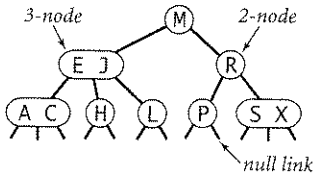
3.3 BALANCED SEARCH TREES

The algorithms in the previous section work well for a wide variety of applications, but they have poor worst-case performance. We introduce in this section a type of binary search tree where costs are *guaranteed* to be logarithmic, no matter what sequence of keys is used to construct them. Ideally, we would like to keep our binary search trees perfectly balanced. In an N-node tree, we would like the height to be $\sim \lg N$ so that we can guarantee that all searches can be completed in $\sim \lg N$ compares, just as for binary search (see PROPOSITION B). Unfortunately, maintaining perfect balance for dynamic insertions is too expensive. In this section, we consider a data structure that slightly relaxes the perfect balance requirement to provide guaranteed logarithmic performance not just for the *insert* and *search* operations in our symbol-table API but also for all of the ordered operations (except range search).

2-3 search trees The primary step to get the flexibility that we need to guarantee balance in search trees is to allow the nodes in our trees to hold more than one key. Specifically, referring to the nodes in a standard BST as *2-nodes* (they hold two links and one key), we now also allow *3-nodes*, which hold three links and two keys. Both 2-nodes and 3-nodes have one link for each of the intervals subtended by its keys.



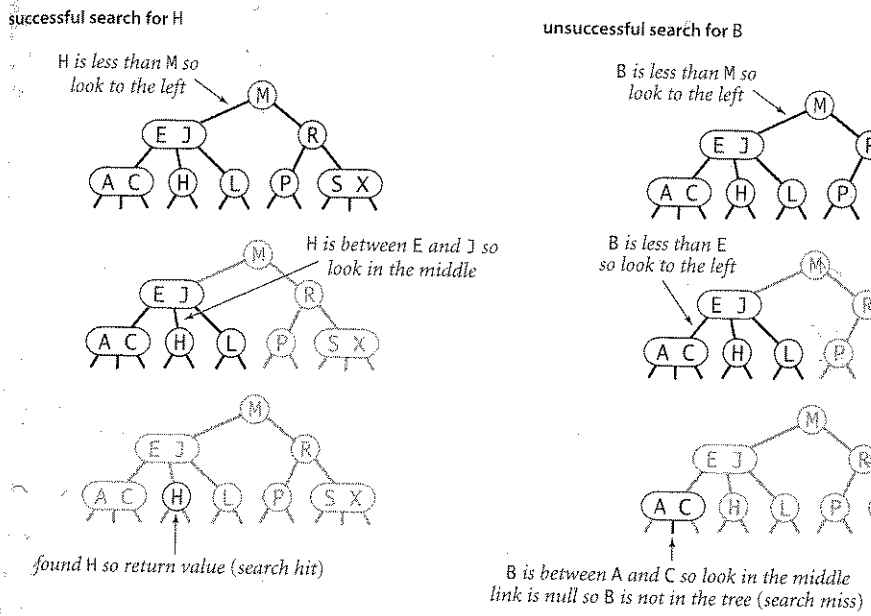
Anatomy of a 2-3 search tree

Definition. A *2-3 search tree* is a tree that is either empty or

- A *2-node*, with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A *3-node*, with two keys (and associated values) and *three* links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys

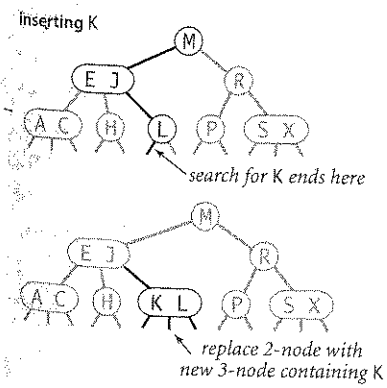
As usual, we refer to a link to an empty tree as a *null link*.

A *perfectly balanced* 2-3 search tree is one whose null links are all the same distance from the root. To be concise, we use the term *2-3 tree* to refer to a perfectly balanced 2-3 search tree (the term denotes a more general structure in other contexts). Later, we shall see efficient ways to define and implement the basic operations on 2-nodes, 3-nodes, and 2-3 trees; for now, let us assume that we can manipulate them conveniently and see how we can use them as search trees.



