Chapters 7 and 8 presented several ideas for how to search in a dictionary, but all of them can degenerate to brute-force sequential search in the worst case. In this chapter we shall see how to build search trees that obey balance conditions that keep them from becoming too scrawny. The balance conditions imply that a search in a tree on \( n \) items can be performed in \( O(\log n) \) time. An obvious use for balanced search trees is to solve the dynamic dictionary problem when we cannot take the chance that the dictionary operations will take longer than time logarithmic in the size of the list.

A more important property of balanced trees, however, is that they let us implement a more powerful data type that supports several operations besides those in a standard dynamic dictionary. This data type is called a sorted list, though it has nothing to do with the ordered list data structure in Section 5.2. We can traverse a sorted list visiting the items in order according to their keys. Sorted lists also let us find the bracketing pair of an item with key \( k \). The bracketing pair contains the two neighbors of \( k \) in the sorted list; if the list does not contain items with equal keys, then the bracketing pair contains the item in the list with the largest key smaller than \( k \), and the item with the smallest key larger than \( k \). Many applications need to find one or both members of an item's bracketing pair.

There are several ways to define balance conditions on trees. We shall see two kinds of balanced trees, then combine some ideas from each to define and implement a third variety. For each kind of balanced tree, we shall consider only how to insert and delete elements. Section 9.4 considers other operations on balanced trees.
An AVL tree is a binary search tree in which the heights of the two child subtrees of any node differ by at most one. The following invariants must be maintained:

1. The tree is a binary search tree.
2. The heights of the two child subtrees of any node differ by at most one.
3. The heights of the two child subtrees of any node differ by at most one.

The height of an AVL tree is defined as the maximum depth of any node's subtrees. The height of an AVL tree is at least 1. A tree with a single node has height 1. An empty tree has height 0.

The balance factor of a node is the difference in height between its left and right subtrees: $\text{bf}(x) = \text{height}(x) - \max\{\text{height}(x.l), \text{height}(x.r)\}$.

An AVL tree is defined as follows:

- An empty tree is an AVL tree.
- A tree with a single node is an AVL tree.
- A tree is an AVL tree if both of its subtrees are AVL trees and the difference in height between its subtrees is at most 1.

The height of an AVL tree is at least $\log_2(n+1)$, where $n$ is the number of nodes in the tree. This is because the height of an AVL tree is at most $\log_2(n+1)$. If the height of an AVL tree is greater than $\log_2(n+1)$, then there must be a node whose subtrees are themselves AVL trees.

To insert a new item into an AVL tree, we first perform a search as if it were a binary search tree. If the item is already present, it is ignored. If the item is not present, we insert it into the tree as a leaf node. If the resulting tree violates the height-balanced property, we perform a series of rotations to restore balance.

The rotations are of three types:

- Left rotation
- Right rotation
- Double rotation

The double rotation is a combination of a left rotation and a right rotation.

The height-balanced property is maintained by ensuring that the heights of the two child subtrees of any node differ by at most one. This is achieved by performing rotations whenever the height difference exceeds one.

The height of an AVL tree is at most $2\log_2(n+1)$, and the height of an AVL tree with $n$ nodes is at least $\log_2(n+1)$.
The AVL tree after insertion of a and b.

Figure 9.2

Reinsert the right subtree of the root as shown in Figure 9.7. p. 197.

The construction of the AVL tree in Figure 9.7 shows the two
nodes (a) and (b). This is an example of double rotation.

Insertion of d into an AVL tree, the tree shown in (e), immediately after the
insertion of c into an AVL tree, the tree shown in (d). Immediate after insertion of
the property, there is no need to obey the AVL property; a double rotation corrects the
violation, since it obeys the AVL rule. Hence, the tree shown in Figure 9.6.

Figure 9.6

Binary search tree shown in Figure 9.4.

In the AVL property, this insertion order produces the sequence.
We illustrate this by inserting the letters a, b, c, e, and d in that
order, into a binary search tree, keeping the tree
shortest. However, when we do this the tree does not obey the AVL tree.

When c is inserted into the tree shown in Figure 9.6, the result is still
an example of a single rotation.

The construction of the AVL tree as shown in Figure 9.7. p. 197.

Insertion of d into an AVL tree, the tree shown in (e), immediately after the
insertion of c into an AVL tree, the tree shown in (d). Immediate after insertion of
the property, there is no need to obey the AVL property; a double rotation corrects the
violation, since it obeys the AVL rule. Hence, the tree shown in Figure 9.6.

