Advanced Tree Structures – Self-Adjusting Trees

Introduction

The main concern with search trees is to maintain balance to keep them from becoming skewed (lopsided) and ideally, to allow leaves to occur on only one or two levels. Therefore, if a newly arriving element endangers the balance of the tree, the potential imbalance is resolved either by restructuring the tree locally (the AVL and red-black technique) or by re-building the tree (the DSW technique). The reason for balancing the tree is to ensure that the height of the tree is logarithmic in terms of the number of elements in the tree. This of course, puts an upper bound on the complexity of the search, insert and deletion operations which is also logarithmic in terms of the number of elements in the search tree.

The question now becomes, "is this the only way to improve the performance of these operations in a search tree?" From our earlier discussions concerning self-organizing lists, I hope that you answered the question negatively. Recall that the self-organizing lists restructured themselves based upon the access patterns to the data space. The overall length of the structure was not an issue, rather how much of that length needed to be searched for a given operation. AVL trees and red-black trees restructure based only on insertion and deletion and never on a search operation. Therefore, the access pattern in not an issue in either of these trees. Analogous to the linked list situation, the issue in searching, inserting, and deleting in a search tree does not need to be concerned with the shape of the tree but rather how fast these operations can be performed.

Before we go any further, consider the following extreme case. Suppose we build a search tree from the following data: 10, 20, 30, 40, 50, 60, 70. If the tree is built from the data in the order it arrives, the tree will be a right-skewed tree (in fact it will be a linked list). If our access pattern were to involve 80% searches for data element 10, we would in fact, see a better overall access time if we leave the tree in this skewed format than if we were to balance the tree!

The basic idea behind a self-adjusting tree is the same as that for a self-adjusting list. Move the elements which are accessed most frequently close to the root of the tree. Using this technique, the shape of the tree, i.e., how well it is balanced, becomes less of an issue as elements which are infrequently accessed may be quite deep in the unbalanced tree, but the infrequency of access does not constitute a serious penalty for the access.

There are many different techniques for self-adjusting trees. A common technique which is analogous to the count method for linked lists is to place a counter field in each node of the tree with the most frequently accessed nodes moving up the tree. Typically, no restructuring occurs on an insert. While this may occasionally promote an element to a position higher in the tree than it should have based upon the access pattern, it will, in time, find its proper position. Overall, the higher the frequency of access, the higher up the tree the element will be located. For the most part, the most frequently accessed elements will populate the first few levels of the tree.

Self-Restructuring Search Trees

A strategy proposed by Brian Allen, Ian Munro, and James Bitner consists of two possibilities:

1. Single rotation: Rotate a child about its parent if an element in a child is accessed (unless it is the root).
2. Move to the root: Repeat the child-parent rotation until the element being accessed is in the root.

Under the first strategy, frequently accessed elements will eventually move up close to the root so that subsequent accesses are faster than previous ones. Under the second strategy, it is assumed that the element being accessed has a very high probability of being accessed again, so it is moved directly to the root. Even if it turns out not to be used again immediately, the element will still remain close to the root.

As an example of technique #1 consider the following BST:

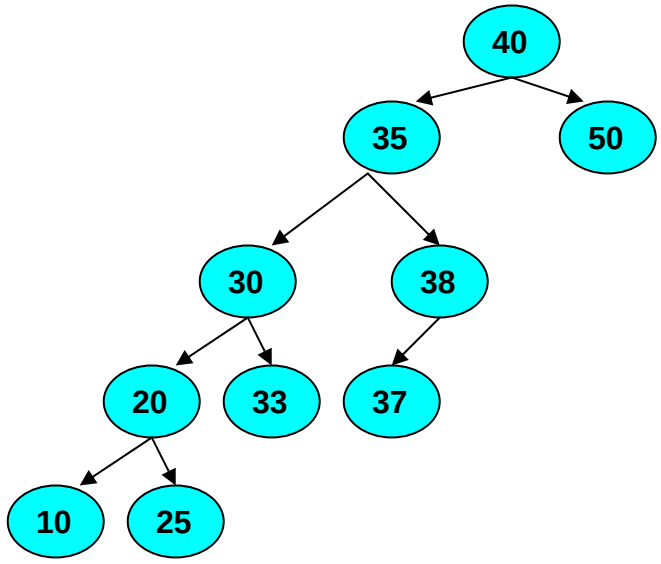

Figure S1 – Initial BST

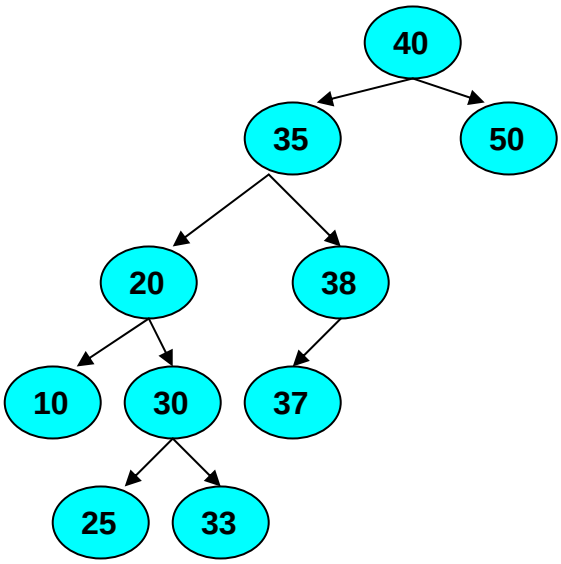

Figure S2 – BST after first search for 20.