Search hit (left) and search miss (right) in a 2-3 tree

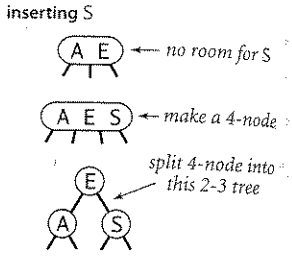
Search. The search algorithm for keys in a 2-3 tree directly generalizes the search algorithm for BSTs. To determine whether a key is in the tree, we compare it against the keys at the root. If it is equal to any of them, we have a search hit; otherwise, we follow the link from the root to the subtree corresponding to the interval of key values that could contain the search key. If that link is null, we have a search miss; otherwise we recursively search in that subtree.



Insert into a 2-node

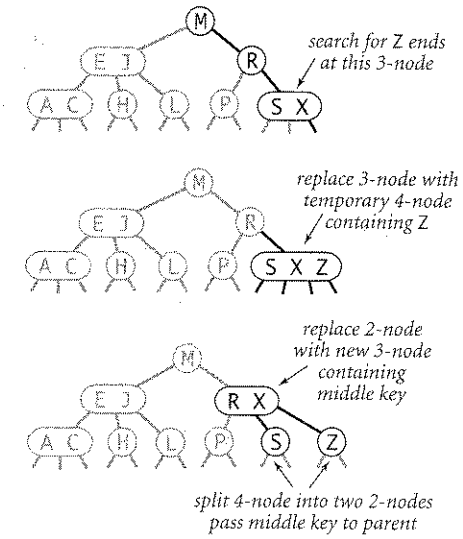
Insert into a 2-node. To insert a new key in a 2-3 tree, we might do an unsuccessful search and then hook on a new node with the key at the bottom, as we did with BSTs, but the new tree would not remain perfectly balanced. The primary reason that 2-3 trees are useful is that we can do insertions and still maintain perfect balance. It is easy to accomplish this task if the node at which the search terminates is a 2-node: we just replace the node with a 3-node containing its key and the new key to be inserted. If the node where the search terminates is a 3-node, we have more work to do.

Insert into a tree consisting of a single 3-node. As a first warmup before considering the general case, suppose that we want to insert into a tiny 2-3 tree consisting of just a single 3-node. Such a tree has two keys, but no room for a new key in its one node. To be able to perform the insertion, we temporarily put the new key into a *4-node*, a natural extension of our node type that has three keys and four links. Creating the 4-node is convenient because it is easy to convert it into a 2-3 tree made up of three 2-nodes, one with the middle key (at the root), one with the smallest of the three keys (pointed to by the left link of the root), and one with the largest of the three keys (pointed to by the right link of the root). Such a tree is a 3-node BST and also a perfectly balanced 2-3 search tree, with all the null links at the same distance from the root. Before the insertion, the height of the tree is 0; after the insertion, the height of the tree is 1. This case is simple, but it is worth considering because it illustrates height growth in 2-3 trees.



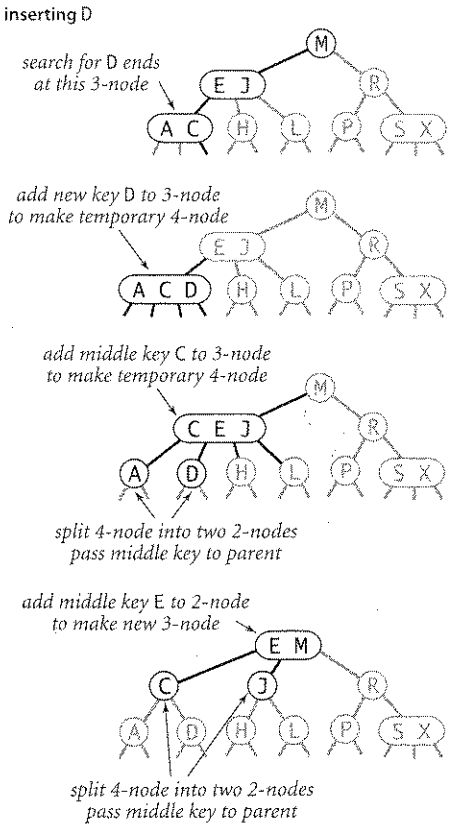
Insert into a single 3-node

Insert into a 3-node whose parent is a 2-node. As a second warmup, suppose that the search ends at a 3-node at the bottom whose parent is a 2-node. In this case, we can still make room for the new key *while maintaining perfect balance in the tree*, by making a temporary 4-node as just described, then splitting the 4-node as just described, but then, instead of creating a new node to hold the middle key, moving the middle key to the node's parent. You can think of the transformation as replacing the link to the old 3-node in the parent by the middle key with links on either side to the new 2-nodes. By our assumption, there is room for doing so in the parent: the parent was a 2-node (with one key and two links) and becomes a 3-node (with two keys and three links). Also, this transformation does not affect the defining properties of (perfectly balanced) 2-3 trees. The tree remains ordered because the middle key is moved to the parent, and it remains perfectly balanced: if all null links are the same distance from the root before the insertion, they are all the same distance from the root after the insertion. Be certain that you understand this transformation—it is the crux of 2-3 tree dynamics.



