1. In Ch. 4 of *Weiss*, Qs 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.8, 4.9

2. Qs 4.14, 4.19, 4.25

3. In Ch. 5, read §5.5 (as well as §5.1 - 5.4)

Scans of questinos from book follow.

Summary

We have seen uses of trees in operating systems, compiler design, and searching. Expression trees are a small example of a more general structure known as a **parse tree**, which is a central data structure in compiler design. Parse trees are not binary, but are relatively simple extensions of expression trees (although the algorithms to build them are not quite so simple).

Search trees are of great importance in algorithm design. They support almost all the useful operations, and the logarithmic average cost is very small. Nonrecursive implementations of search trees are somewhat faster, but the recursive versions are sleeker, more elegant, and easier to understand and debug. The problem with search trees is that their performance depends heavily on the input being random. If this is not the case, the running time increases significantly, to the point where search trees become expensive linked lists.

We saw several ways to deal with this problem. AVL trees work by insisting that all nodes' left and right subtrees differ in heights by at most one. This ensures that the tree cannot get too deep. The operations that do not change the tree, as insertion does, can all use the standard binary search tree code. Operations that change the tree must restore the tree. This can be somewhat complicated, especially in the case of deletion. We showed how to restore the tree after insertions in $O(\log N)$ time.

We also examined the splay tree. Nodes in splay trees can get arbitrarily deep, but after every access the tree is adjusted in a somewhat mysterious manner. The net effect is that any sequence of M operations takes $O(M \log N)$ time, which is the same as a balanced tree would take.

B-trees are balanced M-way (as opposed to 2-way or binary) trees, which are well suited for disks; a special case is the 2–3 tree ($M = 3$), which is another way to implement balanced search trees.

In practice, the running time of all the balanced tree schemes, while slightly faster for searching, is worse (by a constant factor) for insertions and deletions than the simple binary search tree, but this is generally acceptable in view of the protection being given against easily obtained worst-case input. Chapter 12 discusses some additional search tree data structures and provides detailed implementations.

A final note: By inserting elements into a search tree and then performing an inorder traversal, we obtain the elements in sorted order. This gives an $O(N \log N)$ algorithm to sort, which is a worst-case bound if any sophisticated search tree is used. We shall see better ways in Chapter 7, but none that have a lower time bound.

Exercises

Questions 4.1 to 4.3 refer to the tree in Figure 4.73.

4.1 For the tree in Figure 4.73:
a. Which node is the root?
b. Which nodes are leaves?

4.2 For each node in the tree of Figure 4.73:
a. Name the parent node.

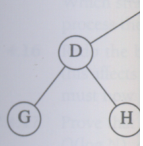

Figure 4.73 Tree

b. List the
c. List the
d. Comput
e. Comput

4.3 What is the

4.4 Show that
children.

4.5 Show that

4.6 A *full node* i
is equal to

4.7 Suppose a b
Prove that

4.8 Give the p
ure 4.74.

4.9 a. Show th
search t
b. Show th

4.10 Let $f(N)$
a. Dete
b. Shov

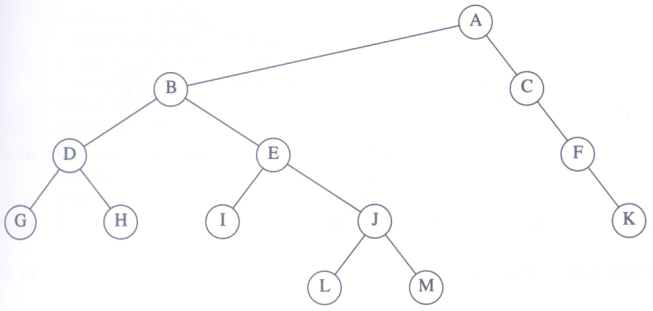

Figure 4.73 Tree for Exercises 4.1 to 4.3

b. List the children.
c. List the siblings.
d. Compute the depth.
e. Compute the height.

4.3 What is the depth of the tree in Figure 4.73?

4.4 Show that in a binary tree of N nodes, there are $N + 1$ `NULL` links representing children.

4.5 Show that the maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$.

4.6 A *full node* is a node with two children. Prove that the number of full nodes plus one is equal to the number of leaves in a nonempty binary tree.

4.7 Suppose a binary tree has leaves $l_1, l_2, \ldots, l_M$ at depths $d_1, d_2, \ldots, d_M$, respectively. Prove that $\sum_{i=1}^{M} 2^{-d_i} \leq 1$ and determine when the equality is true.

4.8 Give the prefix, infix, and postfix expressions corresponding to the tree in Figure 4.74.

4.9 a. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.
b. Show the result of deleting the root.