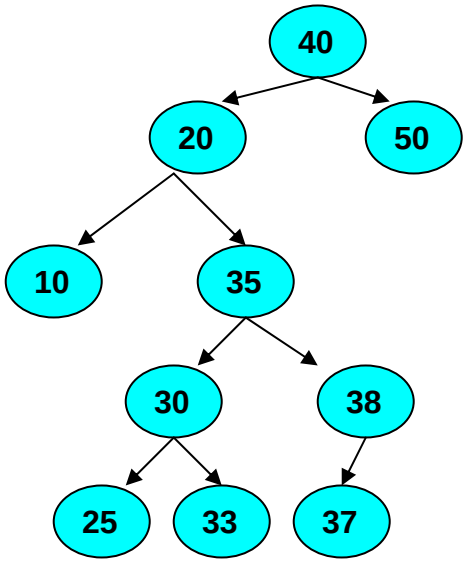

Figure S3 – BST after second search for 20.

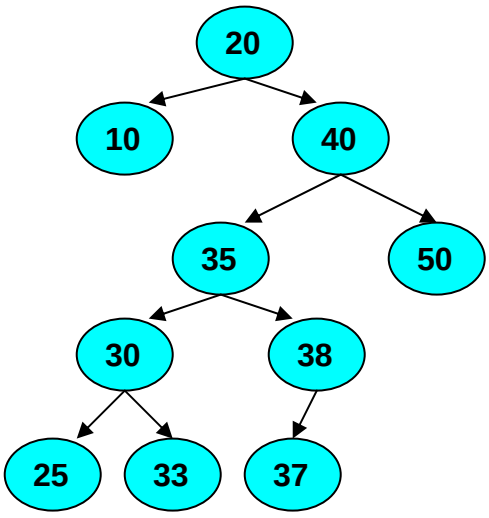

Figure S4 – BST after third search for 20.

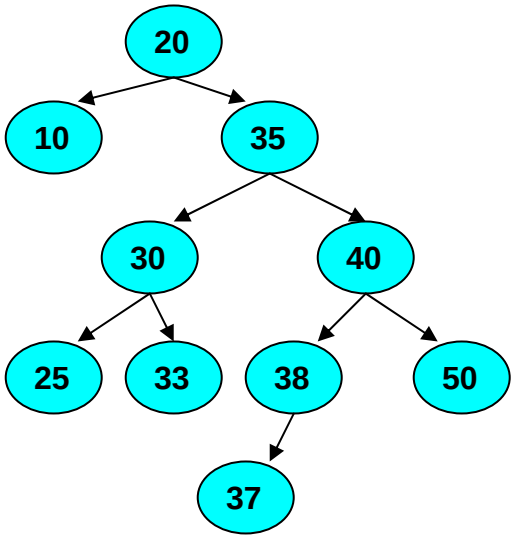

Figure S5 – BST after first search for 35.

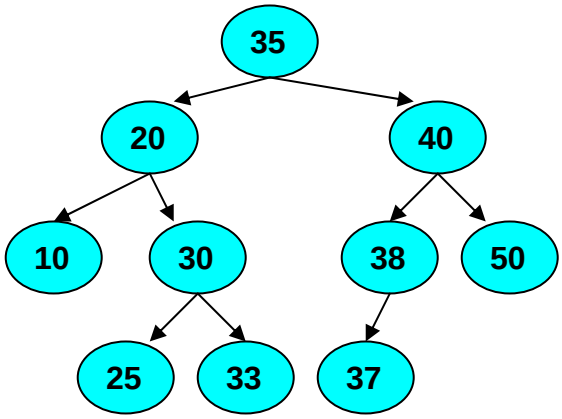

Figure S6 – BST after second search for 35.

Splay Trees

A common application of AVL trees and red-black trees is the implementation of dictionaries. The most common dictionary operations are searching, insertion, and deletion. A *dictionary* is a collection of pairs of the form (*k, e*), where *k* is a *key* and *e* is the *element* associated with the key *k*. (equivalently, *e* is the element whose key is *k*), In most forms of dictionaries, no two pairs will have the same key. The following operations are defined on a dictionary:

get (k, e): Get the element *e* associated with key *k* from the dictionary. This operation is equivalent to a search in the dictionary.

put (k, e): Put the element *e* associated with key *k* into the dictionary. This operation is equivalent to an insertion into the dictionary.

remove (k, e): Remove the element *e* associated with key *k* from the dictionary. This operation is equivalent to a deletion from the dictionary.

No known data structure provides a better worst-case time complexity for these operations than does a dictionary. However, in many applications of a dictionary, we are less concerned with the time required by an individual operation than we are in the time taken by a sequence of operations. This is the case for applications such as histogramming and best-fit bin packing problems (often implemented using search trees with duplicate values). The complexity of these applications depends on the time taken to perform a sequence of dictionary operations, not on the time required for any individual operation.