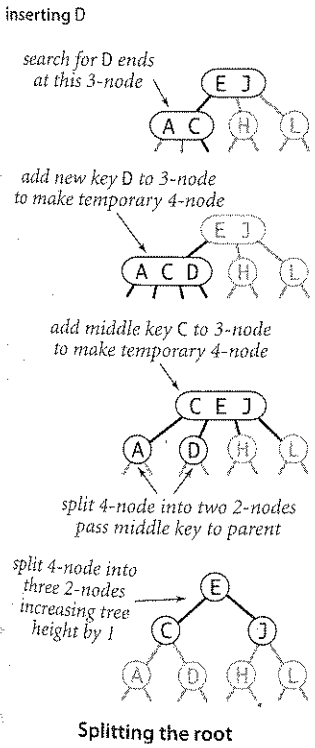
Insert into a 3-node whose parent is a 2-node

Insert into a 3-node whose parent is a 3-node. Now suppose that the search ends at a node whose parent is a 3-node. Again, we make a temporary 4-node as just described, then split it and insert its middle key into the parent. The parent was a 3-node, so we replace it with a temporary new 4-node containing the middle key from the 4-node split. Then, we perform *precisely the same transformation on that node.* That is, we split the new 4-node and insert its middle key into *its* parent. Extending to the general case is clear: we continue up the tree, splitting 4-nodes and inserting their middle keys in their parents until reaching a 2-node, which we replace with a 3-node that does not need to be further split, or until reaching a 3-node at the root.



Insert into a 3-node whose parent is a 3-node

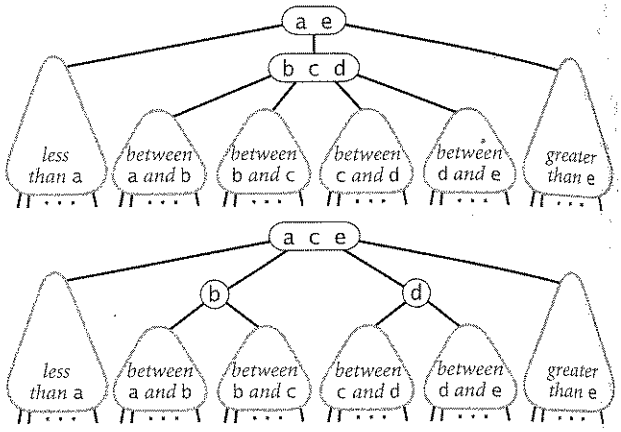
Splitting the root. If we have 3-nodes along the whole path from the insertion point to the root, we end up with a temporary 4-node at the root. In this case we can proceed in precisely the same way as for insertion into a tree consisting of a single 3-node. We split the temporary 4-node into three 2-nodes, increasing the height of the tree by 1. Note that this last transformation preserves perfect balance because it is performed at the root.



Splitting the root

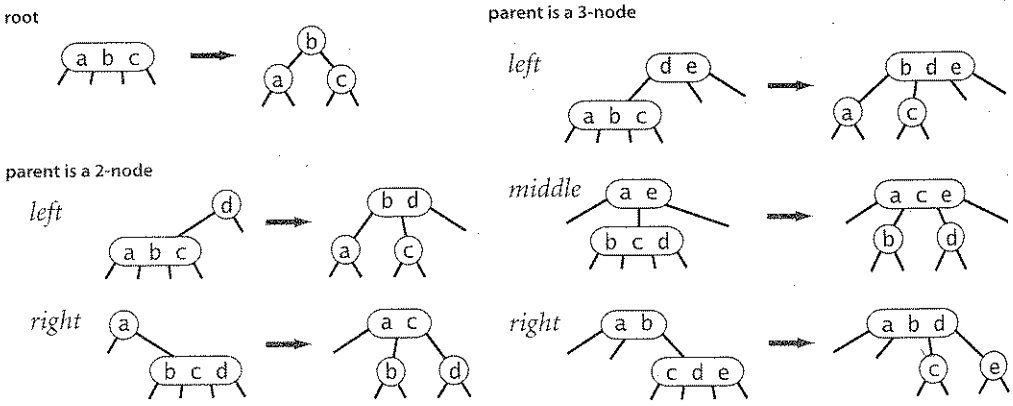
Local transformations. Splitting a temporary 4-node in a 2-3 tree involves one of six transformations, summarized at the bottom of the next page. The 4-node may be the root; it may be the left child or the right child of a 2-node; or it may be the left child, middle child, or right child of a 3-node. The basis of the 2-3 tree insertion algorithm is that all of these transformations are purely *local*: no part of the tree needs to be examined or modified other than the specified nodes and links. The number of