Figure 9.6

Binary search tree shown in Figure 9.4.

In the AVL property, this insertion order produces the sequence.
We illustrate this by inserting the letters a, b, c, e, and d in that
order, into a binary search tree, keeping the tree
shortest. However, when we do this the tree does not obey the AVL tree.

When c is inserted into the tree shown in Figure 9.6, the result is still
an example of a single rotation.

The construction of the AVL tree as shown in Figure 9.7. p. 197.

Insertion of d into an AVL tree, the tree shown in (e), immediately after the
insertion of c into an AVL tree, the tree shown in (d). Immediate after insertion of
the property, there is no need to obey the AVL property; a double rotation corrects the
violation, since it obeys the AVL rule. Hence, the tree shown in Figure 9.6.

Figure 9.6

Binary search tree shown in Figure 9.4.

In the AVL property, this insertion order produces the sequence.
We illustrate this by inserting the letters a, b, c, e, and d in that
order, into a binary search tree, keeping the tree
shortest. However, when we do this the tree does not obey the AVL tree.

When c is inserted into the tree shown in Figure 9.6, the result is still
an example of a single rotation.

The construction of the AVL tree as shown in Figure 9.7. p. 197.

Insertion of d into an AVL tree, the tree shown in (e), immediately after the
insertion of c into an AVL tree, the tree shown in (d). Immediate after insertion of
the property, there is no need to obey the AVL property; a double rotation corrects the
violation, since it obeys the AVL rule. Hence, the tree shown in Figure 9.6.
Deletion from an AVL Tree

A double rotation leaves the tree as is (d). There is a symmetric variant of a single rotation.

The result of the tree in Figure 9.6 is shown in (a). The result of a single rotation is shown as (b). There is a symmetric variant of a single rotation.

FIGURE 9.5

We can insert an item into an AVL tree in a node in O(log n) time. Once an AVL tree contains n nodes, we can search for a value x in the tree in O(log n) time using the search path of the AVL property. Since the search path is one single rotation a single operation of double rotation is can office after insertion can occur out of nowhere.

If the tree is as shown in Figure 9.6b, then Figure 9.9 shows how a single rotation restores the AVL property and preserves symmetry of the tree.

Before insertion, the tree was an AVL tree. After insertion, the tree may no longer be an AVL tree. If we rearrange the tree to an AVL tree, we can search for a single rotation a single rotation out of nowhere.

FIGURE 9.6
A 2.4 tree is a tree with the following properties:

Interpretation of non-binary leaves:

If a child node has a height k node depth have on operations and natural section at that height, then the tree height k + 1 of the tree is minimum.

There is no need to restrict ourselves to binary trees for candidates for

Properties 3 & 4 make searching in a 2.4 tree only slightly more

Figure 9.13 shows an example of a 2.4-tree. This tree on 22 items is

4 items in the leaves appear in order from left to right.

The leaves in the leaves appear in order from left to right.

Suppose we delete 1 from the AVL tree in Figure 9.2. During the

Suppose we delete 1 from the AVL tree in Figure 9.2. During the

The AVL property.

Theorem 9.12 shows the two conditions that are needed to restore the

Theorem 9.12 shows the two conditions that are needed to restore the

From an AVL tree on n nodes in O(log n) time.

Since the search path is of length O(log n) and we perform at

AVL Property.

Properties 1 & 2 require that we need to perform a rotation

Properties 1 & 2 require that we need to perform a rotation

The AVL property. This will proceed much as in the case of insert.

We must perform two double rotations if shown in (b) and (c).

Property (a) shows the tree of Figure 9.2 after 1 is deleted. To restore the AVL

Property (a) shows the tree of Figure 9.2 after 1 is deleted. To restore the AVL
**Deletion from a 2,4 Tree**

In a binary search tree, deletion is more complicated than insertion. The first step is to search for the item. After the item is detected, we must stop since the length of the access path is \(O(n)\), and a split requires \(\log(n)\) time.

Since the length of the access path is \(O(n)\), we can insert an item into a 2,4 tree.

In a binary search tree, deletion is into a 2,4 tree begins with a.

A 2,4 Tree insertion takes \(O(n)\) time.

