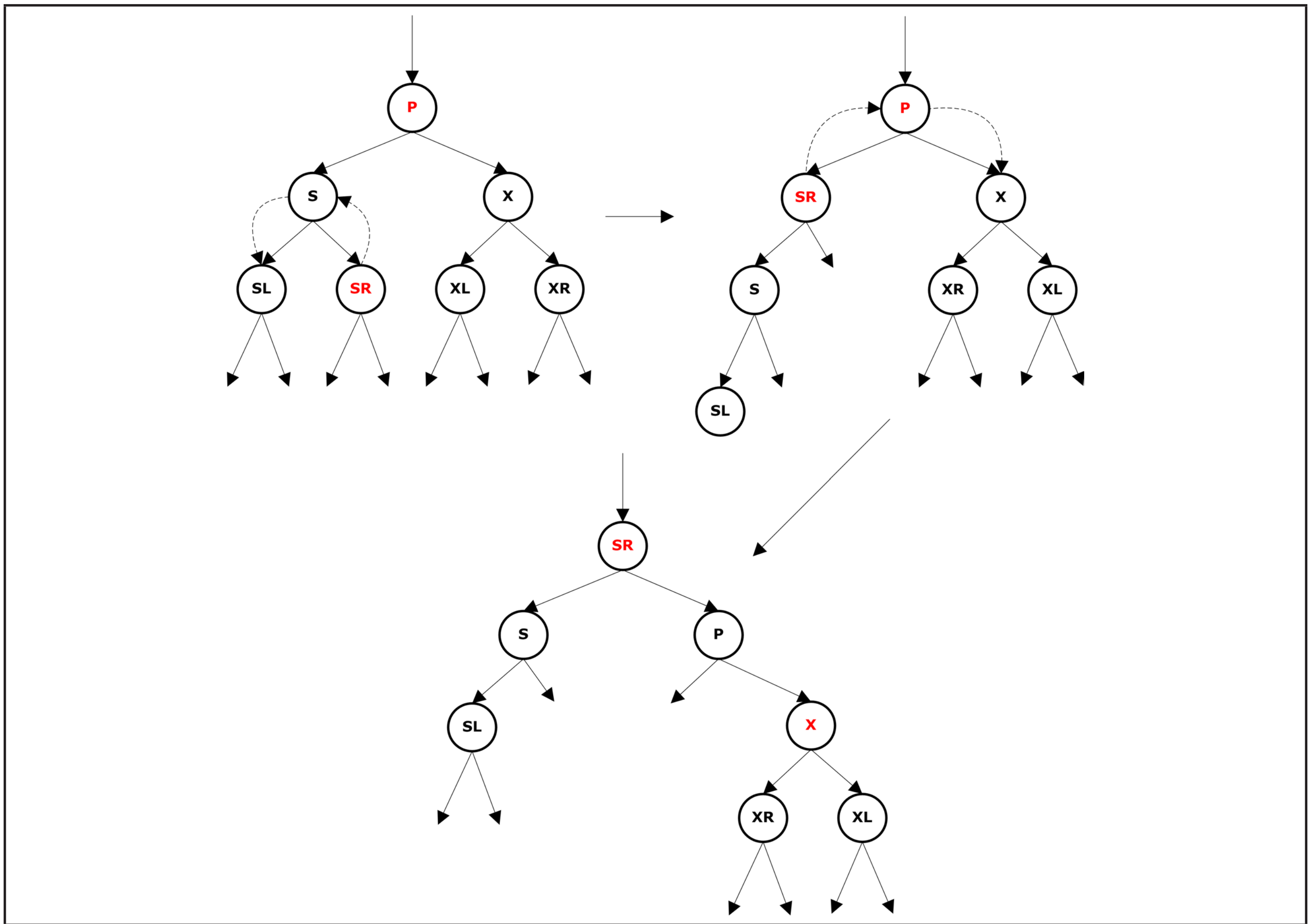


Figure 10.5. Top-down removal, subcase 2.1B-M, in a red-black tree.

Subcase 2.1C - X is in P’s left subtree and S’s right child is red

Here we do a color swap with nodes P, X, and S, and change S’s right child to black. We then pe form a case 4 rotation around P and continue. This is shown in Figure 10.6.

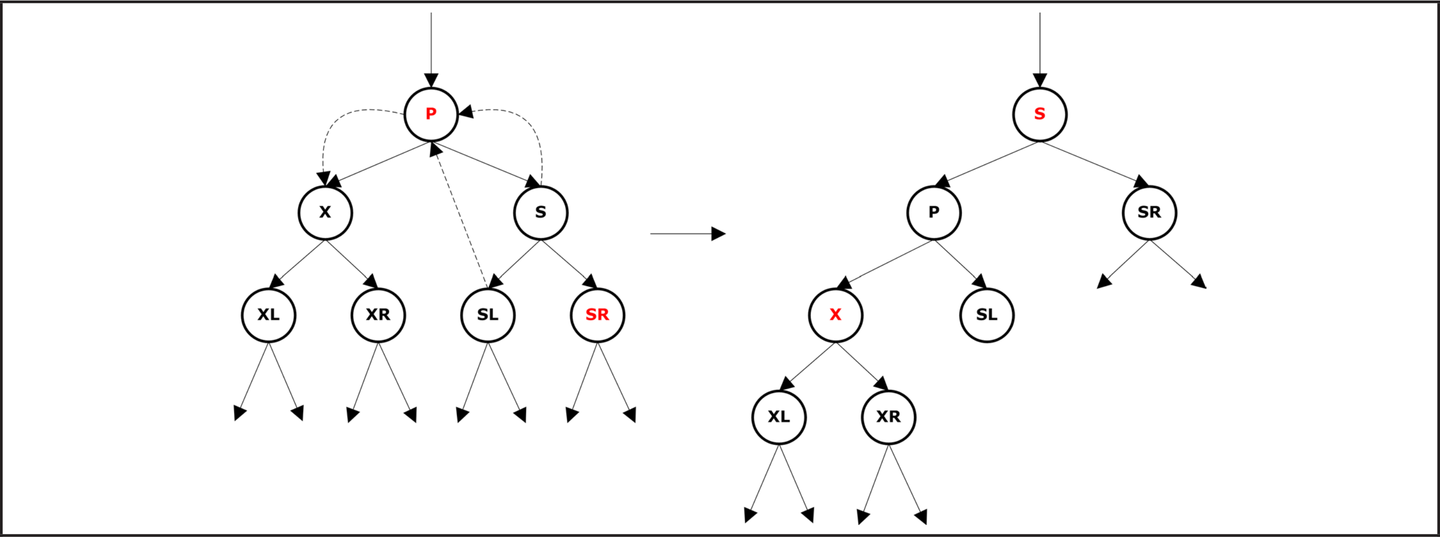


Figure 10.6. Top-down removal, subcase 2.1C, in a red-black tree.

