This chapter introduces several tree structures designed for use in specialized applications. The trie of Section 13.1 is commonly used to store and retrieve strings. It also serves to illustrate the concept of a key space decomposition. The AVL tree and splay tree of Section 13.2 are variants on the BST. They are examples of self-balancing search trees and have guaranteed good performance regardless of the insertion order for records. An introduction to several spatial data structures used to organize point data by \( xy \)-coordinates is presented in Section 13.3.

Descriptions of the fundamental operations are given for each data structure. One purpose for this chapter is to provide opportunities for class programming projects, so detailed implementations are left to the reader.

13.1 Tries

Recall that the shape of a BST is determined by the order in which its data records are inserted. One permutation of the records might yield a balanced tree while another might yield an unbalanced tree, with the extreme case becoming the shape of a linked list. The reason is that the value of the key stored in the root node splits the key range into two parts: those key values less than the root’s key value, and those key values greater than the root’s key value. Depending on the relationship between the root node’s key value and the distribution of the key values for the other records in the tree, the resulting BST might be balanced or unbalanced. Thus, the BST is an example of a data structure whose organization is based on an \textbf{object space decomposition}, so called because the decomposition of the key range is driven by the objects (i.e., the key values of the data records) stored in the tree.

The alternative to object space decomposition is to redefine the splitting position within the key range for each node in the tree. In other words, the root could be predefined to split the key range into two equal halves, regardless of the particular values or order of insertion for the data records. Those records with keys in the lower half of the key range will be stored in the left subtree, while those records
with keys in the upper half of the key range will be stored in the right subtree. While such a decomposition rule will not necessarily result in a balanced tree (the tree will be unbalanced if the records are not well distributed within the key range), at least the shape of the tree will not depend on the order of key insertion. Furthermore, the depth of the tree will be limited by the resolution of the key range; that is, the depth of the tree can never be greater than the number of bits required to store a key value. For example, if the keys are integers in the range 0 to 1023, then the resolution for the key is ten bits. Thus, two keys can be identical only until the tenth bit. In the worst case, two keys will follow the same path in the tree only until the tenth branch. As a result, the tree will never be more than ten levels deep. In contrast, a BST containing \( n \) records could be as much as \( n \) levels deep.

Splitting based on predetermined subdivisions of the key range is called **key space decomposition**. In computer graphics, the technique is known as **image space decomposition**, and this term is sometimes used to describe the process for data structures as well. A data structure based on key space decomposition is called a **trie**. Folklore has it that “trie” comes from “retrieval.” Unfortunately, that would imply that the word is pronounced “tree,” which would lead to confusion with regular use of the word “tree.” “Trie” is actually pronounced as “try.”

Like the B\(^+\)-tree, a trie stores data records only in leaf nodes. Internal nodes serve as placeholders to direct the search process, but since the split points are predetermined, internal nodes need not store “traffic-directing” key values. Figure 13.1 illustrates the trie concept. Upper and lower bounds must be imposed on the key values so that we can compute the middle of the key range. Because the largest value inserted in this example is 120, a range from 0 to 127 is assumed, as 128 is the smallest power of two greater than 120. The binary value of the key determines whether to select the left or right branch at any given point during the search. The most significant bit determines the branch direction at the root. Figure 13.1 shows a **binary trie**, so called because in this example the trie structure is based on the value of the key interpreted as a binary number, which results in a binary tree.

The Huffman coding tree of Section 5.6 is another example of a binary trie. All data values in the Huffman tree are at the leaves, and each branch splits the range of possible letter codes in half. The Huffman codes are actually reconstructed from the letter positions within the trie.

These are examples of binary tries, but tries can be built with any branching factor. Normally the branching factor is determined by the alphabet used. For binary numbers, the alphabet is \{0, 1\} and a binary trie results. Other alphabets lead to other branching factors.

One application for tries is to store a dictionary of words. Such a trie will be referred to as an **alphabet trie**. For simplicity, our examples will ignore case in letters. We add a special character ($) to the 26 standard English letters. The $ character is used to represent the end of a string. Thus, the branching factor for
Figure 13.1 The binary trie for the collection of values 2, 7, 24, 31, 37, 40, 42, 120. All data values are stored in the leaf nodes. Edges are labeled with the value of the bit used to determine the branching direction of each node. The binary form of the key value determines the path to the record, assuming that each key is represented as a 7-bit value representing a number in the range 0 to 127.

each node is (up to) 27. Once constructed, the alphabet trie is used to determine if a given word is in the dictionary. Consider searching for a word in the alphabet trie of Figure 13.2. The first letter of the search word determines which branch to take from the root, the second letter determines which branch to take at the next level, and so on. Only the letters that lead to a word are shown as branches. In Figure 13.2(b) the leaf nodes of the trie store a copy of the actual words, while in Figure 13.2(a) the word is built up from the letters associated with each branch.