Splay trees are a variation of binary search trees in which the complexity of an individual dictionary operation is O(*n*). However, every sequence of *get* O(g), *put* O(p), and *remove* O(r) operations is done in O((*g* + *p* + *r*) log *p*) time. This is the same asymptotic complexity as when AVL or red-black trees are used. Empirical results have demonstrated that for random sequences of dictionary operations, splay trees are faster than either AVL trees or red-black trees. As an added bonus, splay trees are easier to code and understand.

The Splay Operation

The normal search tree operations of searching, insertion, and deletion are performed exactly as they are in a normal binary search tree, however, they are followed by a *splay operation* that starts at a *splay node*. When the splay operation terminates, the splay node will have become the root of the search tree. The splay node is selected to be the highest level (i.e., deepest) node, in the resulting tree that was examined (i.e., a comparison was done with the key in this node and the node was either, newly created (insert), removed (delete), or we moved to either the left or right child of this node) during the dictionary operation.

Consider the binary search tree shown in Figure 1.

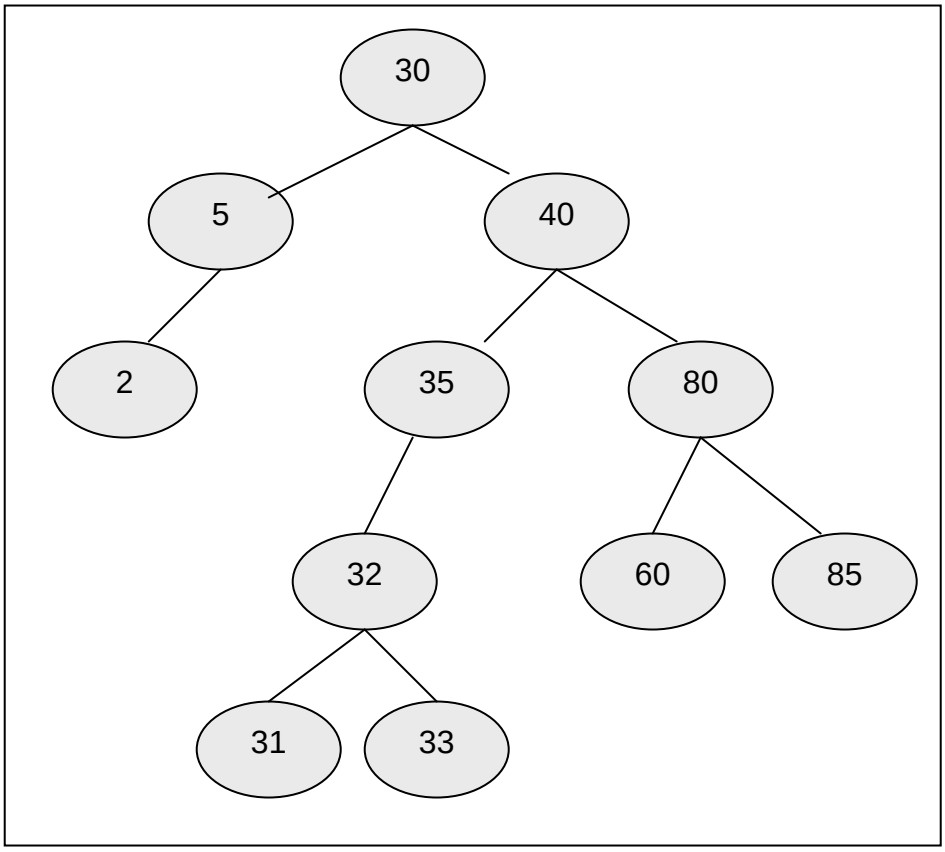

Figure 1. A Binary Search Tree

Suppose that we perform a search (get operation) on the tree of Figure 1 for the value 80. The deepest node in the tree which is examined as part of this search is the node containing 80, since the search is successful; this node becomes the splay node. Similarly, a search of the tree in Figure 1 for the value 31 will be successful and the splay node will be node 31 which is at the deepest possible level in this tree. On the other hand, a search for the value 55 in this tree will be unsuccessful and result in the node containing 60 becoming the splay node.

A put operation (insertion) may create a new node or overwrite the value in an existing node. When a new node is created, this new node will be the deepest node examined and thus becomes the splay node. When an existing element is overwritten, the node containing this overwritten element will be the deepest node examined and thus the splay node. In the tree shown in Figure 1, the splay node for the operation *put*(5, e) is the left child of the root and thus the node containing 5 becomes the splay node. The splay node for the operation *put*(65,e) is the newly created node that would be the right child of the node containing 60.

When the binary search tree has an element with key value *k*, the deepest node examined by a *remove*(k) is the node which is removed from the tree. This node cannot be the splay node as it no longer logically (and perhaps physically as well) exists in the resulting tree. Rather it is the parent of the deleted node that becomes the splay node since it was the deepest node that was examined of those that remain in the tree. For the tree of Figure 1, the splay node for the operation *remove*(33) is the node containing key value 32, which is the parent of the deleted node. Similarly, for the operation *remove*(35) the splay node is the node with key value 40. For the operation *remove*(40) the splay node is the node with key value 30.