Subcase 2.1C-M - X is in P's right subtree and S's left child is red

This is the mirror image of subcase 2.1C, so instead of doing a case 4 rotation around P we do a case 1 rotation, and continue.

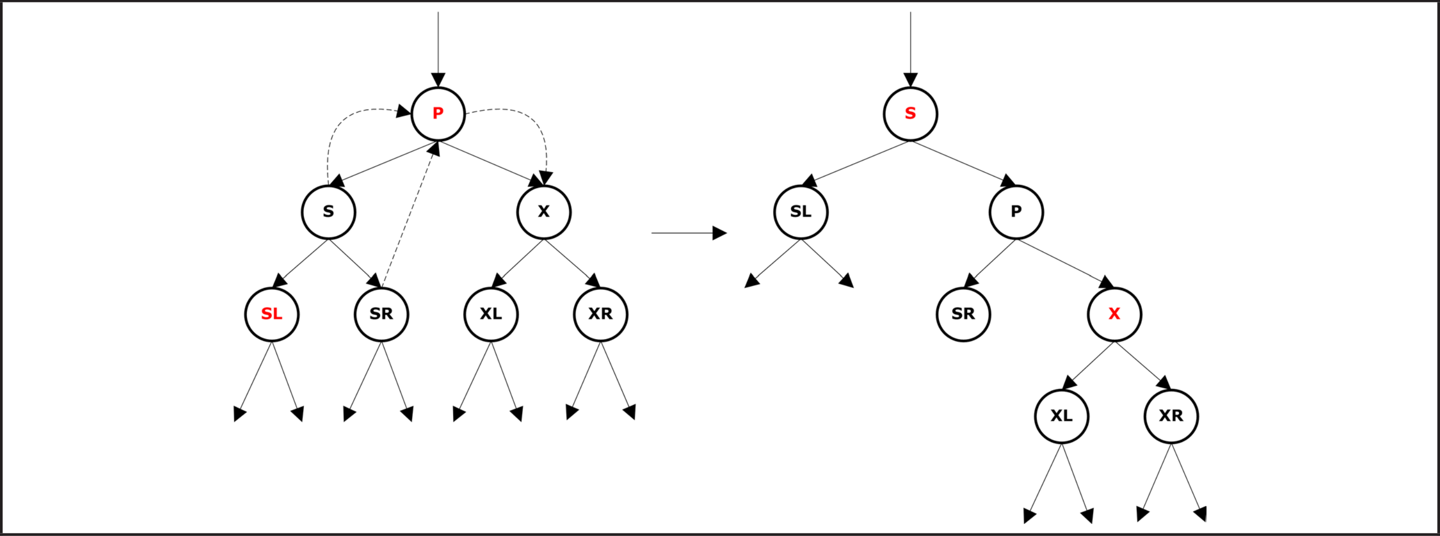


Figure 10.7. Top-down removal, subcase 2.1C-M, in a red-black tree.

Subcase 2.2 - X has at least one red child

In this case we just fall through to the next level. If we land on a red node, we don't need to do anything else, since the new X is red. If we land on a black node, we do a single rotation of S around P. If X is P's left child we do a case 4 rotation, otherwise we do a case 1 rotation. Following the rotation, we color P red and S black.

3) Remove the node

When we finally arrive at the item to be removed, there are three cases:

1. The node has 2 children.
2. The node has 1 child.
3. The node has 0 children.

If the node has two children, we follow the same procedure as for generic binary search trees. The other two cases have mirror images that must be considered separately. In any event, the removal is very simple — no rotations or color swaps are required.

4) Make sure the root node is black

The final step is explicitly color the root node black, in case it was colored red at the beginning of the algorithm.

Red-Black Tree Implementation

The `RedBlackTree` class implements the `BinarySearchTree` interface. The implementation is quite lengthy, but it does not use recursion.

The Nested Class **`RedBlackNode`**

The **`RedBlackNode`** class stores 5 attributes: 1) a data item, 2) reference to its left child, 3) reference to its right child, 4) height, and 5) color. While height will not be used to maintain balance, it is included since the `BinarySearchTree` interface specifies the ability to retrieve height information.

Attributes

Figure 10.8 shows the attributes of the `RedBlackTree` class.

Attribute	Purpose
`BLACK`	indicates a node is black
`RED`	indicates a node is red
`theSize`	keeps track of the number of items in the tree
`modCount`	keeps track of the number of structural modifications
`header`	stores a reference to the root node
`nullNode`	used to provide an equivalent to Java's `null` object, that is of type `RedBlackNode`
`current`	reference to the current node when searching for an insertion/removal point
`parent`	reference to the parent of `current`
`grand`	reference to the grandparent of `current`
`great`	reference to the great-grandparent of `current`

Figure 10.8. Attributes of the `RedBlackTree` class.

We use a header node to reference the root node, for the same reasons we used a header node for LinkedList. It simplifies the special case when the root node is involved in the addition or removal of items. By convention, the header node's right child is always the root of the tree; it's left child is always null.

The `nullNode` attribute is interesting. We use it as a substitute for Java's standard `null` reference to represent a null node. It is of type `RedBlackNode`, which means we can treat null nodes like regular `RedBlackNode` objects, instead of having to explicitly include an additional check to see if a node == `null`. The `nullNode` object itself is statically initialized using a code block that has class scope within `RedBlackTree`, and is treated as a separate object. Because it is static, all instances of `RedBlackTree` will reference the same instance of `nullNode`.