One way to implement a node of the alphabet trie is as an array of 27 pointers indexed by letter. Because most nodes have branches to only a small fraction of the possible letters in the alphabet, an alternate implementation is to use a linked list of pointers to the child nodes, as in Figure 6.9.

The depth of a leaf node in the alphabet trie of Figure 13.2(b) has little to do with the number of nodes in the trie, or even with the length of the corresponding string. Rather, a node’s depth depends on the number of characters required to distinguish this node’s word from any other. For example, if the words “anteater” and “antelope” are both stored in the trie, it is not until the fifth letter that the two words can be distinguished. Thus, these words must be stored at least as deep as level five. In general, the limiting factor on the depth of nodes in the alphabet trie is the length of the words stored.

Poor balance and clumping can result when certain prefixes are heavily used. For example, an alphabet trie storing the common words in the English language would have many words in the “th” branch of the tree, but none in the “zq” branch.

Any multiway branching trie can be replaced with a binary trie by replacing the original trie’s alphabet with an equivalent binary code. Alternatively, we can use the techniques of Section 6.3.4 for converting a general tree to a binary tree without modifying the alphabet.
Figure 13.2 Two variations on the alphabet trie representation for a set of ten words. (a) Each node contains a set of links corresponding to single letters, and each letter in the set of words has a corresponding link. “$” is used to indicate the end of a word. Internal nodes direct the search and also spell out the word one letter per link. The word need not be stored explicitly. “$” is needed to recognize the existence of words that are prefixes to other words, such as ‘ant’ in this example. (b) Here the trie extends only far enough to discriminate between the words. Leaf nodes of the trie each store a complete word; internal nodes merely direct the search.
Figure 13.3 The PAT trie for the collection of values 2, 7, 24, 32, 37, 40, 42, 120. Contrast this with the binary trie of Figure 13.1. In the PAT trie, all data values are stored in the leaf nodes, while internal nodes store the bit position used to determine the branching decision, assuming that each key is represented as a 7-bit value representing a number in the range 0 to 127. Some of the branches in this PAT trie have been labeled to indicate the binary representation for all values in that subtree. For example, all values in the left subtree of the node labeled 0 must have value 0xxxxxx (where x means that bit can be either a 0 or a 1). All nodes in the right subtree of the node labeled 3 must have value 0101xxx. However, we can skip branching on bit 2 for this subtree because all values currently stored have a value of 0 for that bit.

Example 13.1 When searching for the value 7 (0000111 in binary) in the PAT trie of Figure 13.3, the root node indicates that bit position 0 (the leftmost bit) is checked first. Because the 0th bit for value 7 is 0, take the left branch. At level 1, branch depending on the value of bit 1, which again is 0. At level 2, branch depending on the value of bit 2, which again is 0. At level 3, the index stored in the node is 4. This means that bit 4 of the key is checked next. (The value of bit 3 is irrelevant, because all values stored in that subtree have the same value at bit position 3.) Thus, the single branch that extends from the equivalent node in Figure 13.1 is just skipped. For key value 7, bit 4 has value 1, so the rightmost branch is taken. Because
this leads to a leaf node, the search key is compared against the key stored in that node. If they match, then the desired record has been found.

Note that during the search process, only a single bit of the search key is compared at each internal node. This is significant, because the search key could be quite large. Search in the PAT trie requires only a single full-key comparison, which takes place once a leaf node has been reached.

Example 13.2 Consider the situation where we need to store a library of DNA sequences. A DNA sequence is a series of letters, usually many thousands of characters long, with the string coming from an alphabet of only four letters that stand for the four amino acids making up a DNA strand. Similar DNA sequences might have long sections of their string that are identical. The PAT trie would avoid making multiple full key comparisons when searching for a specific sequence.