### Properties of 2,4 Trees

- Each 2-node has at most 2 keys.
- Each 4-node has at most 4 keys.
- Each 2-node has at most 2 children.
- Each 4-node has at most 4 children.

In a 2,4 tree, child of the 4-node \(x\), which contains the item with key \(x\), because \(p > x > t\). Finally, we would go to the middle.

When the right subtree is a 2,4 tree on 13 items.

A 2,4 tree on 22 items. The left subtree of the root is a 2,4 tree on 7 items.
The text on the page discusses the implementation of red-black trees, focusing on the properties and algorithms involved in maintaining balance in these tree structures. It includes diagrams illustrating the operations and conditions necessary for maintaining the red-black tree properties, such as the balance of black height and the absence of red-red violations. The text is technical and detailed, aiming to provide a clear understanding of how red-black trees work and how they are used in various data structures and algorithms.
Our implementation uses a header node head, whose right child

defines a color, a search key value, and pointers
to its left and right children; a node whose left child points to itself is
a red-black tree node has a color, a search key value, and pointers
to its left and right children.

9.14 shows the definitions for our implementation of red-black trees.

A red-black tree on nine items whose black height is two.

DEFINITIONS

node head;

(node root context) (head)

node head;

(lambda (node (root context) (head))

(node root context) (head)

(type node head)

(boolean root red?)

(type strict node head)

((A) -> (x)) (A x) (A x)

typehead the kind!

(type strict node head)

(def the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)

(type the empty red-black tree)
Red-black trees are similar enough to 24 trees that you can probably
see how to insert in them and report imbalance during a walk back
into a Red-black Tree

FIGURE 9.19

Data Structure view of the ''empty'' red-black tree constructed by insert().

FIGURE 9.18

9.3 Red-black Trees

PROGRAM 9.1b

```c
node = NULL;
node = newNode(MINSENTINEL);
head = node;
```

```c
true
'

if (parent != NULL) {
    if (parent->left == NULL)
        parent->left = node;
    else
        parent->right = node;
}
```

```c
Program 9.1b

```
Figures 9.21 and 9.22. These figures illustrate the deletion of a node from a red-black tree. The tree is modified by removing the node and then re-balancing the tree to maintain the red-black properties.

The process involves adjusting the tree to ensure that the properties of red-black trees are preserved after the deletion. The adjustments may include recoloring nodes and possibly swapping or rotating branches to maintain the balance.

The diagrams show the tree before and after the deletion, with labels indicating the changes made. The red-black properties, such as the color of nodes and the balance of paths, are evident in the illustrations.
If important to note that in this section, the root black color is denoted by a red-black."}

The root block (represented by the root node) of the red-black tree is the first node considered in the algorithm. When the root node is red or black, it is denoted by a red-black tree. If the root is red, then the tree is considered to be a red-black tree. If the root is black, then the tree is considered to be a red-black tree. The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.

The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.

The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.

The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.

The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.

The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.

The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.

The red-black tree is defined as a binary search tree in which every node is colored either red or black. The root node is always black. Each node is considered to be a root node. The root node is colored red if it has no children. The root node is colored black if it has at least one child. The red-black tree is maintained by recursively applying the properties of the red-black tree to the root node. The red-black tree is maintained by ensuring that the root node is always black.
The idea behind the deletion algorithm for red-black trees is just the opposite of the movement for the insertion algorithm. We would be deleting a node, while in the insertion algorithm, we added a new node. One way to understand this transformation is to imagine that if we had a single deletion operation in a red-black tree, it would change the tree from a red node into a black node. Since we would like to keep the tree balanced and ensure that the balance condition is maintained, we proceed with the deletion operation by splitting the tree into two subtrees, the left subtree and the right subtree. If necessary, we rotate the subtrees to maintain the balance condition.

In the simpler kind, the current node and the parent are both black, then the idea is to delete the node. If the deleted node has only one child, we replace the deleted node with its child. If the deleted node has two children, we find the node with the smallest value in the right subtree (or largest value in the left subtree) and replace the deleted node with this node. We then delete the node that was found, as it is guaranteed to be red.