The attributes, `current`, `parent`, `grand`, and `great` are typically referred to as **sentinels**. In order to accomplish additions and removals in a single pass it is necessary to maintain references to certain up-stream nodes, since traversals are one-way and we aren't using recursion to travel back up the tree. Since they are so intimately connected with the red-black tree operations, they have been made class attributes rather than local variables within methods.

Fixing Imbalances

As the class is currently implemented, the `add(Comparable obj)` method uses two helper methods, `balance(Comparable obj)` and `rotate(Comparable item, RedBlackNode parent)`, to restore balance when necessary. The `balance` method first does a color swap, followed by either a single or double rotation. The actual rotations are carried out by the `rotate` method.

For the `remove` method, the color swaps are integrated into the method's code — the `remove` method does not use the `balance` method. Rotations are performed using the same rotation methods used for the AVL tree.

Adding an Item Using `add(Comparable obj)`

The `add` method is not all that complicated. The most important thing to note that the method is iterative, not recursive. In order to implement the method iteratively, imbalances caused by having two consecutive red nodes must be anticipated and corrected before they can occur. Sentinels must also be used to maintain references to nodes higher in the tree than the current node, since we have no other way to move back to those nodes.

Removing an Item Using `remove(Comparable obj)`

The first thing you will probably notice about this method is its length. It can be optimized and broken down somewhat, but even then the remove method will still be extremely complex. Again, potential imbalances are anticipated on the way down the tree, but the process is more difficult. When adding an item we know we will always be adding a red node to the tree, so we will not be in danger of violating rule #4. When removing an item, however, there is no way to know whether the node containing that item will be red or black, or how many children it will have. As a result, many cases must be considered. As with adding items, the sentinels are used to maintain references to relevant upstream nodes.

Running Times for the Major Operations (add, retrieve, and remove)

All three major operations — adding items, retrieving items, and removing items, have O(log N) running times, like the AVL tree. The iterative nature of the red-black tree operations, however, make them slightly faster than their recursive counterparts in the AVL tree.

AA Trees

The AA tree is an alternative to the red-black tree. The performance of an AA tree is comparable to that of the red-black tree (i.e., add, retrieve and remove operations are logarithmic), but recursion is used to simplify the code. Since recursion is used, the add and remove operations will be two-pass operations. The first pass proceeds down the tree locates the insertion/removal point, and the second pass returns upward to the root, correcting imbalances along the way. I'm only going to present the basics of the AA tree, without implementation details.

Balance in an AA Tree

In an AA tree the balance information is based on a node's level in the tree. A node's level is the number of left links from that node to a null link. Leaf nodes are considered to be level 1. The AA tree introduces a **horizontal link**, which is a link between a parent node and a child node of the same level. The rules for an AA tree are as follows:

1. All horizontal links must be right links (no left horizontal links are allowed).
2. There may never be 2 consecutive horizontal links.
3. Nodes at level 2 or higher must have 2 children.
4. If a node does not have a horizontal link, and has 2 children, both child nodes are at the same level.

Correcting Imbalances in an AA Tree

There are two rebalancing operations used with AA trees: the **skew** operation and the **split** operation. Either operation can introduce a new imbalance, so a combination of skew/split operations often need to be performed from the insertion or deletion point back up to the root.

The Skew Operation

The skew operation fixes the problem of having an illegal left horizontal link, as shown in Figure 10.9, by flipping the left horizontal link to make it a right horizontal link.

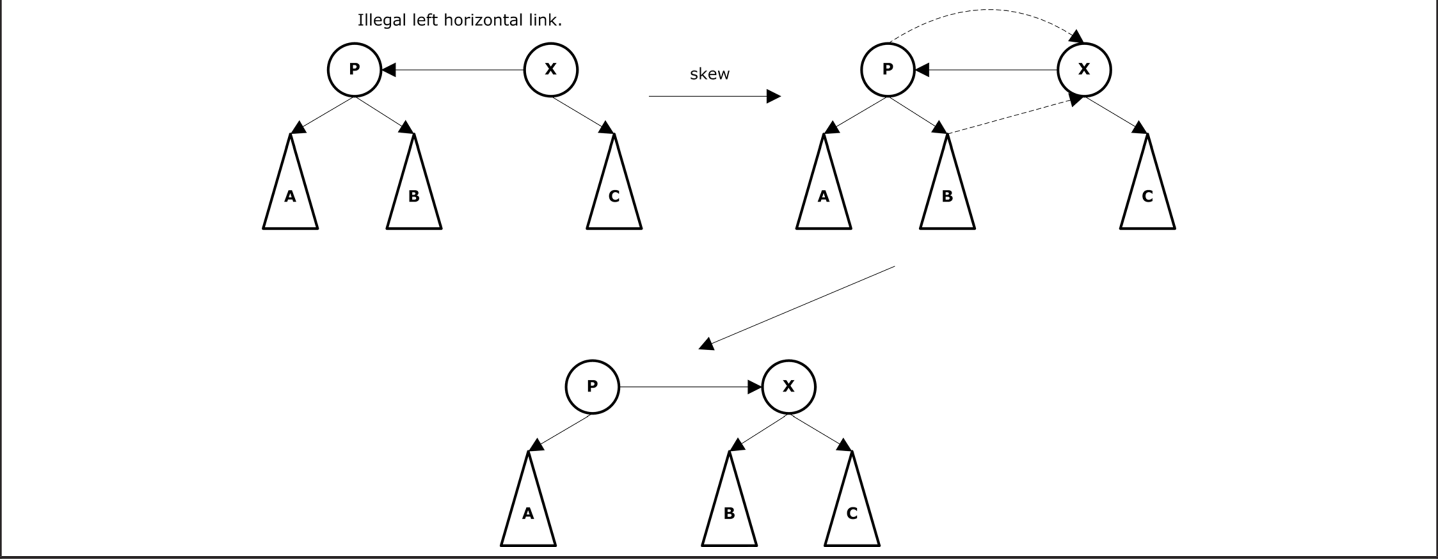


Figure 10.9. Illustration of the skew operation in an AA tree.

The Split Operation

The split operation fixes the problem of having 2 consecutive horizontal links, as shown in Figure 10.10.