13.2 Balanced Trees

We have noted several times that the BST has a high risk of becoming unbalanced, resulting in excessively expensive search and update operations. One solution to this problem is to adopt another search tree structure such as the 2-3 tree or the binary trie. An alternative is to modify the BST access functions in some way to guarantee that the tree performs well. This is an appealing concept, and it works well for heaps, whose access functions maintain the heap in the shape of a complete binary tree. Unfortunately, requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update, as discussed in Section 10.3.

If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure. The AVL tree works in this way, using insertion and deletion routines altered from those of the BST to ensure that, for every node, the depths of the left and right subtrees differ by at most one. The AVL tree is described in Section 13.2.1.

A different approach to improving the performance of the BST is to not require that the tree always be balanced, but rather to expend some effort toward making the BST more balanced every time it is accessed. This is a little like the idea of path compression used by the UNION/FIND algorithm presented in Section 6.2. One example of such a compromise is called the splay tree. The splay tree is described in Section 13.2.2.
Figure 13.4 Example of an insert operation that violates the AVL tree balance property. Prior to the insert operation, all nodes of the tree are balanced (i.e., the depths of the left and right subtrees for every node differ by at most one). After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

13.2.1 The AVL Tree

The AVL tree (named for its inventors Adelson-Velskii and Landis) should be viewed as a BST with the following additional property: For every node, the heights of its left and right subtrees differ by at most 1. As long as the tree maintains this property, if the tree contains \( n \) nodes, then it has a depth of at most \( O(\log n) \). As a result, search for any node will cost \( O(\log n) \), and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost \( O(\log n) \), even in the worst case.

The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property. Of course, to be practical, we must be able to implement the revised update routines in \( \Theta(\log n) \) time.

Consider what happens when we insert a node with key value 5, as shown in Figure 13.4. The tree on the left meets the AVL tree balance requirements. After the insertion, two nodes no longer meet the requirements. Because the original tree met the balance requirement, nodes in the new tree can only be unbalanced by a difference of at most 2 in the subtrees. For the bottommost unbalanced node, call it \( S \), there are 4 cases:

1. The extra node is in the left child of the left child of \( S \).
2. The extra node is in the right child of the left child of \( S \).
3. The extra node is in the left child of the right child of \( S \).
4. The extra node is in the right child of the right child of \( S \).

Cases 1 and 4 are symmetrical, as are cases 2 and 3. Note also that the unbalanced nodes must be on the path from the root to the newly inserted node.

Our problem now is how to balance the tree in \( O(\log n) \) time. It turns out that we can do this using a series of local operations known as rotations. Cases 1 and
Figure 13.5 A single rotation in an AVL tree. This operation occurs when the excess node (in subtree $A$) is in the left child of the left child of the unbalanced node labeled $S$. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the right child of the right child of the unbalanced node is handled in the same way.

Figure 13.6 A double rotation in an AVL tree. This operation occurs when the excess node (in subtree $B$) is in the right child of the left child of the unbalanced node labeled $S$. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the left child of the right child of $S$ is handled in the same way.

4 can be fixed using a single rotation, as shown in Figure 13.5. Cases 2 and 3 can be fixed using a double rotation, as shown in Figure 13.6.

The AVL tree insert algorithm begins with a normal BST insert. Then as the recursion unwinds up the tree, we perform the appropriate rotation on any node.
that is found to be unbalanced. Deletion is similar; however, consideration for unbalanced nodes must begin at the level of the \texttt{deletemin} operation.

\begin{example}
In Figure 13.4 (b), the bottom-most unbalanced node has value 7. The excess node (with value 5) is in the right subtree of the left child of 7, so we have an example of Case 2. This requires a double rotation to fix. After the rotation, 5 becomes the left child of 24, 2 becomes the left child of 5, and 7 becomes the right child of 5.
\end{example}

\subsection{The Splay Tree}

Like the AVL tree, the splay tree is not actually a distinct data structure, but rather reimplements the BST insert, delete, and search methods to improve the performance of a BST. The goal of these revised methods is to provide guarantees on the time required by a series of operations, thereby avoiding the worst-case linear time behavior of standard BST operations. No single operation in the splay tree is guaranteed to be efficient. Instead, the splay tree access rules guarantee that a series of \( m \) operations will take \( O(m \log n) \) time for a tree of \( n \) nodes whenever \( m \geq n \). Thus, a single insert or search operation could take \( O(n) \) time. However, \( m \) such operations are guaranteed to require a total of \( O(m \log n) \) time, for an average cost of \( O(\log n) \) per access operation. This is a desirable performance guarantee for any search-tree structure.