The *splay operation* consists of a sequence of *splay steps.* When the splay node is the root of the search tree, this sequence of steps is empty. When the splay node is not the root node, each splay step moves the splay node either one level or two levels up the tree. A one-level move is made only when the splay node is at level two in the search tree (one level below the root).

There are two types of one-level splay steps. One is done when the splay node, let's call it *q*, is the left-child of its parent *p*; the other is done when the splay node *q* is the right-child of its parent *p*. The first type is called a **type L** splay step (equivalent to a right rotation), and the second is called a **type R** splay step (equivalent to a left rotation).

Figure 2, illustrates a **type L** splay step. In Figure 2, nodes *a, b,* and *c,* represent subtrees and the splay node is highlighted in green.

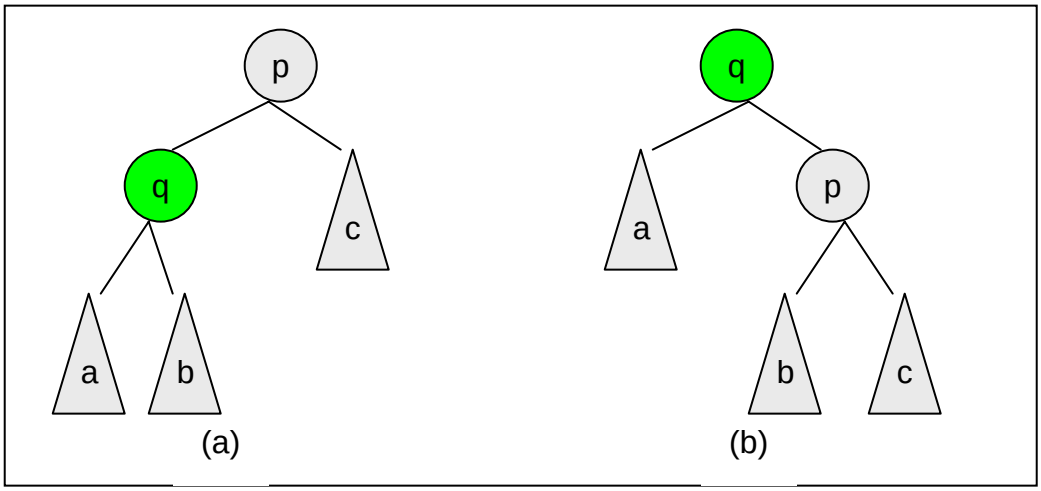

Figure 2. (a) Search tree, splay node in green. (b) Search tree after **type L** splay step.

Notice in Figure 2 that the splay node has become the root. A similar situation is depicted in Figure 3 which illustrates a **type R** splay step.

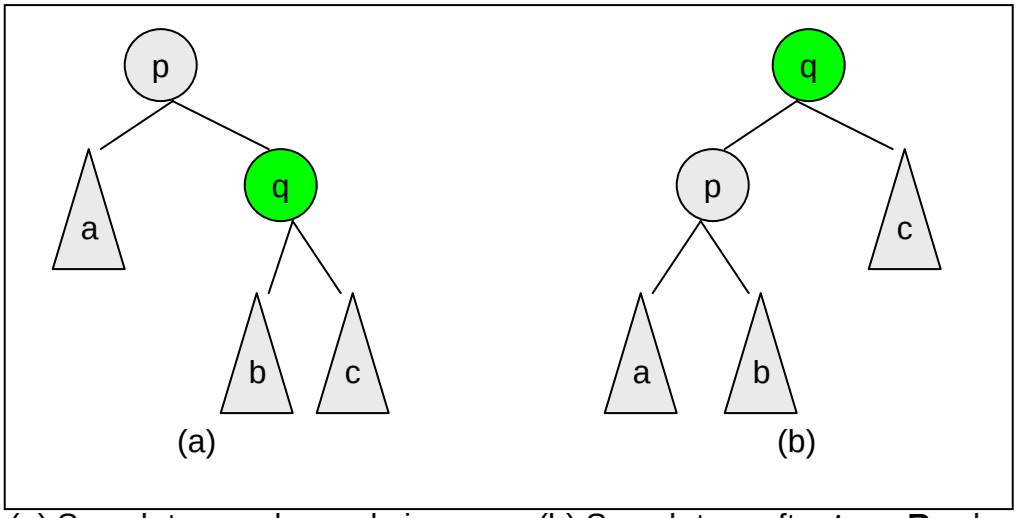

Figure 3. (a) Search tree, splay node in green. (b) Search tree after **type R** splay step.

Two-level splay steps are done when the level of the splay node is greater than 2. Therefore, when a two-level splay step is done, the splay node *q* has a parent *p* and a grandparent *gp*. There are four types of two-level splay steps: **LL, LR, RR,** and **RL** defined as follows:

Type LL: *p* is the left child of *gp*, and *q* is the left child of *p*.
Type LR: *p* is the left child of *gp*, and *q* is the right child of *p*.
Type RR: *p* is the right child of *gp*, and *q* is the right child of *p*.
Type RL: *p* is the right child of *gp*, and *q* is the left child of *p*.

In each of these two-level splay steps, the splay node is elevated two levels in the tree. Figure 4 illustrates the **Type LL** two-level splay step.