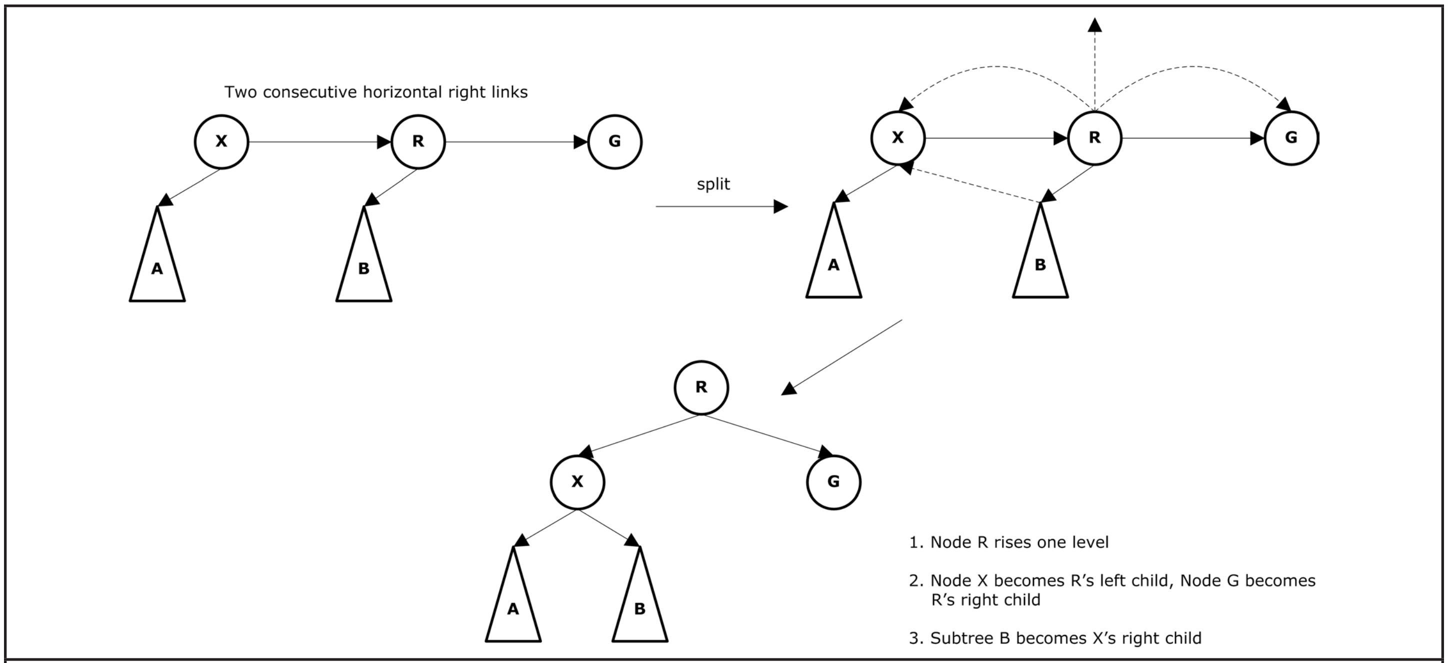

Two consecutive horizontal right links
X
R
G
A
B
split
R
X
G
A
B
1. Node R rises one level
2. Node X becomes R's left child, Node G becomes R's right child
3. Subtree B becomes X's right child

Figure 10.10. Illustration of the split operation in an AA tree.

Example of Adding Items to an AA Tree

To better show how the skew and split operations work, we'll build an AA tree from scratch. The steps are shown in Figure 10.11 (parts 1 & 2).

