11.5 (2,4) Trees

In this section, we consider a data structure known as a (2,4) tree. It is a particular example of a more general structure known as a multiway search tree, in which internal nodes may have more than two children. Other forms of multiway search trees will be discussed in Section 15.3.

11.5.1 Multiway Search Trees

Recall that general trees are de·ned so that internal nodes may have many children. In this section, we discuss how general trees can be used as multiway search trees. Map items stored in a search tree are pairs of the form (k,v), where k is the key and v is the value associated with the key.

De·nition of a Multiway Search Tree

Let w be a node of an ordered tree. We say that w is a d-node if w has d children. We de·ne a multiway search tree to be an ordered tree T that has the following properties, which are illustrated in Figure 11.23a:

· Each internal node of T has at least two children. That is, each internal node is a d-node such that d ·2.

· Each internal d-node w of T with children c1,...,cd stores an ordered set of d ·1 key-value pairs (k1,v1),..., (kd·1,vd·1), where k1 ···· ·kd·1.

· Let us conventionally de·ne k0 = ··and kd = +·. For each item (k,v) stored at a node in the subtree of w rooted at ci, i = 1,...,d, we have that ki·1 ·k ·ki.

That is, if we think of the set of keys stored at w as including the special ·ctitious keys k0 = ··and kd = +·, then a key k stored in the subtree of T rooted at a child node ci must be ·in between· two keys stored at w. This simple viewpoint gives rise to the rule that a d-node stores d ·1 regular keys, and it also forms the basis of the algorithm for searching in a multiway search tree.

By the above de·nition, the external nodes of a multiway search do not store any data and serve only as ·placeholders.· These external nodes can be ef·ciently represented by None references, as has been our convention with binary search trees (Section 11.1). However, for the sake of exposition, we will discuss these as actual nodes that do not store anything. Based on this de·nition, there is an interesting relationship between the number of key-value pairs and the number of external nodes in a multiway search tree.

Proposition 11.7: An n-item multiway search tree has n+1 external nodes.

We leave the justi·cation of this proposition as an exercise (C-11.52).

5.2.2
Compact Arrays in Python
In the introduction to this section, we emphasized that strings are represented using an array of characters (not an array of references). We will refer to this more direct representation as a compact array because the array is storing the bits that represent the primary data (characters, in the case of strings).

Compact arrays have several advantages over referential structures in terms of computing performance. Most signi·cantly, the overall memory usage will be much lower for a compact structure because there is no overhead devoted to the explicit storage of the sequence of memory references (in addition to the primary data). That is, a referential structure will typically use 64-bits for the memory address stored in the array, on top of whatever number of bits are used to represent the object that is considered the element. Also, each Unicode character stored in a compact array within a string typically requires 2 bytes. If each character were stored independently as a one-character string, there would be signi·cantly more bytes used.
As another case study, suppose we wish to store a sequence of one million, 64-bit integers. In theory, we might hope to use only 64 million bits. However, we estimate that a Python list will use four to ·ve times as much memory. Each element of the list will result in a 64-bit memory address being stored in the primary array, and an int instance being stored elsewhere in memory. Python allows you to query the actual number of bytes being used for the primary storage of any object. This is done using the getsizeof function of the sys module. On our system, the size of a typical int object requires 14 bytes of memory (well beyond the 4 bytes needed for representing the actual 64-bit number). In all, the list will be using 18 bytes per entry, rather than the 4 bytes that a compact list of integers would require.
Another important advantage to a compact structure for high-performance computing is that the primary data are stored consecutively in memory. Note well that

9.3 Heaps

The two strategies for implementing a priority queue ADT in the previous section demonstrate an interesting trade-off. When using an unsorted list to store entries, we can perform insertions in O(1) time, but ·nding or removing an element with minimum key requires an O(n)-time loop through the entire collection. In contrast, if using a sorted list, we can trivially ·nd or remove the minimum element in O(1) time, but adding a new element to the queue may require O(n) time to restore the sorted order.

In this section, we provide a more ef·cient realization of a priority queue using a data structure called a binary heap. This data structure allows us to perform both insertions and removals in logarithmic time, which is a signi·cant improvement over the list-based implementations discussed in Section 9.2. The fundamental way the heap achieves this improvement is to use the structure of a binary tree to ·nd a compromise between elements being entirely unsorted and perfectly sorted.