links changed for each transformation is bounded by a small constant. In particular, the transformations are effective when we find the specified patterns *anywhere* in the tree, not just at the bottom. Each of the transformations passes up one of the keys from a 4-node to that node's parent in the tree and then restructures links accordingly, without touching any other part of the tree.



Splitting a 4-node is a local transformation that preserves order and perfect balance

Global properties. Moreover, these *local* transformations preserve the *global* properties that the tree is ordered and perfectly balanced: the number of links on the path from the root to any null link is the same. For reference, a complete diagram illustrating this point for the case that the 4-node is the middle child of a 3-node is shown above. If the length of every path from a root to a null link is h before the transformation, then it is h after the transformation. *Each transformation preserves this property*, even while splitting the 4-node into two 2-nodes and while changing the parent from a 2-node to a 3-node or from a 3-node into a temporary 4-node. When the root splits into three 2-nodes, the length of every path from the root to a null link increases by 1. If you are not fully convinced, work EXERCISE 3.3.7, which asks you to



Splitting a temporary 4-node in a 2-3 tree (summary)

extend the diagrams at the top of the previous page for the other five cases to illustrate the same point. Understanding that every local transformation preserves order and perfect balance in the whole tree is the key to understanding the algorithm.

UNLIKE STANDARD BSTS, which grow down from the top, 2-3 trees grow up from the bottom. If you take the time to carefully study the figure on the next page, which gives the sequence of 2-3 trees that is produced by our standard indexing test client and the sequence of 2-3 trees that is produced when the same keys are inserted in increasing order, you will have a good understanding of the way that 2-3 trees are built. Recall that in a BST, the increasing-order sequence for 10 keys results in a worst-case tree of height 9. In the 2-3 trees, the height is 2.

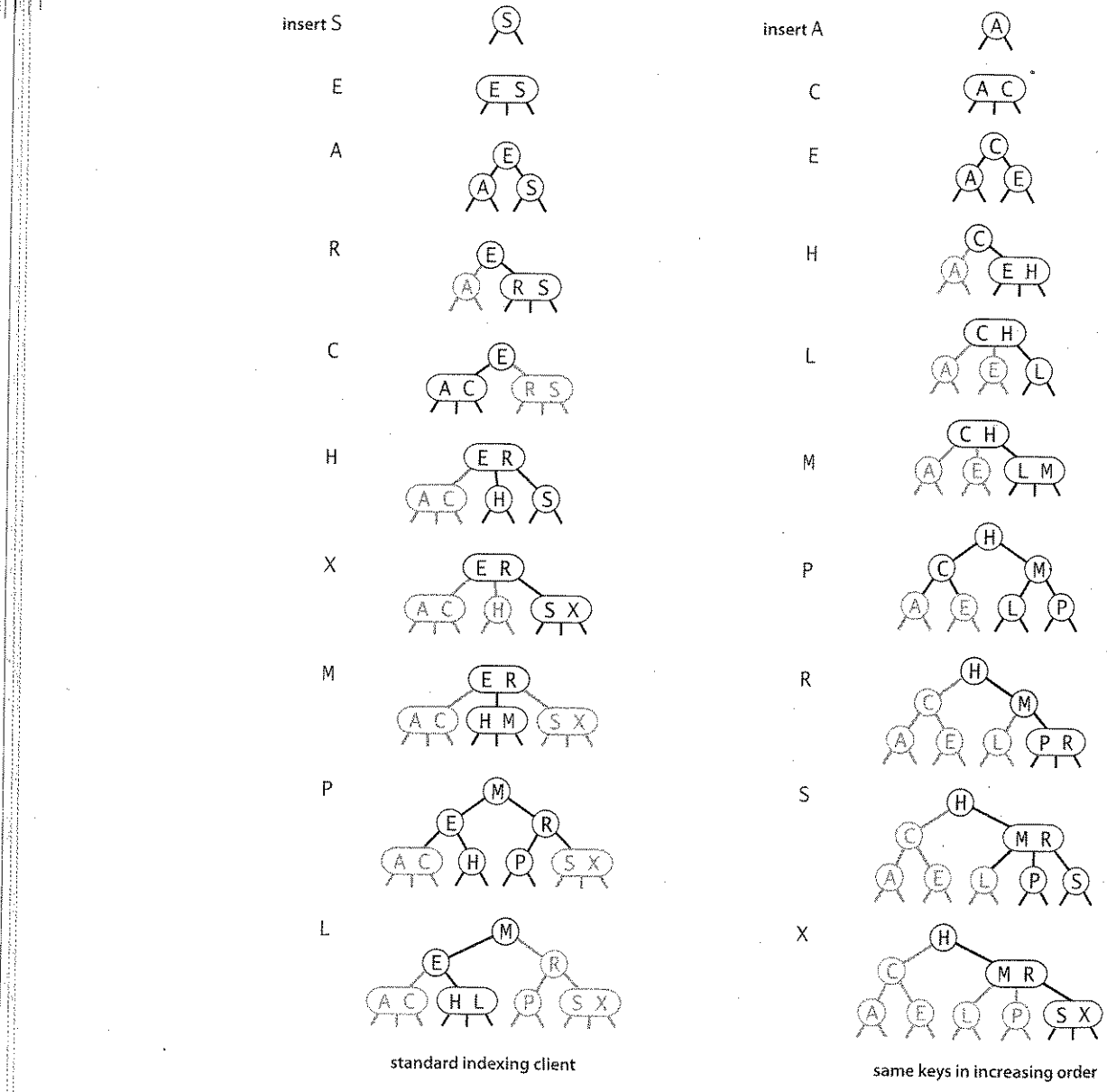
The preceding description is sufficient to define a symbol-table implementation with 2-3 trees as the underlying data structure. Analyzing 2-3 trees is different from analyzing BSTs because our primary interest is in *worst-case* performance, as opposed to average-case performance (where we analyze expected performance under the random-key model). In symbol-table implementations, we normally have no control over the order in which clients insert keys into the table and worst-case analysis is one way to provide performance guarantees.