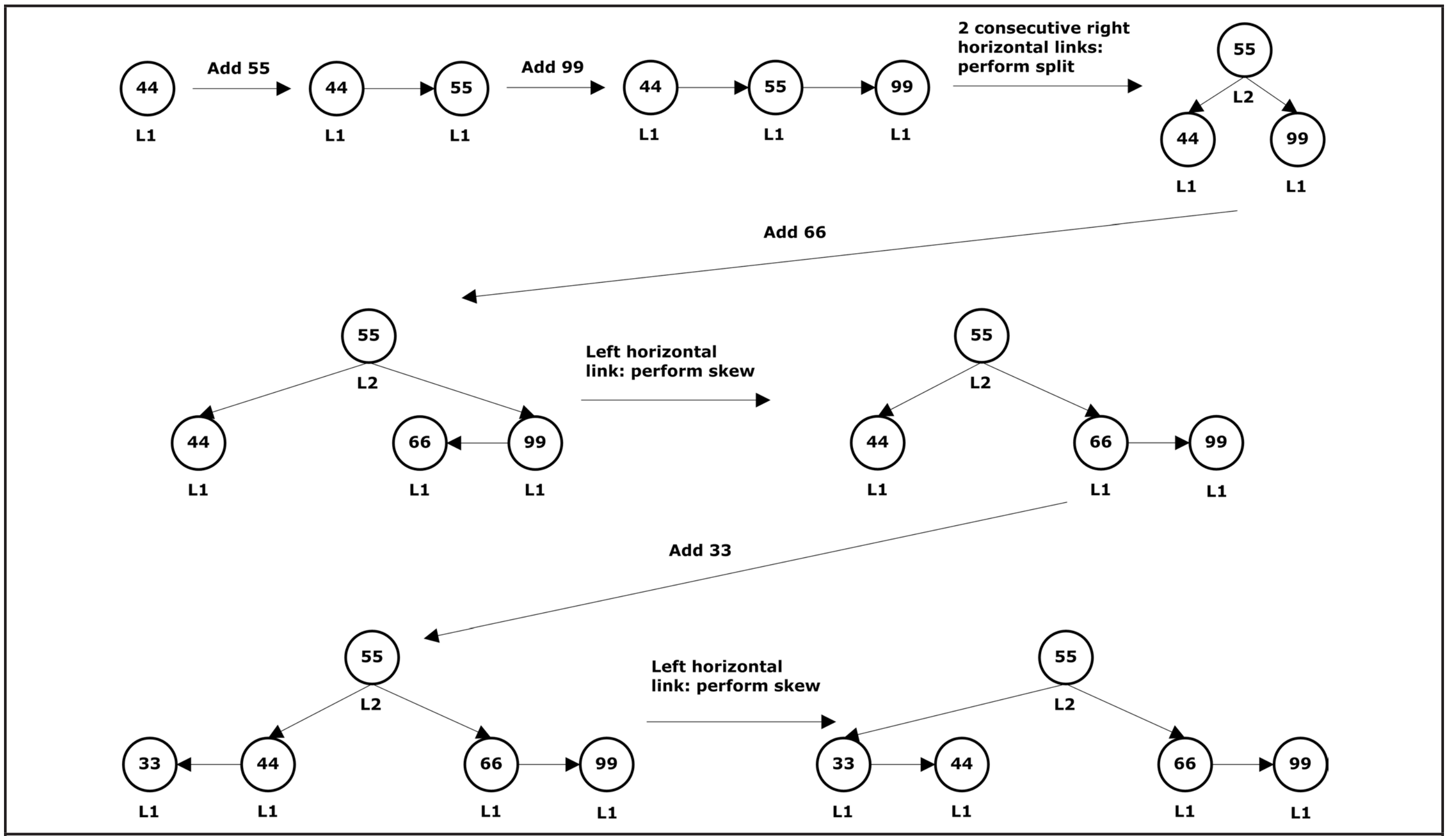
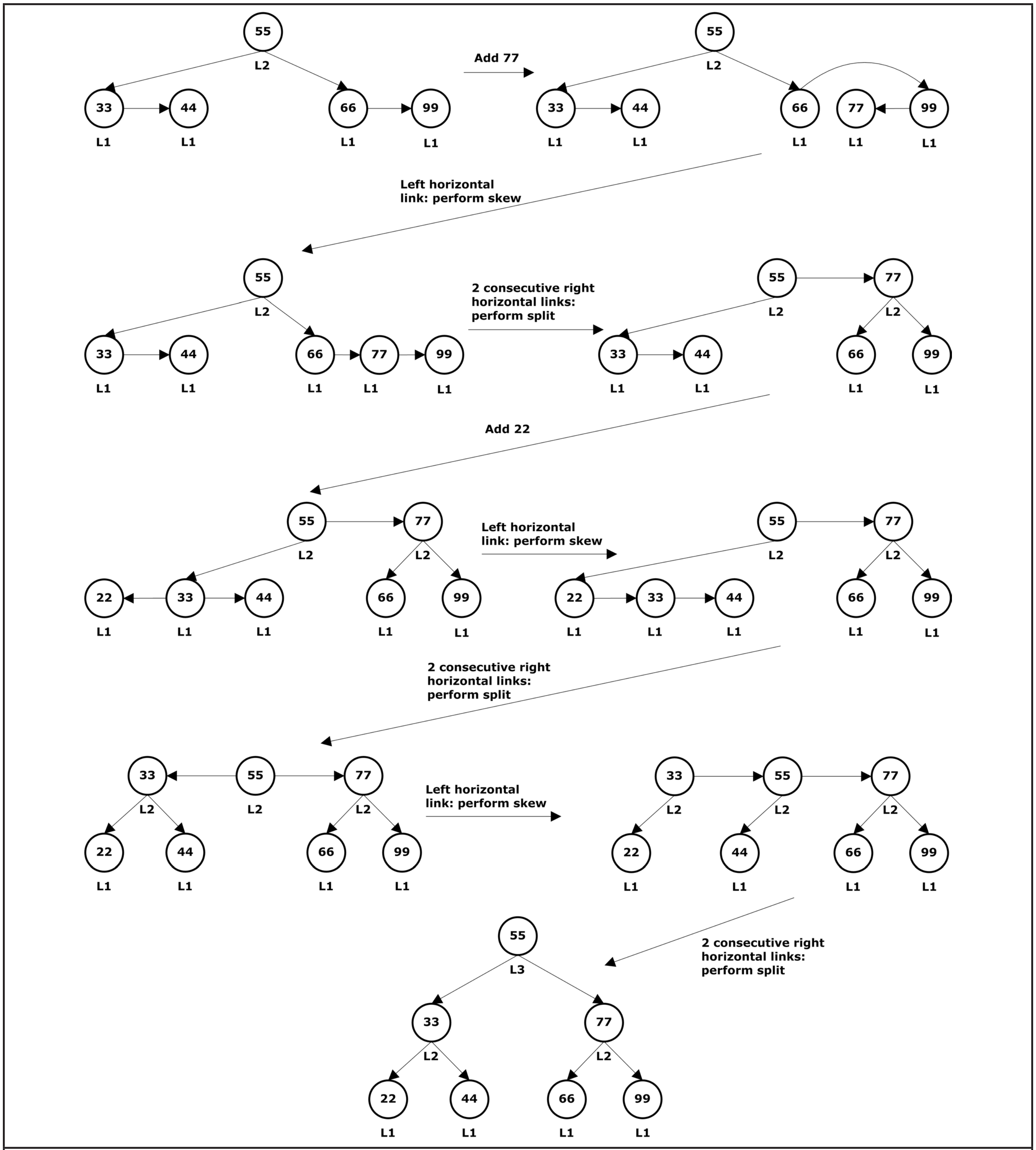


Figure 10.11 (Part 1). Building an AA tree beginning with an empty tree, to illustrate how the skew and split operations work.



55
L2
33
44
66
99
L1
Add 77
77
Left horizontal link: perform skew
2 consecutive right horizontal links: perform split
Add 22
22
L3

Figure 10.11 (Part 2). Building an AA tree beginning with an empty tree, to illustrate how the skew and split operations work.

The Need for Higher Order Trees: The Effect of Disk Accesses on Running Time

For all the data structures we have talked about so far, we have assumed all the data will fit into the main memory of the computer. However, there are times when this is not possible, and it becomes necessary to store part of the data externally, such as on a disk.

Storing part of the data on disk raises a new problem. When all the data are in main memory, all operations are, for the most part, equal. When part of the data are located on a disk, operations that require disk accesses become very expensive. When disk accesses are involved they tend to dominate the running time. Disk accesses are manifested as enormous constants which cannot be ignored when doing Big-Oh analysis.