Function for insertion into a red-black tree:

```c
void insert(x) { /* treat red as black */
  node = find(x);
  if (node == leaf) { /* new node */
    node = new node;
    node->color = black;
    /* node has no children */
    return;
  }
  /* node has one child */
  node = split(node);
  if (node->parent->color == black)
    return;
  /* node has two children */
  node = find(node->parent);
  if (node->parent->parent->color == black) { /* new black node */
    node->parent->color = black;
    node = node->parent;
    if (node->left->parent->color == black)
      return;
    node = node->left;
    node->left = split(node); /* node has no children */
    return;
  } /* node->parent->parent->color != black */
  node = node->parent;
  node->parent = split(node);
  node->parent = split(node); /* node has no children */
  return;
}
```

One case red stepped too high:

```c
/* red stepped too high */
int main() {
  node = find(x);
  if (node == leaf) { /* new node */
    node = new node;
    node->color = black;
    /* node has no children */
    return;
  }
  /* node has one child */
  node = split(node);
  if (node->parent->color == black)
    return;
  /* node has two children */
  node = find(node->parent);
  if (node->parent->parent->color == black) { /* new black node */
    node->parent->color = black;
    node = node->parent;
    if (node->left->parent->color == black)
      return;
    node = node->left;
    node->left = split(node); /* node has no children */
    return;
  } /* node->parent->parent->color != black */
  node = node->parent;
  node->parent = split(node);
  node->parent = split(node); /* node has no children */
  return;
}
```
Some red-black trees are represented by their definitions. Each tree consists of nodes connected by edges, where each node is either red or black. The root of the tree is always black, and every path from the root to a leaf node contains the same number of black nodes. Red-black trees are often used in data structures like sets and maps because they maintain balance to ensure fast search times.

The algorithm for inserting a new element into a red-black tree involves several steps:
1. Insert the new node as in a binary search tree.
2. Color the new node red (this makes it a red-black tree by default).
3. If the parent of the new node is black, the tree is already balanced.
4. If the parent of the new node is red, it may violate the red-black tree properties. Adjust the tree accordingly to maintain balance.
5. Repeat steps 3 and 4 if necessary, moving up the tree until the root is reached.

After each insertion, the tree may become unbalanced, but the red-black tree properties ensure that the height of the tree remains logarithmic in the number of nodes, leading to efficient search and other operations.

Figure 9.24 illustrates the process of inserting a new node into a red-black tree. The figure shows the steps involved in maintaining the tree's properties after an insertion. The text also discusses the implications of using red-black trees in implementing hash tables, where the balanced nature of the tree allows for efficient lookups.

Program 9.16 is a C program to define the red-black tree data structure and provide functions to maintain its properties.

The second part of the definition for a red-black tree is shown in the code, which includes methods for insertion, deletion, and other operations. The code ensures that the red-black tree properties are maintained throughout these operations.

In summary, red-black trees are a type of balanced tree structure that guarantees logarithmic time complexity for basic operations, making them a powerful tool in various applications.
When a sorted list is so large that it cannot be stored entirely in memory, a min heap is worth considering. Typically we choose a min heap because it is efficient for managing a priority queue. A min heap is a tree whose nodes are arranged in a binary tree, and each node has a value that is less than or equal to the values of its children. The tree is complete, meaning that all levels of the tree are filled except possibly the last, which is filled from left to right.

If a node is removed from the heap, we need to maintain the heap property. We can do this by performing a down-heap operation. We compare the node with its children and swap it with the smaller child. This process is repeated until the heap property is satisfied.

In this chapter, we will discuss the implementation of a heap, which can be used to implement a priority queue. We will also discuss the advantages and disadvantages of using a heap.

Self-Adjusting Trees

Balanced trees provide several advantages over unbalanced trees. In a balanced tree, the height is logarithmic in the number of nodes, which means that the worst-case time complexity for operations like search, insert, and delete is O(log n). This is in contrast to unbalanced trees, where the worst-case time complexity for these operations can be O(n).

Although the chapter introduction mentioned the data type sorted tree, it is not necessary to perform any operations on balanced trees. Therefore, this section will not cover further topics related to balanced trees.

Other Topics

Out of Print
4. Explain why the subtrees in the subtree $h+2$ have height $h+1$.

$9$. Can there be multiple trees on a node that have height $h$?