4.10 Let $f(N)$ be the average number of full nodes in a binary search tree.
a. Determine the values of $f(0)$ and $f(1)$.
b. Show that for $N > 1$

$$f(N) = \frac{N-2}{N} + \frac{1}{N}\sum_{i=0}^{N-1}(f(i) + f(N-i-1))$$

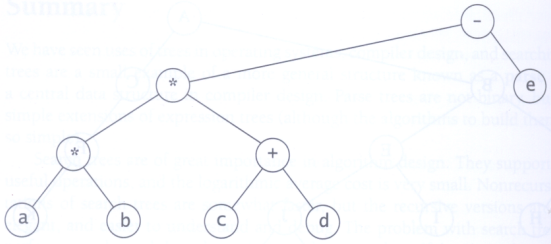

Figure 4.74 Tree for Exercise 4.8

c. Show (by induction) that $f(N) = (N - 2)/3$ is a solution to the equation in part (b), with the initial conditions in part (a).
d. Use the results of Exercise 4.6 to determine the average number of leaves in a binary search tree.

4.11 Write an implementation of the `set` class, with associated iterators using a binary search tree. Add to each node a link to the parent node.

4.12 Write an implementation of the `map` class by storing a data member of type `set<Pair<KeyType,ValueType> >`.

4.13 Write an implementation of the `set` class, with associated iterators using a binary search tree. Add to each node a link to the next smallest and next largest node. To make your code simpler, add a header and tail node which are not part of the binary search tree, but help make the linked list part of the code simpler.

4.14 Suppose you want to perform an experiment to verify the problems that can be caused by random `insert`/`remove` pairs. Here is a strategy that is not perfectly random, but close enough. You build a tree with N elements by inserting N elements chosen at random from the range 1 to $M = \alpha N$. You then perform N^2 pairs of insertions followed by deletions. Assume the existence of a routine, `randomInteger(a,b)`, which returns a uniform random integer between `a` and `b` inclusive.
a. Explain how to generate a random integer between 1 and M that is not already in the tree (so a random insertion can be performed). In terms of N and α, what is the running time of this operation?
b. Explain how to generate a random integer between 1 and M that is already in the tree (so a random deletion can be performed). What is the running time of this operation?
c. What is a good choice of α? Why?

4.15 Write a program to evaluate empirically the following strategies for removing nodes with two children:
a. Replace with the largest node, X, in T_L and recursively remove X.

b. Alternately replace with the largest node in T_L and the smallest node in T_R, and recursively remove the appropriate node.
c. Replace with either the largest node in T_L or the smallest node in T_R (recursively removing the appropriate node), making the choice randomly.

Which strategy seems to give the most balance? Which takes the least CPU time to process the entire sequence?

4.16 Redo the binary search tree class to implement lazy deletion. Note carefully that this affects all of the routines. Especially challenging are `findMin` and `findMax`, which must now be done recursively.

****4.17** Prove that the depth of a random binary search tree (depth of the deepest node) is $O(\log N)$, on average.

4.18 *a. Give a precise expression for the minimum number of nodes in an AVL tree of height h.
b. What is the minimum number of nodes in an AVL tree of height 15?

4.19 Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.

***4.20** Keys $1, 2, \ldots, 2^k - 1$ are inserted in order into an initially empty AVL tree. Prove that the resulting tree is perfectly balanced.

4.21 Write the remaining procedures to implement AVL single and double rotations.

4.22 Design a linear-time algorithm that verifies that the height information in an AVL tree is correctly maintained and that the balance property is in order.

4.23 Write a nonrecursive function to insert into an AVL tree.

***4.24** How can you implement (nonlazy) deletion in AVL trees?

4.25 a. How many bits are required per node to store the height of a node in an N-node AVL tree?
b. What is the smallest AVL tree that overflows an 8-bit height counter?

4.26 Write the functions to perform the double rotation without the inefficiency of doing two single rotations.

4.27 Show the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in Figure 4.75.

4.28 Show the result of deleting the element with key 6 in the resulting splay tree for the previous exercise.

4.29 a. Show that if all nodes in a splay tree are accessed in sequential order, the resulting tree consists of a chain of left children.
****b. Show that if all nodes in a splay tree are accessed in sequential order, then the total access time is $O(N)$, regardless of the initial tree.

4.30 Write a program to perform random operations on splay trees. Count the total number of rotations performed over the sequence. How does the running time compare to AVL trees and unbalanced binary search trees?

4.31 Write efficient functions that take only a pointer to the root of a binary tree, T, and compute:
a. The number of nodes in T.
b. The number of leaves in T.