Some considerations (this info may be a little outdated, but it will suffice to illustrate the point):

1. Suppose we have a 500 MIPS machine, meaning you can execute 500 million machine language instructions per second.

2. A typical hard drive spins at around 7200 rpm.

3. A single disk access takes approximately (60 seconds/7200 revolutions * 1000 ms/second) = 8.3 milliseconds. Although we might expect to find what we are looking for in half a revolution, on average, there is overhead involved in precisely positioning the read/write head. Usually, most hard drives are rated at about 9-11 ms for a single read/write access. Thus, it is possible to get about 90-110 disk accesses per second.

4. To put it in proportion, in one second we can either perform 90-110 disk accesses or execute about 500 million instructions in memory. This means one disk access is worth about 4 million instructions. The problem gets worse if our program must share resources, access other relatively slow devices, or use a network connection.

The point is that we can possibly avoid a very expensive disk access by writing a very complicated piece of code that can execute in memory. Since we have about 4 million instructions to work with that amounts to a lot of code. Furthermore, if we can reduce the number of disk accesses we need by a fourth, or by half, we can decrease the running time by a fourth or by half.

An Example

Assume we have a data structure with 10,000,000 items, and that all the data will not fit in main memory, so we must access it from a hard disk. The items are 256 bytes in size, and each item is associated with a 32-byte search key.

If we used a standard binary search tree, the worst case would be a tree that has become unbalanced and degenerated into a linked list, and the item we need is the last item in the list. The search operation would require 10,000,000 disk accesses, which translates to 10,000,000 accesses/100 accesses/second = 100,000 seconds (almost 28 hours!), assuming we have 100% of the machine’s resources at our disposal. If we only have 1/20 of the machine’s resources (which is not uncommon), the same operation would take almost 560 hours, or about 23 days! Obviously, this is unacceptable.

On average, though, a successful search will require approximately 1.38 log N (remember when we say log N we really mean log2 N), or 1.38 log(10,000,000) = 1.38 * 24 = 33, disk accesses. If we have full use of the machine’s resources, it will take about 1/3 second to find the item. However, if we only have use of 1/20 of the machine’s resources, the operation will take about (1/3 second) * 20 = 6.67 seconds.

If we use a red-black tree, we will rarely encounter the worst case (1.44 log N); typical searches will be very close to log N. A typical search would therefore require log(10,000,000) = 24 disk accesses, which will require about 24 accesses/100 accesses/sec = ~1/4 second with 100% of the resources available, or ~5 seconds. This is much better than a standard binary search tree.

Although 5 seconds for a search is much better than 28 hours, it is still too long. If our program needs to handle even as little as 1000 searches per day, it will still spend a whopping 5000 seconds, or ~83 minutes to perform those searches. We need to reduce the running time for a typical search to a much lower value. In a search tree the time required for a search is proportional to the height of the tree. The solution is to use a search tree that has more branching. More branching results in shorter height, which should result in shorter search times. If we have a quinary (or 5-ary) tree, we will have an average height of log5N; if we use a 10-ary (or denary) tree, we will have an average height of log10N. To generalize, if we use an M-ary search tree we will be able to get an average height of logMN. As with the binary search trees, we must ensure that the tree is balanced in order to get optimal performance.

B Trees

A B-tree of order M is an M -ary tree such that:

1. The data items are stored in the leaves.

2. Non-leaf nodes store up to M - 1 keys to direct searching.

3. The root is either a leaf or has between 2 and M child nodes (it cannot have just 1 child node).*

4. All non-leaf nodes except the root have between M/2 (round up) and M child nodes.

5. All leaves are at the same depth and contain between L/2 (round up) and L data items for some L (L does not have to equal M).*

   *for the first L insertions, these properties cannot be followed strictly, since there are not enough data records to do so

Each node in the tree represents a disk block. The values of M and L are chosen based on the sizes of the items being stored. Using the example above we would calculate the values for M and L as follows:

1. Assume that one disk block can hold 8192 bytes.

2. Since each key is 32 bytes long, and we need M - 1 keys, the keys will consume 32M - 32 bytes.

3. Each node will have M branches. A branch is simply a reference to another disk block, which we will assume takes up 4 bytes. Thus, we need 4M bytes to store the branches.

4. So, each non-leaf node will need 32M - 32 + 4M bytes, or 36M - 32 bytes.

5. To find the value of M, we choose the largest value of M such that 36M - 32 <= 8192. This will allow us to store in a single node the keys for the maximum number of items that will fit in one disk block. In this example, M would be 228: (36*228 - 32 = 8176).

6. Since each data record is 256 bytes long, we can fit exactly 32 records into 8192 bytes, or 1 disk block. Thus, we would choose a value of 32 for L.

7. Each leaf will therefore contain between 16 and 32 records. Each non-leaf node will have between 114 and 228 child nodes.

8. For 10,000,000 items there will be between 312,500 and 625,000 leaves, assuming each leaf contains 32 or 16 items, respectively.

The resulting tree would look something like this:

root	
|	
228	At this level, the tree could hold a maximum of: 228 leaves * 32 items/leaf = **7296** items
|	
51,984	At this level, the tree could hold a maximum of: 51,984 leaves * 32 items/leaf = **1,663,488** items
|	
11,852,352	At this level, the tree could hold a maximum of: 11,852,352 leaves * 32 items/leaf = **379,275,264** items

So, what we end up with is a 228-ary tree; that is, each node can have a maximum of 228 child nodes. In this example the tree is only 3 levels deep. It's an extremely short, but extremely wide tree. Searches will be very quick since at most any search can only descend 3 levels.

The book does not provide any source code for a B-tree, which is fine since we don't really have time for a detailed examination of a B-tree's code. But it is important that you know some general information about how B-trees work, though, so we will go through a brief discussion of the mechanisms used for adding and removing items in a B-tree.

Adding Items to a B-Tree

Adding items is easy as long as the items are inserted into leaves that do not already contain the maximum number of items. When a leaf is full, rearrangement is necessary to obey the B-tree rules. One solution is to split the leaf into 2 leaves, dividing the number of items evenly between the leaves. In doing so each leaf will be assured to contain the minimum number of items it must have according to the B tree definition.