1. Verify that the three $h$-trees are all AVL trees.

2. Explain how these particular trees appear.

3. How do $h$-trees work in general?

4. What is the height of an AVL tree of order $n$?

5. Is it possible to delete a node from an AVL tree?

6. How does an AVL tree remain balanced after a deletion?

7. What is the significance of the height of an AVL tree?

8. How do we maintain the balance of an AVL tree during insertions?

9. What properties must an AVL tree satisfy?

10. How does an AVL tree ensure balanced height?

11. What is the maximum height of an AVL tree with $n$ nodes?

12. How do we perform a search in an AVL tree?

13. What is the time complexity of searching in an AVL tree?

14. How do we insert a new node into an AVL tree?

15. What is the time complexity of inserting a new node into an AVL tree?

16. How do we delete a node from an AVL tree?

17. What is the time complexity of deleting a node from an AVL tree?

18. How do we maintain the balance of an AVL tree after a deletion?

19. What is the significance of the balance factor in an AVL tree?

20. How do we ensure the balance of an AVL tree after an insertion?

21. What is the difference between an AVL tree and a binary search tree?

22. How does an AVL tree maintain its balance?

23. What is the importance of maintaining the balance of an AVL tree?

24. How do we perform a search in an AVL tree with $n$ nodes?

25. What is the time complexity of searching in an AVL tree with $n$ nodes?

26. How do we insert a new node into an AVL tree with $n$ nodes?

27. What is the time complexity of inserting a new node into an AVL tree with $n$ nodes?

28. How do we delete a node from an AVL tree with $n$ nodes?

29. What is the time complexity of deleting a node from an AVL tree with $n$ nodes?

30. How do we maintain the balance of an AVL tree with $n$ nodes after a deletion?

31. What is the difference between an AVL tree and a binary search tree with $n$ nodes?

32. How does an AVL tree maintain its balance with $n$ nodes?

33. What is the importance of maintaining the balance of an AVL tree with $n$ nodes?

34. How do we perform a search in an AVL tree with $n$ nodes?

35. What is the time complexity of searching in an AVL tree with $n$ nodes?

36. How do we insert a new node into an AVL tree with $n$ nodes?

37. What is the time complexity of inserting a new node into an AVL tree with $n$ nodes?

38. How do we delete a node from an AVL tree with $n$ nodes?

39. What is the time complexity of deleting a node from an AVL tree with $n$ nodes?

40. How do we maintain the balance of an AVL tree with $n$ nodes after a deletion?
In Chapter 4 of the textbook, "Data Structures and Algorithms with Applications Part B," the author discusses the organization and implementation of data structures. Specifically, the chapter focuses on the use of self-balancing binary search trees, which are a type of data structure that maintains a balanced tree height to ensure efficient search, insertion, and deletion operations.

1. Explain why we must require that 2 ≤ b ≤ 2a + 1 in a red-black tree.

2. Draw the 2^2 red-black tree that corresponds to Figure 1.16 backward.

3. Draw the 2^3 red-black tree that corresponds to Figure 1.16 forward.

4. Explain why we must require that 2 ≤ b ≤ 2a + 1 in a red-black tree.

5. To maintain an AVL tree, one must use the relative heights of each node in an AVL tree to maintain a 2^4 red-black tree by keeping at each internal node both the same height or that the right subtree is one tall, and one node in an AVL tree's balance factor is one tall.

6. The recursive operations for AVL trees need to be balanced operations during the whole round the root.

7. To maintain an AVL tree, one must use the relative heights of each node in an AVL tree by maintaining a 2^4 red-black tree by keeping at each internal node both the same height or that the right subtree is one tall, and one node in an AVL tree's balance factor is one tall.

8. Draw the 2^4 red-black tree that corresponds to Figure 1.16 back.

9. Draw the 2^4 red-black tree that corresponds to Figure 1.16 forward.

10. When are the symmetric versions of the trees in Figure 8.27 relevant?

11. Insert an integer into a red-black tree in 27 steps.

12. Show how to maintain a 2^4 red-black tree by keeping at each internal node both the same height or that the right subtree is one tall, and one node in an AVL tree's balance factor is one tall.

13. Show how to maintain an AVL tree by keeping at each internal node both the same height or that the right subtree is one tall, and one node in an AVL tree's balance factor is one tall.

14. Show how to maintain a 2^4 red-black tree by keeping at each internal node both the same height or that the right subtree is one tall, and one node in an AVL tree's balance factor is one tall.

15. Show how to maintain a 2^4 red-black tree by keeping at each internal node both the same height or that the right subtree is one tall, and one node in an AVL tree's balance factor is one tall.

REFERENCES
Queues
Priority

10