Proposition F. Search and insert operations in a 2-3 tree with N keys are guaranteed to visit at most $\lg N$ nodes.

Proof: The height of an N-node 2-3 tree is between $\lfloor \log_3 N \rfloor = \lfloor (\lg N)/(\lg 3) \rfloor$ (if the tree is all 3-nodes) and $\lfloor \lg N \rfloor$ (if the tree is all 2-nodes) (see EXERCISE 3.3.4).

Thus, we can guarantee good worst-case performance with 2-3 trees. The amount of time required at each node by each of the operations is bounded by a constant, and both operations examine nodes on just one path, so the total cost of any search or insert is guaranteed to be logarithmic. As you can see from comparing the 2-3 tree depicted at the bottom of page 431 with the BST formed from the same keys on page 405, a perfectly balanced 2-3 tree strikes a remarkably flat posture. For example, the height of a 2-3 tree that contains 1 billion keys is between 19 and 30. It is quite remarkable that we can guarantee to perform arbitrary search and insertion operations among 1 billion keys by examining at most 30 nodes.

However, we are only part of the way to an implementation. Although it is possible to write code that performs transformations on distinct data types representing 2- and 3-nodes, most of the tasks that we have described are inconvenient to implement in

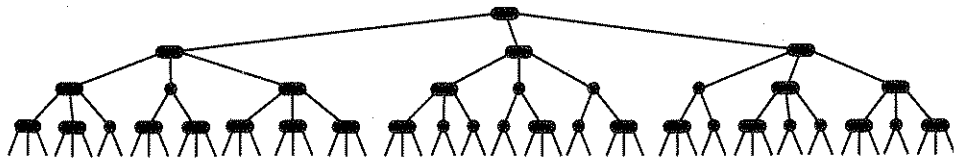


standard indexing client

same keys in increasing order

2-3 construction traces

this direct representation because there are numerous different cases to be handled. We would need to maintain two different types of nodes, compare search keys against each of the keys in the nodes, copy links and other information from one type of node to another, convert nodes from one type to another, and so forth. Not only is there a substantial amount of code involved, but the overhead incurred could make the algorithms slower than standard BST search and insert. The primary purpose of balancing is to provide insurance against a bad worst case, but we would prefer the overhead cost for that insurance to be low. Fortunately, as you will see, we can do the transformations in a uniform way using little overhead.



Typical 2-3 tree built from random keys