Obviously this requires changing the values in each of those leaves, but it also requires that the keys in the parent node be changed. Although this is complicated, remember we are willing to write complicated code if it will save us disk accesses. Note that we do incur at least 2 additional disk writes, one for each of the nodes resulting from the split. However, if we have chosen values for M and L properly, this should be a rare occurrence.

What happens if we need to split a node, but the node's parent already has the maximum number of child nodes it is allowed to have? In that case it is necessary to also split the parent node. The splitting process continues up the tree, as needed. If the root node must be split, a new root node is created that has as its children the 2 nodes that result from splitting the root node. Splitting the root node adds another level of depth to the tree.

Alternatively, instead of splitting a node, you could use any available space in any of the sibling nodes, rearranging the items and the keys in the parent node as necessary. Over time this could prove to be a less efficient method, since splitting nodes creates space for future additions. The cost is analogous to doubling the size of the internal array for the ArrayList class, as opposed to simply creating a new array that is one cell larger than it was before.

An example is shown in figures 10.12 through 10.15. It assumes a 5-ary tree, in which each leaf node is capable of storing 5 items, and each non-leaf node stores M - 1, or 4, keys. Note how each key maps to the smallest value in the leaf node it references.

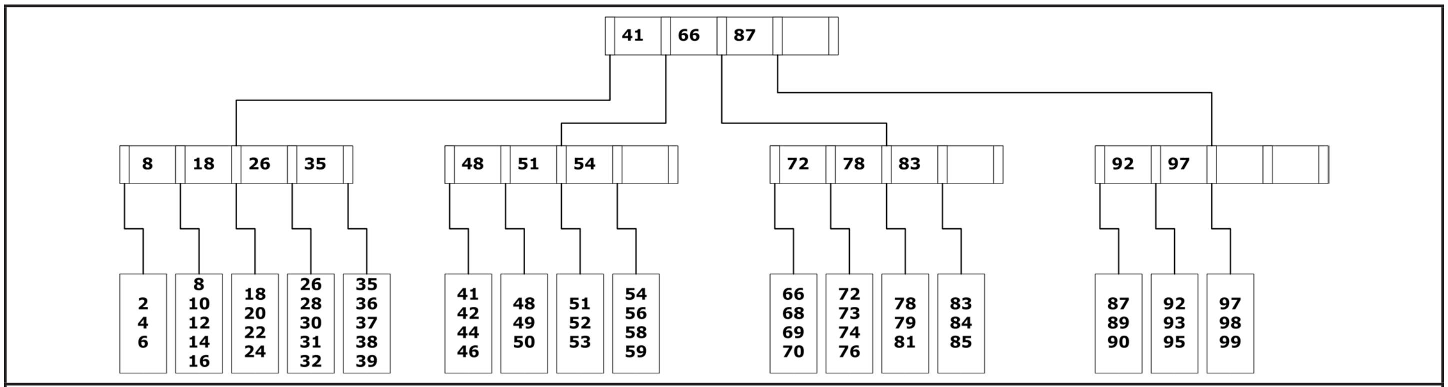


Figure 10.12. An example B-tree of order 5.

If we want to insert the value '57' into this B-tree, we can simply add it to the appropriate node since the node has room for one more item. Figure 10.13 shows what the tree would look like after insertion.

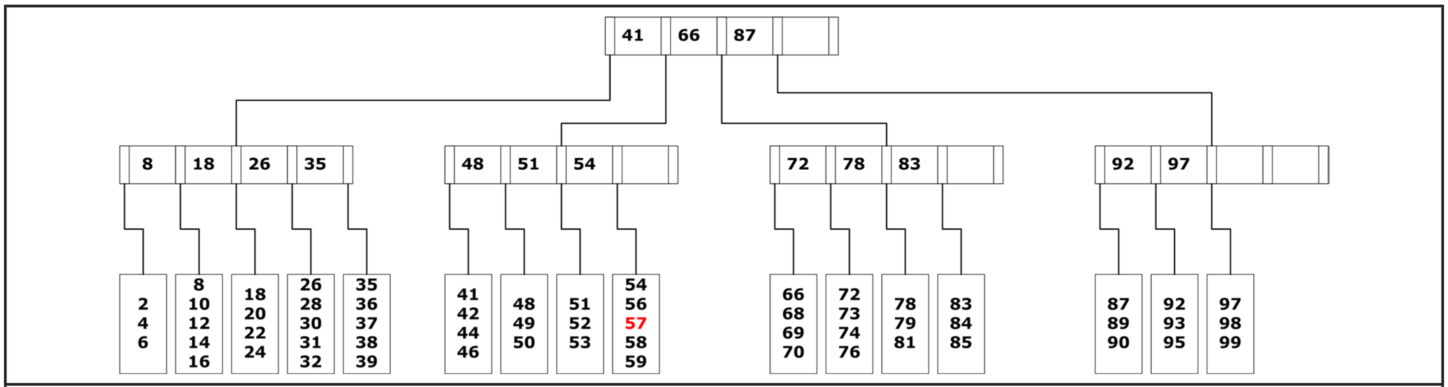


Figure 10.13. The B-tree from Figure 10.12, after adding the value '57'.

Now suppose we want to insert the value '55'. It should go into the same node in which we placed the '57', but that node is full, so we must split the leaf node into 2 leaf nodes. This can be done fairly easily since the parent node currently only contains references to 4 leaf nodes. A new key must be added to the parent node to map to the smallest item in the new leaf. Figure 10.14 shows the result.