9.3.1 The Heap Data Structure

A heap (see Figure 9.1) is a binary tree T that stores a collection of items at its positions and that satis·es two additional properties: a relational property de·ned in terms of the way keys are stored in T and a structural property de·ned in terms of the shape of T itself. The relational property is the following:

Heap-Order Property: In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p·s parent.

As a consequence of the heap-order property, the keys encountered on a path from the root to a leaf of T are in nondecreasing order. Also, a minimum key is always stored at the root of T. This makes it easy to locate such an item when min or remove min is called, as it is informally said to be ·at the top of the heap· (hence, the name ·heap· for the data structure). By the way, the heap data structure de·ned here has nothing to do with the memory heap (Section 15.1.1) used in the run-time environment supporting a programming language like Python.

For the sake of ef·ciency, as will become clear later, we want the heap T to have as small a height as possible. We enforce this requirement by insisting that the heap T satisfy an additional structural property·it must be what we term complete.

Complete Binary Tree Property: A heap T with height h is a complete binary tree if levels 0,1,2,...,h·1 of T have the maximum number of nodes possible (namely, level i has 2i nodes, for 0 ·i ·h·1) and the remaining nodes at level h reside in the leftmost possible positions at that level.

Proposition 5.1: Let S be a sequence implemented by means of a dynamic array with initial capacity one, using the strategy of doubling the array size when full. The total time to perform a series of n append operations in S, starting from S being empty, is O(n).

Justi·cation:

Let us assume that one cyber-dollar is enough to pay for the execution of each append operation in S, excluding the time spent for growing the array. Also, let us assume that growing the array from size k to size 2k requires k cyber-dollars for the time spent initializing the new array. We shall charge each append operation three cyber-dollars. Thus, we overcharge each append operation that does not cause an over·ow by two cyber-dollars. Think of the two cyber-dollars pro·ted in an insertion that does not grow the array as being ·stored· with the cell in which the element was inserted. An over·ow occurs when the array S has 2i elements, for some integer i ·0, and the size of the array used by the array representing S is 2i. Thus, doubling the size of the array will require 2i cyber-dollars. Fortunately, these cyber-dollars can be found stored in cells 2i·1 through 2i ·1. (See Figure 5.14.) Note that the previous over·ow occurred when the number of elements became larger than 2i·1 for the ·rst time, and thus the cyber-dollars stored in cells 2i·1 through 2i ·1 have not yet been spent. Therefore, we have a valid amortization scheme in which each operation is charged three cyber-dollars and all the computing time is paid for. That is, we can pay for the execution of n append operations using 3n cyber-dollars. In other words, the amortized running time of each append operation is O(1); hence, the total running time of n append operations is O(n).

(a)

0
2
4
5
6
7
3
1
$
$
$
$
$
$
$

5.2
Low-Level Arrays
To accurately describe the way in which Python represents the sequence types, we must ·rst discuss aspects of the low-level computer architecture. The primary memory of a computer is composed of bits of information, and those bits are typically grouped into larger units that depend upon the precise system architecture. Such a typical unit is a byte, which is equivalent to 8 bits.
A computer system will have a huge number of bytes of memory, and to keep track of what information is stored in what byte, the computer uses an abstraction known as a memory address. In effect, each byte of memory is associated with a unique number that serves as its address (more formally, the binary representation of the number serves as the address). In this way, the computer system can refer to the data in ·byte #2150· versus the data in ·byte #2157,· for example. Memory addresses are typically coordinated with the physical layout of the memory system, and so we often portray the numbers in sequential fashion. Figure 5.1 provides such a diagram, with the designated memory address for each byte.
2160
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2144
2159
Figure 5.1: A representation of a portion of a computer·s memory, with individual bytes labeled with consecutive memory addresses.
Despite the sequential nature of the numbering system, computer hardware is designed, in theory, so that any byte of the main memory can be ef·ciently accessed based upon its memory address. In this sense, we say that a computer·s main memory performs as random access memory (RAM). That is, it is just as easy to retrieve

5.6 Multidimensional Data Sets

Lists, tuples, and strings in Python are one-dimensional. We use a single index to access each element of the sequence. Many computer applications involve multidimensional data sets. For example, computer graphics are often modeled in either two or three dimensions. Geographic information may be naturally represented in two dimensions, medical imaging may provide three-dimensional scans of a patient, and a company·s valuation is often based upon a high number of independent ·nancial measures that can be modeled as multidimensional data. A two-dimensional array is sometimes also called a matrix. We may use two indices, say i and j, to refer to the cells in the matrix. The ·rst index usually refers to a row number and the second to a column number, and these are traditionally zero-indexed in computer science. Figure 5.22 illustrates a two-dimensional data set with integer values. This data might, for example, represent the number of stores in various regions of Manhattan.