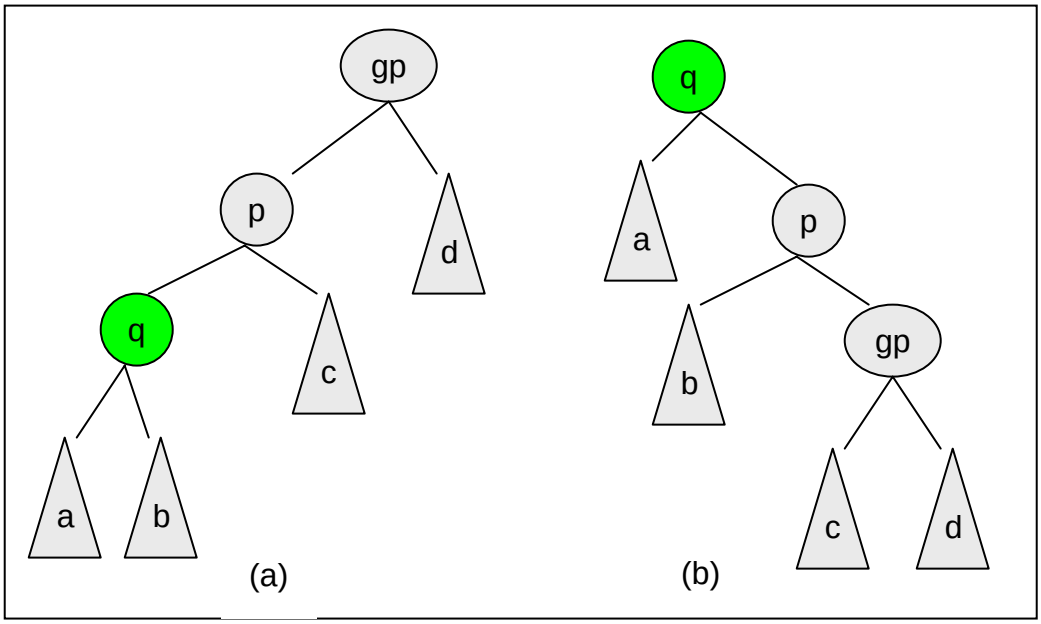

Figure 4. (a) Initial search tree. (b) Search tree after **type LL** splay step.

Figure 5 illustrates the type **LR** two-level splay step.

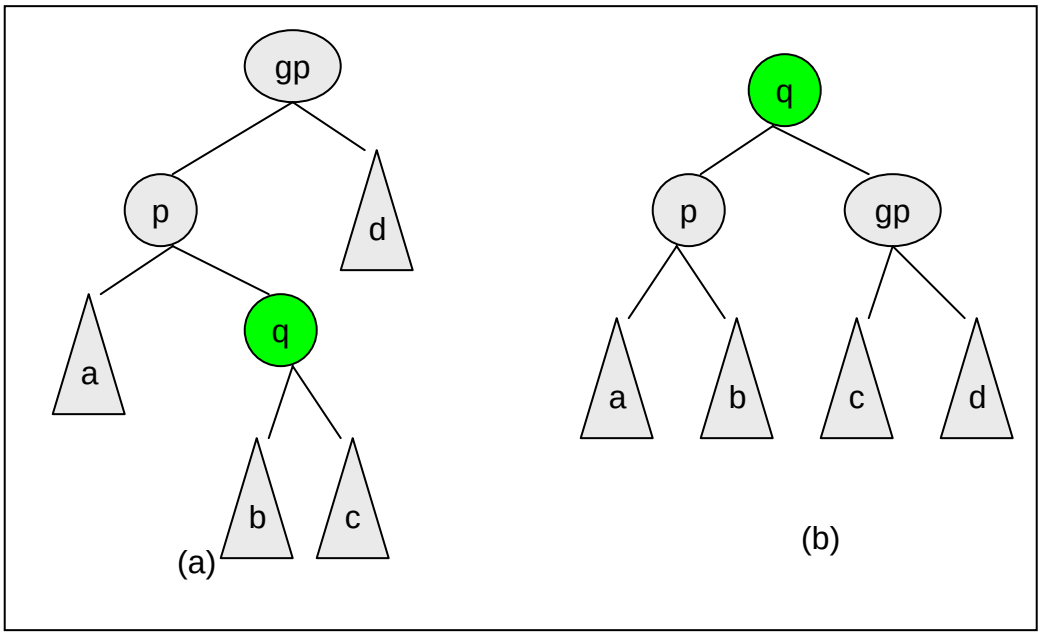

Figure 5. (a) Initial search tree. (b) Search tree after two-level **type LR** splay step.

Figure 6 illustrates the type **RR** two-level splay step. By now you should have noticed the similarity with the various splay steps and the rotations that occur in an AVL tree.