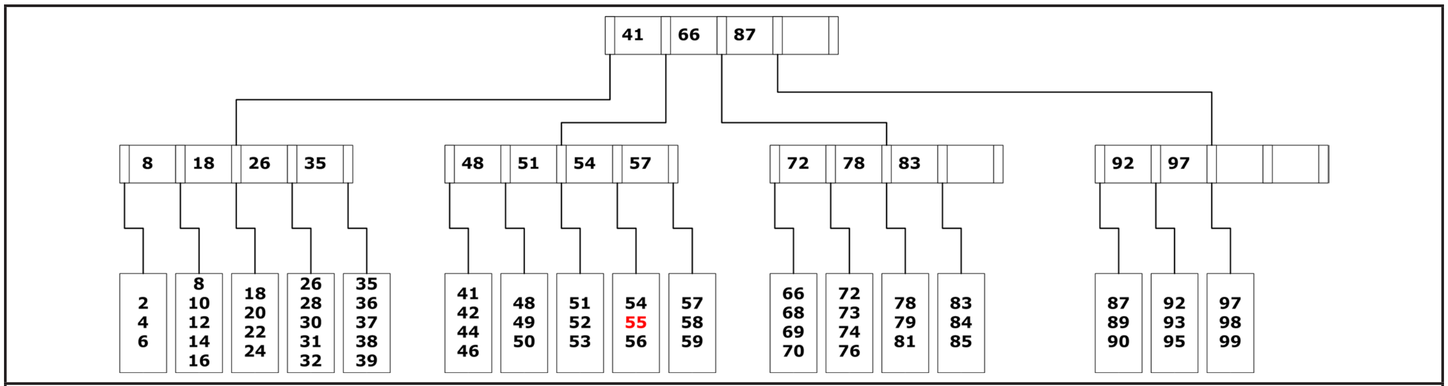


Figure 10.14. The B-tree from Figure 10.13, after adding the value '55'.

Now suppose we want to add the item '40'. Tracing down the tree we see the item should be placed into the fifth leaf node from the left. However, that leaf is full so we need to split it. Splitting the leaf node is more complicated this time because the parent node already contains the maximum number of leaves it may have. Therefore, we must also split the parent node. The result is shown in Figure 10.15.

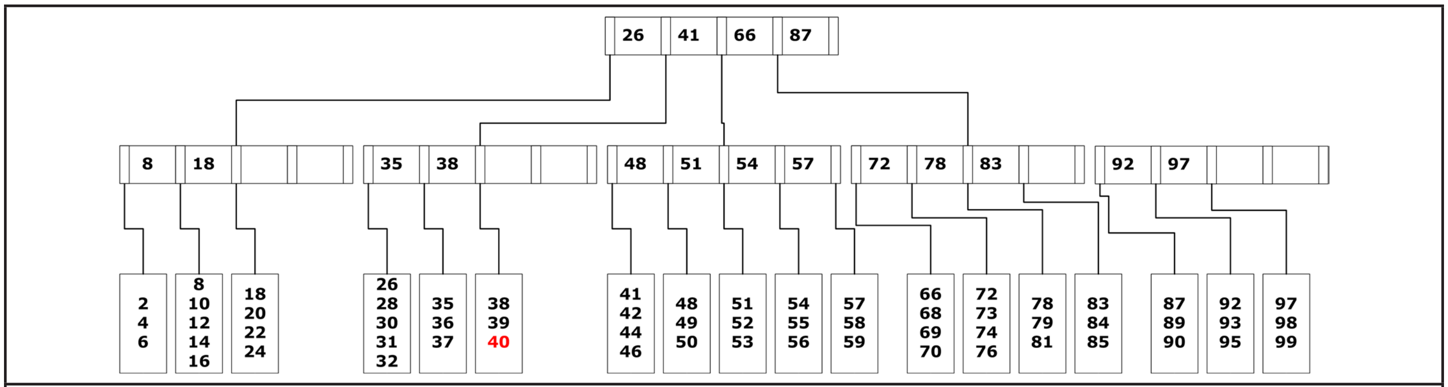


Figure 10.15. The B-tree from Figure 10.14, after adding the value '40'.

Removing Items from a B Tree

To remove an item, we simply need to find the item and remove it. If this causes a leaf to have fewer than the minimum number of items it must have, an item from a sibling node must be moved over to compensate. If there are not enough items for each leaf node to have its minimum number of items, two leaf nodes can be combined. If this combination results in the parent having fewer than its minimum number of child nodes, the process must be continued up the tree until the B-tree rules are satisfied for all nodes. If the root is left with only one child as the result of a deletion, the root node is removed and its sole child node becomes the new root of the tree, thus shortening the height of the tree by one level.

An example is shown in Figure 10.16. It takes the B-tree from Figure 15 and shows what it would look like after deletion of the value '99'. Deleting the '99' results in its leaf node having only 2 items, so the node is combined with its sibling leaf node.

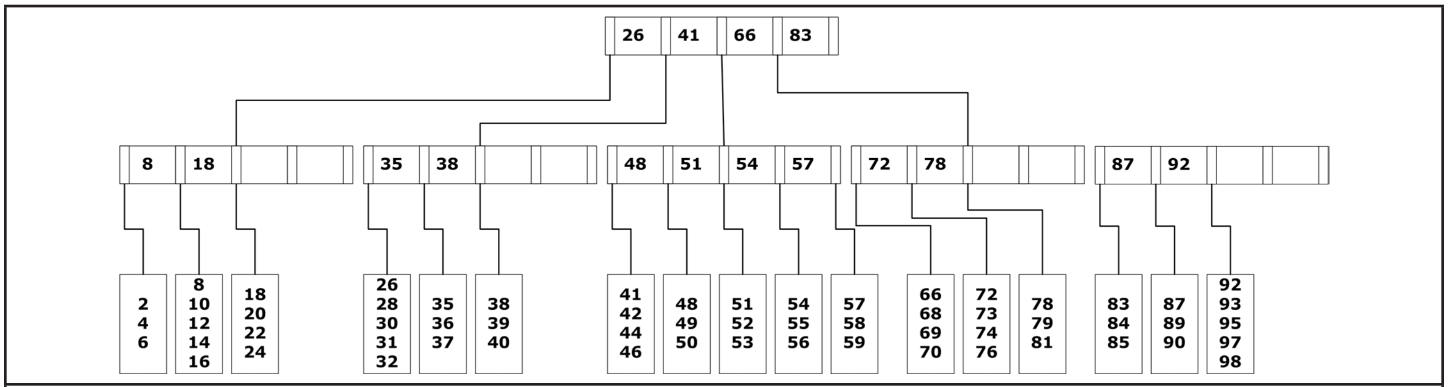


Figure 10.16. The B-tree from Figure 10.15, after removing the value '99'.