To implement the PUSH operation, we increment the stack pointer, read the ap- propriate page into memory from disk, copy the element to be pushed to the ap- propriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.

Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and ,.m/ CPU time.

> ***a.*** Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)
>
> Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.
>
> ***b.*** What is the worst-case number of disk accesses required for n PUSH opera- tions? What is the CPU time?
>
> ***c.*** What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?
>
> Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).
>
> ***d.*** Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

Consider the following operation on an arbitrary set *Y:*

Next(A, x): find the two elements x_1 and x_2 such that x_1 is the largest element in the set, strictly smaller than x, while x_2 is the smallest element in the set, strictly smaller than x.

Develop an $O(\log n)$-time algorithm, assuming that the set is stored in a B-tree.

For the ***Y*** where x_1 is the largest element in set Y and is **strictly smaller** than x and $\boldsymbol{x_2}$ is the smallest element that is strictly larger than x.

1. Identify leaf positions corresponding to Y where elements x_1 be inserting
2. If there is a suitable value for y_1 and y_2 where $y_1 < x < y_2$ then return y_1 and y_2
3. If not, recurse through to find such values
    a. Y_1

i. Find the parent node than has a child that is larger than *S*, check the value nearest and then for this parent, find the smallest element in the sub-tree

b. Y_2

i. Find the parent node that has a smaller child and then for that parent node find the largest element in that specific sub-tree

The time complexity of this algorithm is O(log *n*) clearly by the performance of (1) and (3) by recursing through a binary tree. The worst case is log *n* therefore the algorithm's complexity is O(log *n*) completely.

18-2 Joining and splitting 2-3-4 trees

The join operation takes two dynamic sets S0 and S00 and an element x such that for any x0 2 S0 and x00 2 S00, we have x0:key < x:key < x00:key. It returns a set S D S0 [fxg [S00. The split operation is like an "inverse" join: given a dynamic set S and an element x 2 S, it creates a set S0 that consists of all elements in S fxg whose keys are less than x:key and a set S00 that consists of all elements in S fxg whose keys are greater than x:key. In this problem, we investigate

Exercises

18.3-1

Show the results of deleting C , P , and V , in order, from the tree of Figure 18.8(f).

18.3-2

Write pseudocode for B-TREE-DELETE.

18.4

Given a B-tree, *T*, of *n* leaves, and an integer *x* such that log $n \leq x \leq n$, find the *x* smallest root-to-leaf outcomes in the tree *T*. Develop an O(*x*)-time algorithm for the problem. HINT: Recurse over the

As described in pages 191-193, here we assume that there is an order defined on the elements. Thus, any two elements x and y can be compared using one of the relations $x<y, y<x, x\leq y, y\leq x, and x=y$. Asearchingalgorithmbasedoncomparisonsuses only comparisons to gain order information of the input. Therefore, no matter how the set S is organized, a searching algorithm A based on comparisons can always be depicted as a decision-tree T . The decision-tree T is a binary tree in which each internal node corresponds to a comparison and its two children correspond to the two outcomes of the comparison, and each leaf corresponds to a conclusion of the search result (i.e., "Yes, x is in S", or "No, x is not in S"). In

this model, for each given input (S, x), a comparison (i.e., an internal node in T) will have a unique outcome, so the algorithm on this input will follow a particular path from the root to a leaf in T , in which the leaf gives the conclusion of the algorithm on the input.
Now fix a set S of n elements, S = {1,2,...,n}, and consider any searching algorithm A on the input (S,x), where x may vary. Thus, if x is an integer i between 1 and n, then the algorithm A will return Yes. Otherwise, the algorithm returns No.
We first show that the decision-tree T corresponding to the algorithm A contains at least n Yes-leaves. For this, it suffices to show that for two integers i and j between 1 and n, i < j, the two root-leaf paths in T corresponding to the inputs (S, i) and (S, j), respectively, are different. Assume the contrary that the inputs (S,i) and (S,j) correspond to the same root-leaf path P in T . Then consider the input (S, i + 0.5).

Give a formal proof that a 2-3 tree with n leaves has height bounded by log2 n. The height of a 2-3 tree is defined to be the length of the path from its root to any of its leaves.

> *Proof*. We prove this by induction on the number n of leaves in the 2-3 tree. If n = 1, then the 2-3 tree is a single node tree, which has a height 0 = log2 1, so the conclusion holds true.
>
> Now consider a 2-3 tree T with n > 1 leaves. The root r of T has at least two children r1 and r2 (and possible a third child r3), which are roots of two 2-3 trees T1 and T2, respectively. Since T has n leaves, at least one of T1 and T2 has no more than n/2 leaves. Without loss of generality, suppose that T1 has n1 ≤ n/2 leaves. Since n1 < n, we can apply the induction on T1. Thus, the height of T1 is bounded by log2 n1 ≤ log2(n/2) = log2 n − 1. Since all paths from the root to leaves in a 2-3 tree have the same length, the height of T must be equal to the height of T1 plus 1. Therefore, we have
>
> height(T) = height(T1) + 1 ≤ (log2 n − 1) + 1 = log2 n.