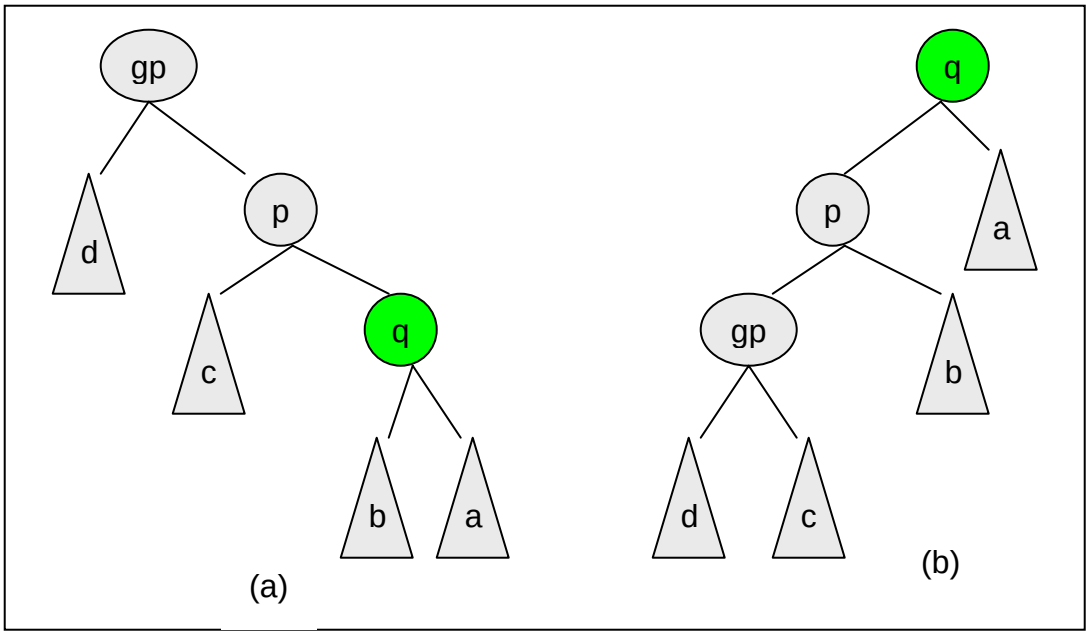

Figure 6. (a) Initial search tree. (b) Search tree after **type RR** two-level splay step.

Figure 7 illustrates the final type of two-level splay step, the type **RL** splay step.

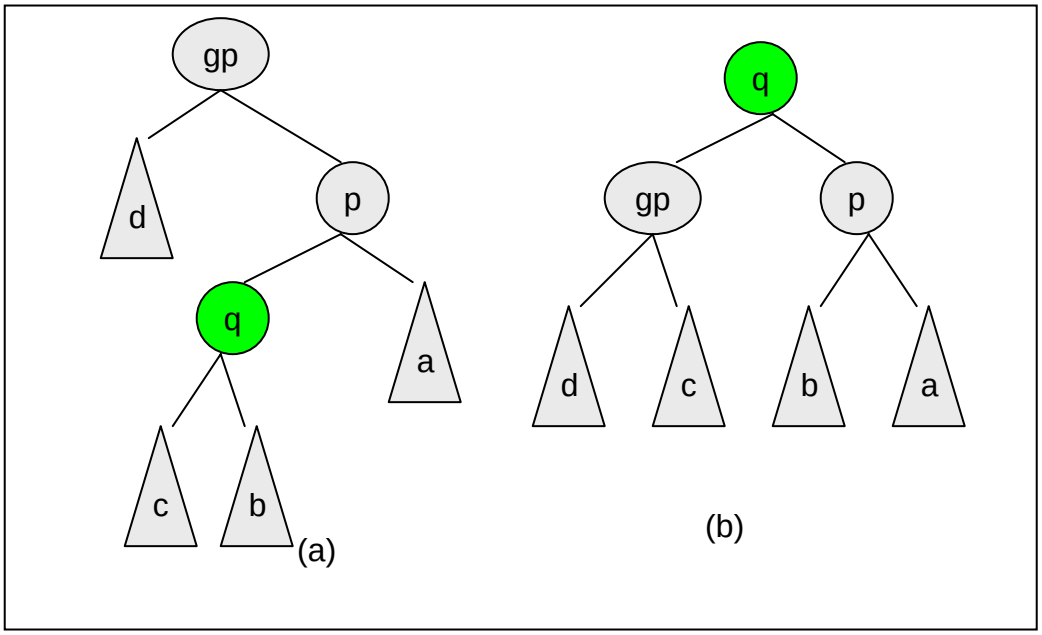

Figure 7. (a) Initial search tree. (b) Search tree after **type RL** two-level splay step.

Example

Assume that the operation *get*(2) is performed on the BST shown in Figure 8. The node containing the key value 2 becomes the splay node since the BST contained the key value specified in the search.