22
18
709
5
33
10
4
56
82
440
45
32
830
120
750
660
13
77
20
105
4
880
45
66

Binary Search Trees
An important application of the inorder traversal algorithm arises when we store an ordered sequence of elements in a binary tree, de·ning a structure we call a binary search tree. Let S be a set whose unique elements have an order relation. For example, S could be a set of integers. A binary search tree for S is a binary tree T such that, for each position p of T:
· Position p stores an element of S, denoted as e(p).
· Elements stored in the left subtree of p (if any) are less than e(p).
· Elements stored in the right subtree of p (if any) are greater than e(p).
An example of a binary search tree is shown in Figure 8.19. The above properties assure that an inorder traversal of a binary search tree T visits the elements in nondecreasing order.
36
25
31
42
12
62
75
58
90
Figure 8.19: A binary search tree storing integers. The solid path is traversed when searching (successfully) for 36. The dashed path is traversed when searching (unsuccessfully) for 70.
We can use a binary search tree T for set S to ·nd whether a given search value v is in S, by traversing a path down the tree T, starting at the root. At each internal position p encountered, we compare our search value v with the element e(p) stored at p. If v < e(p), then the search continues in the left subtree of p. If v = e(p), then the search terminates successfully. If v > e(p), then the search continues in the right subtree of p. Finally, if we reach an empty subtree, the search terminates unsuccessfully. In other words, a binary search tree can be viewed as a binary decision tree (recall Example 8.6), where the question asked at each internal node is whether the element at that node is less than, equal to, or larger than the element being searched for. We illustrate several examples of the search operation in Figure 8.19.
Note that the running time of searching in a binary search tree T is proportional to the height of T. Recall from Proposition 8.8 that the height of a binary tree with n nodes can be as small as log(n+1)·1 or as large as n·1. Thus, binary search trees are most ef·cient when they have small height. Chapter 11 is devoted to the

Removing the Item with Minimum Key
Let us now turn to method remove min of the priority queue ADT. We know that an entry with the smallest key is stored at the root r of T (even if there is more than one entry with smallest key). However, in general we cannot simply delete node r, because this would leave two disconnected subtrees.
Instead, we ensure that the shape of the heap respects the complete binary tree property by deleting the leaf at the last position p of T, de·ned as the rightmost position at the bottommost level of the tree. To preserve the item from the last position p, we copy it to the root r (in place of the item with minimum key that is being removed by the operation). Figure 9.3a and b illustrates an example of these steps, with minimal item (4,C) being removed from the root and replaced by item (13,W) from the last position. The node at the last position is removed from the tree.
Down-Heap Bubbling After a Removal
We are not yet done, however, for even though T is now complete, it likely violates the heap-order property. If T has only one node (the root), then the heap-order property is trivially satis·ed and the algorithm terminates. Otherwise, we distinguish two cases, where p initially denotes the root of T:
· If p has no right child, let c be the left child of p.
· Otherwise (p has both children), let c be a child of p with minimal key.
If key kp ·kc, the heap-order property is satis·ed and the algorithm terminates. If instead kp > kc, then we need to restore the heap-order property. This can be locally achieved by swapping the entries stored at p and c. (See Figure 9.3c and d.) It is worth noting that when p has two children, we intentionally consider the smaller key of the two children. Not only is the key of c smaller than that of p, it is at least as small as the key at c·s sibling. This ensures that the heap-order property is locally restored when that smaller key is promoted above the key that had been at p and that at c·s sibling.
Having restored the heap-order property for node p relative to its children, there may be a violation of this property at c; hence, we may have to continue swapping down T until no violation of the heap-order property occurs. (See Figure 9.3e·h.) This downward swapping process is called down-heap bubbling. A swap either resolves the violation of the heap-order property or propagates it one level down in the heap. In the worst case, an entry moves all the way down to the bottom level. (See Figure 9.3.) Thus, the number of swaps performed in the execution of method remove min is, in the worst case, equal to the height of heap T, that is, it is ·logn· by Proposition 9.2.