Unlike the AVL tree, the splay tree is not guaranteed to be height balanced. What is guaranteed is that the total cost of the entire series of accesses will be cheap. Ultimately, it is the cost of the series of operations that matters, not whether the tree is balanced. Maintaining balance is really done only for the sake of reaching this time efficiency goal.

The splay tree access functions operate in a manner reminiscent of the move-to-front rule for self-organizing lists from Section 9.2, and of the path compression technique for managing parent-pointer trees from Section 6.2. These access functions tend to make the tree more balanced, but an individual access will not necessarily result in a more balanced tree.

Whenever a node \( S \) is accessed (e.g., when \( S \) is inserted, deleted, or is the goal of a search), the splay tree performs a process called \texttt{splaying}. Splaying moves \( S \) to the root of the BST. When \( S \) is being deleted, splaying moves the parent of \( S \) to the root. As in the AVL tree, a splay of node \( S \) consists of a series of \texttt{rotations}. A rotation moves \( S \) higher in the tree by adjusting its position with respect to its parent and grandparent. A side effect of the rotations is a tendency to balance the tree. There are three types of rotation.

A \texttt{single rotation} is performed only if \( S \) is a child of the root node. The single rotation is illustrated by Figure 13.7. It basically switches \( S \) with its parent in a
Figure 13.7 Splay tree single rotation. This rotation takes place only when the node being splayed is a child of the root. Here, node $S$ is promoted to the root, rotating with node $P$. Because the value of $S$ is less than the value of $P$, $P$ must become $S$’s right child. The positions of subtrees $A$, $B$, and $C$ are altered as appropriate to maintain the BST property, but the contents of these subtrees remains unchanged. (a) The original tree with $P$ as the parent. (b) The tree after a rotation takes place. Performing a single rotation a second time will return the tree to its original shape. Equivalently, if (b) is the initial configuration of the tree (i.e., $S$ is at the root and $P$ is its right child), then (a) shows the result of a single rotation to splay $P$ to the root.

way that retains the BST property. While Figure 13.7 is slightly different from Figure 13.5, in fact the splay tree single rotation is identical to the AVL tree single rotation.

Unlike the AVL tree, the splay tree requires two types of double rotation. Double rotations involve $S$, its parent (call it $P$), and $S$’s grandparent (call it $G$). The effect of a double rotation is to move $S$ up two levels in the tree.

The first double rotation is called a zigzag rotation. It takes place when either of the following two conditions are met:

1. $S$ is the left child of $P$, and $P$ is the right child of $G$.
2. $S$ is the right child of $P$, and $P$ is the left child of $G$.

In other words, a zigzag rotation is used when $G$, $P$, and $S$ form a zigzag. The zigzag rotation is illustrated by Figure 13.8.

The other double rotation is known as a zigzig rotation. A zigzig rotation takes place when either of the following two conditions are met:

1. $S$ is the left child of $P$, which is in turn the left child of $G$.
2. $S$ is the right child of $P$, which is in turn the right child of $G$.

Thus, a zigzig rotation takes place in those situations where a zigzag rotation is not appropriate. The zigzig rotation is illustrated by Figure 13.9. While Figure 13.9 appears somewhat different from Figure 13.6, in fact the zigzig rotation is identical to the AVL tree double rotation.
Sec. 13.2 Balanced Trees

Figure 13.8 Splay tree zigzag rotation. (a) The original tree with S, P, and G in zigzag formation. (b) The tree after the rotation takes place. The positions of subtrees A, B, C, and D are altered as appropriate to maintain the BST property.

Figure 13.9 Splay tree zigzig rotation. (a) The original tree with S, P, and G in zigzig formation. (b) The tree after the rotation takes place. The positions of subtrees A, B, C, and D are altered as appropriate to maintain the BST property.

Note that zigzag rotations tend to make the tree more balanced, because they bring subtrees B and C up one level while moving subtree D down one level. The result is often a reduction of the tree’s height by one. Zigzig promotions and single rotations do not typically reduce the height of the tree; they merely bring the newly accessed record toward the root.

Splaying node S involves a series of double rotations until S reaches either the root or the child of the root. Then, if necessary, a single rotation makes S the root. This process tends to re-balance the tree. Regardless of balance, splaying will make frequently accessed nodes stay near the top of the tree, resulting in reduced access cost. Proof that the splay tree meets the guarantee of $O(m \log n)$ is beyond the scope of this book. Such a proof can be found in the references in Section 13.4.