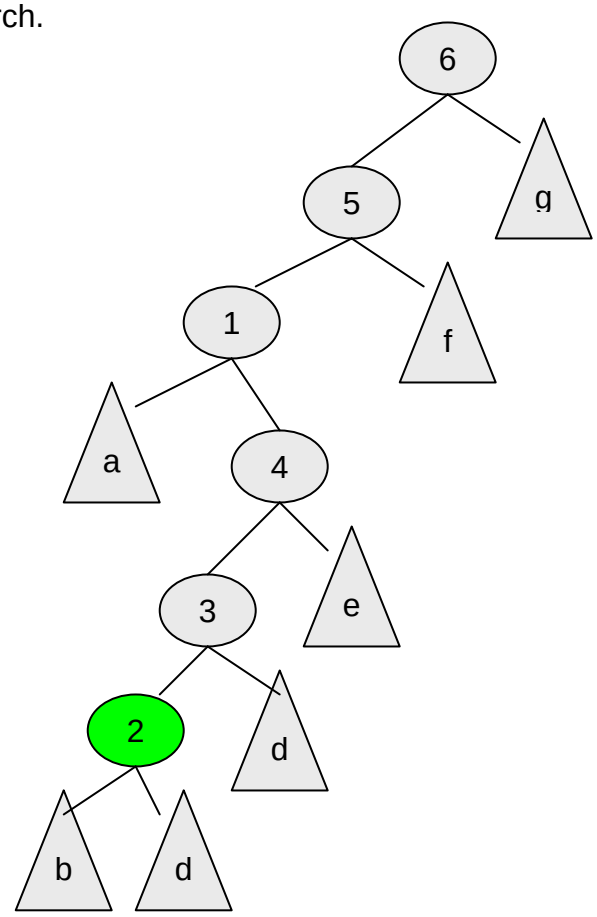

Figure 8. Example BST, splay node from *get*(2) operation shown in green.
This initiates the splay operation which begins by moving the splay node from level 6 to level 4 with the first two-level splay operation which is of type **LL**. This is shown in Figure 9 below.

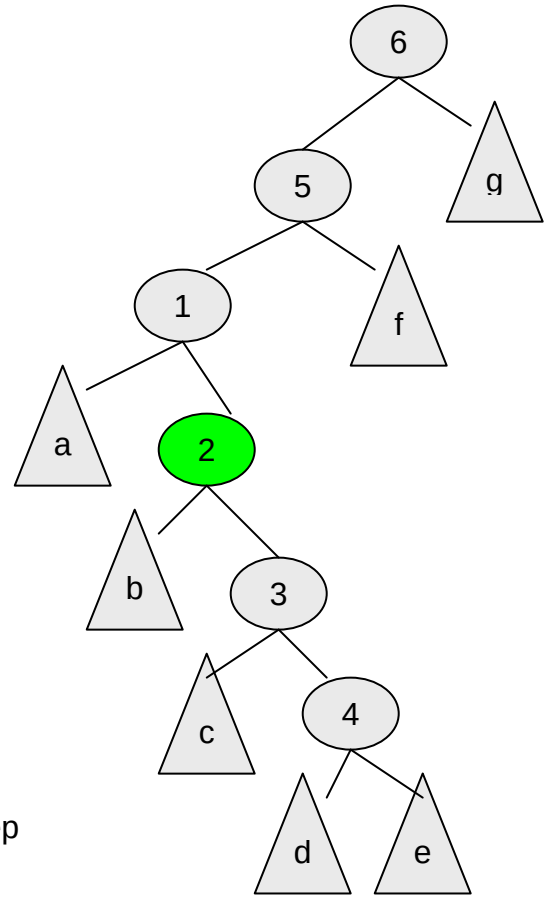

Figure 9. **LL** splay step

The type **LL** splay step shown in Figure 9 is followed by a second two-level splay step to move the splay node up two more levels in the tree from level 4 to level 2. This second splay step is of type **LR** which will produce the BST shown in Figure 10 below.

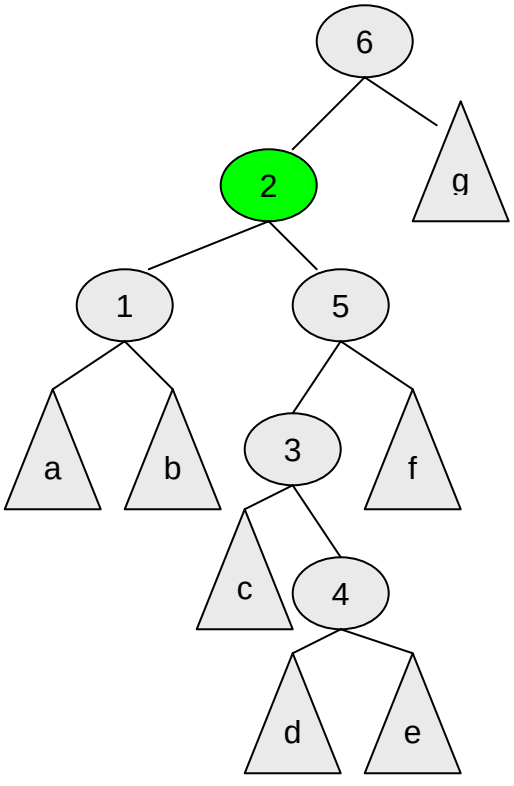

Figure 10. BST after **LR** two-level
splay step, splay node now at level 2.

At this point the splay node is on level 2 so a final one-level splay step is required to move the splay node into the root position of the BST. Since the splay node is a left child of the root, this will be a type **L** splay step and will produce the final BST shown in Figure 11.