Searching in a Multiway Tree

Searching for an item with key k in a multiway search tree T is simple. We perform such a search by tracing a path in T starting at the root. (See Figure 11.23b and c.) When we are at a d-node w during this search, we compare the key k with the keys k1,...,kd·1 stored at w. If k = ki for some i, the search is successfully completed. Otherwise, we continue the search in the child ci of w such that ki·1 < k < ki. (Recall that we conventionally de·ne k0 = ··and kd = +·.) If we reach an external node, then we know that there is no item with key k in T, and the search terminates unsuccessfully.

Data Structures for Representing Multiway Search Trees

In Section 8.3.3, we discuss a linked data structure for representing a general tree. This representation can also be used for a multiway search tree. When using a general tree to implement a multiway search tree, we must store at each node one or more key-value pairs associated with that node. That is, we need to store with w a reference to some collection that stores the items for w.

During a search for key k in a multiway search tree, the primary operation needed when navigating a node is ·nding the smallest key at that node that is greater than or equal to k. For this reason, it is natural to model the information at a node itself as a sorted map, allowing use of the ·nd ge(k) method. We say such a map serves as a secondary data structure to support the primary data structure represented by the entire multiway search tree. This reasoning may at ·rst seem like a circular argument, since we need a representation of a (secondary) ordered map to represent a (primary) ordered map. We can avoid any circular dependence, however, by using the bootstrapping technique, where we use a simple solution to a problem to create a new, more advanced solution.

In the context of a multiway search tree, a natural choice for the secondary structure at each node is the SortedTableMap of Section 10.3.1. Because we want to determine the associated value in case of a match for key k, and otherwise the corresponding child ci such that ki·1 < k < ki, we recommend having each key ki in the secondary structure map to the pair (vi,ci). With such a realization of a multiway search tree T, processing a d-node w while searching for an item of T with key k can be performed using a binary search operation in O(logd) time. Let dmax denote the maximum number of children of any node of T, and let h denote the height of T. The search time in a multiway search tree is therefore O(hlogdmax). If dmax is a constant, the running time for performing a search is O(h).

The primary ef·ciency goal for a multiway search tree is to keep the height as small as possible. We next discuss a strategy that caps dmax at 4 while guaranteeing a height h that is logarithmic in n, the total number of items stored in the map.

P-7.45 An array A is sparse if most of its entries are empty (i.e., None). A list L can be used to implement such an array ef·ciently. In particular, for each nonempty cell A[i], we can store an entry (i,e) in L, where e is the element stored at A[i]. This approach allows us to represent A using O(m) storage, where m is the number of nonempty entries in A. Provide such a SparseArray class that minimally supports methods
getitem
(j) and
setitem
(j, e) to provide standard indexing operations. Analyze the ef·ciency of these methods.
P-7.46 Although we have used a doubly linked list to implement the positional list ADT, it is possible to support the ADT with an array-based implementation. The key is to use the composition pattern and store a sequence of position items, where each item stores an element as well as that element·s current index in the array. Whenever an element·s place in the array is changed, the recorded index in the position must be updated to match. Given a complete class providing such an array-based implementation of the positional list ADT. What is the ef·ciency of the various operations?
P-7.47 Implement a CardHand class that supports a person arranging a group of cards in his or her hand. The simulator should represent the sequence of cards using a single positional list ADT so that cards of the same suit are kept together. Implement this strategy by means of four ··ngers· into the hand, one for each of the suits of hearts, clubs, spades, and diamonds, so that adding a new card to the person·s hand or playing a correct card from the hand can be done in constant time. The class should support the following methods:
· add card(r, s): Add a new card with rank r and suit s to the hand.
· play(s): Remove and return a card of suit s from the player·s hand; if there is no card of suit s, then remove and return an arbitrary card from the hand.
.
iter
(): Iterate through all cards currently in the hand.
· all of suit(s): Iterate through all cards of suit s that are currently in the hand.
Chapter Notes
A view of data structures as collections (and other principles of object-oriented design) can be found in object-oriented design books by Booch [17], Budd [20], Goldberg and