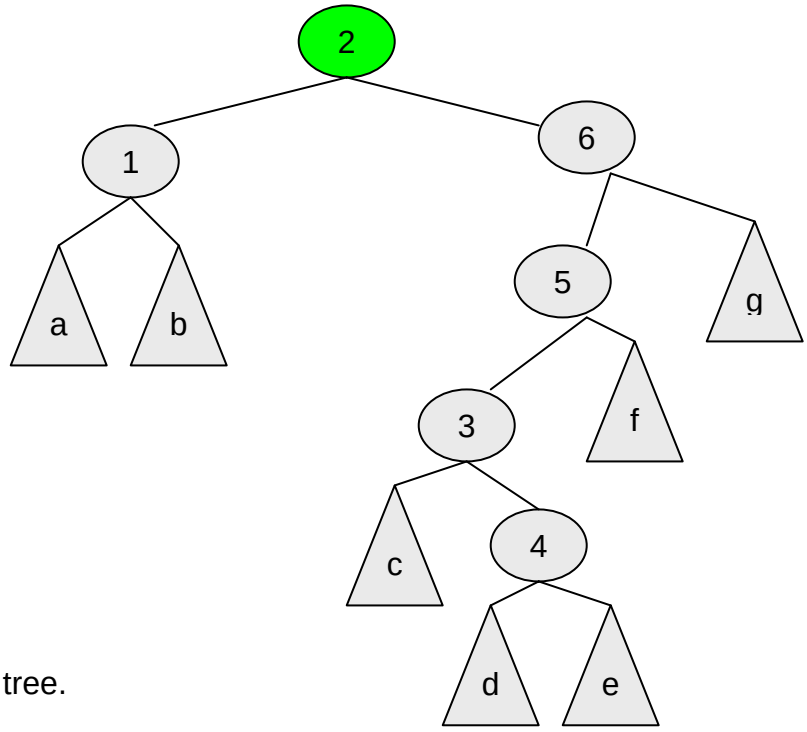

Figure 11. Final splay tree.

Although it is possible to move the splay node from any level to the root (level 1) through a series of one-level splay steps (just type **L** and **R**), limiting a splay operation to a single level does not ensure that every sequence of *get, put,* and *remove* operations will be done in $O((g + p + r) \log p)$ time. To establish this time bound, it is necessary to use a sequence of two-level splay steps which terminate with at most 1 one-level splay step.

Amortized Complexity

Unlike the actual and worst-case complexities of an operation, which are closely related to the step count for that operation, the *amortized complexity* of an operation is an accounting technique that often bears no direct relationship to the actual complexity of that operation. The amortized complexity of an operation can actually be almost anything. The only requirement is that the sum of the amortized complexities of all operations in the sequence be greater than or equal to the sum of their actual complexities. In other words, the following relationship must hold:

$$\sum_{i=1}^{n} \text{amortized(i)} \geq \sum_{i=1}^{n} \text{actual(i)}$$

where *amortized(i)* and *actual(i)*, respectively, denote the amortized and actual complexities of the *i*th operation in a sequence of *n* operations. Due to the this requirement on the sum of the amortized complexities of the operations in any sequence of operations, we can use the sum of the amortized complexities as an upper bound on the complexity of any sequence of operations. [Think of it this way: The amortized cost of an operation is the amount you charge the operation rather than the amount the operation costs. You can charge an operation any amount you want as long as the amount charged to all operations in the sequence is at least equal to the actual cost of the operation sequence.]

What the splaying operation does for you in a splay tree is to amortize the cost of a sequence of operations on the search tree. A splay tree guarantees that *any* sequence of *M* consecutive tree operations (get, put, and remove) take at most O(*M log n*) time. Splay trees are based on the fact that the O(n) worst case time per operation for BSTs is not bad, as long as it occurs relatively infrequently. Any one access, even if it takes O(n) time, is still likely to be quite fast. The problem is that it is not that uncommon to have a sequence of bad accesses take place in which case this worst case run time begins to become quite noticeable. The splaying operation ensures that there are no bad sequences of operations that can occur and results in an O(log n) amortized cost per operation.

With the splay tree, after each access to a node, it is moved to the root position of the tree, no matter how deep it occurred in the tree, through a series of rotations. Notice the effect that this has on all of the nodes along the path from the original root to the splay node. Every one of these nodes moves somewhat closer to the root since the two-level rotations are similar to double rotations in AVL trees. The overall effect is that the splaying operation tends to balance trees which are relatively unbalanced. Consider again, the tree from the example problem above. Notice how much more balanced the final tree is than the original tree, and this is only after a single *get* operation was performed on the tree. Over a long sequence of operations, the tree can become quite well-balanced.

Also note the similarities with splay trees and the self-organizing lists (particularly the move-to-front technique). The overall goal of both of these structures is the same. To reduce the amortized cost per operation for operations performed in sequences on the structure.

A subtle but distinct advantage that a splay tree has over an AVL tree is that no balance factors need to be maintained or calculated for the tree thus, tree height information is not maintained as it is with an AVL tree.

Summary

In our study of advanced tree structures (and advanced list structures) we have seen structures with features that are particularly suited to certain types of applications and care must be made to match the structure with an appropriate application. Failure to properly match the structure with an application may cause unexpected run-time complexities. Table 1 summarizes the worst case and expected run-times for insertion, deletion, and retrieval operations on these advanced structures.

Structure	Worst Case			Expected Case		
	Search	Insert	Delete	Search	Insert	Delete
Skip lists	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n)
BSTs	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n)
AVL trees	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Red-black trees	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Splay trees	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

Table 1. Summary of Operation Complexities for Advanced Data Structures