6.897: Advanced Data Structures Spring 2003

Lecture 16 — April 23, 2003

Prof. Erik Demaine *Scribes: Sonia Garg, Hana Kim*

1 Overview

In the last two lectures, we discussed ordered-file maintenance and the cache-oblivious model. The ordered-file maintenance structure maintains an ordered array of elements with evenly distributed gaps and intervals. In the cache-oblivious model, element accesses trigger automatic block transfers with an optimal block-replacement strategy.

In this lecture we talk about a major technique in cache-oblivious design, *divide and conquer*. The divide-and-conquer approach divides a problem into subproblems and recursively solves each subproblem. We will see how using divide and conquer can improve the running time and number of memory accesses for various search algorithms.

2 Divide and Conquer

The divide-and-conquer technique is a general principle used to simplify complex algorithms. When examining the algorithm, we can recursively split the problem into subproblems all the way down to O(1) time. In analysis, we can stop short and consider the point when the problem is less than or equal to the size of the memory ($\leq M$) or fits in a constant number of blocks ($O(B)$).

2.1 Simple Example

To illustrate the divide-and-conquer technique, we can look at the problem of finding the median in linear time.

Problem: Given an array of n elements, not necessarily in sorted order, find the kth smallest element. If $k = n/2$, then the kth element is the *median*.

Solution: Linear-time worst-case selection algorithm [BFPRT73]

1. Conceptually partition the array into $N/5$ 5-tuples.

   This operation is purely conceptual, so takes 0 time.

2. Compute median of each 5-tuple.

   This operation involves writing the output to a second array as we scan the tuples in parallel—two parallel scans. Thus, we use $O(N)$ time and $O(\lceil N/B \rceil)$ memory transfers, assuming that $M \geq 2B$.

3. Recursively compute median, m, of these medians.

   This operation involves a recursive call on a problem of size $N/5$.

4. Partition the array into the elements $\leq m$ and the elements $> m$.

   This operation involves three parallel scans (with the obvious implementation), so it uses $O(N)$ time and $O(\lceil N/B \rceil)$ memory transfers, assuming that $M \geq 3B$.

5. Count the lengths of these two subarrays and recurse into the appropriate half.

Recurrence for running time (see e.g. CLRS):

$$T(N) = T(\tfrac{1}{5}N) + T(\tfrac{7}{10}N) + O(N).$$

Recurrence for number of memory transfers:

$$T(N) = T(\tfrac{1}{5}N) + T(\tfrac{7}{10}N) + O(\lceil N/B \rceil)$$

Question: What's the base case?

First try: $T(O(1)) = O(1)$.

Let L be the number of leaves in the recursion tree. We guess that $L(N) = N^c$ for some c:

$$\begin{array}{rcl} L(N) & = & L(\tfrac{1}{5}N) + L(\tfrac{7}{10}N) \\ N^c & = & (\tfrac{1}{5}N)^c + (\tfrac{7}{10}N)^c \\ 1 & = & (\tfrac{1}{5})^c + (\tfrac{7}{10})^c \\ c & \approx & 0.8397803\ldots \end{array}$$

Because each leaf costs $\Theta(1)$ by the base case, $T(N) = \Omega(N^{0.83978})$ according to this analysis.

This bound is too weak for our purposes. We want to prove that $T(N) = O(\lceil N/B \rceil)$, but $T(N) = \omega(\lceil N/B \rceil)$ if $N^{0.2} \leq B \leq N$. Our bound needs to work for all values of B.

Second try: $T(O(B)) = O(1)$.

The number of leaves in this case is $(N/B)^c$, with the same value of c as above, because we are stopping at subcases of size B. Thus, the number of memory transfers spent at the leaves of the recursion tree (base cases) is $o(N/B)$. The $\Theta(\lceil N/B \rceil)$ memory transfers at the root of the recursion tree are asymptotically more expensive. Thus, the cost at each level of the recursion tree decreases geometrically as we move down. Therefore, the root dominates and the total number of memory transfers is $\Theta(\lceil N/B \rceil)$ as desired.

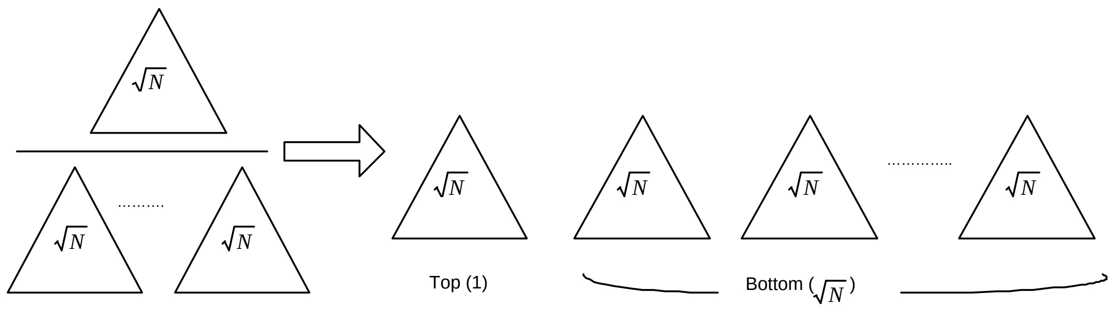
2.2 Binary Search on Static Search Trees

Problem: Improve the cost of search on static search trees using divide-and-conquer.

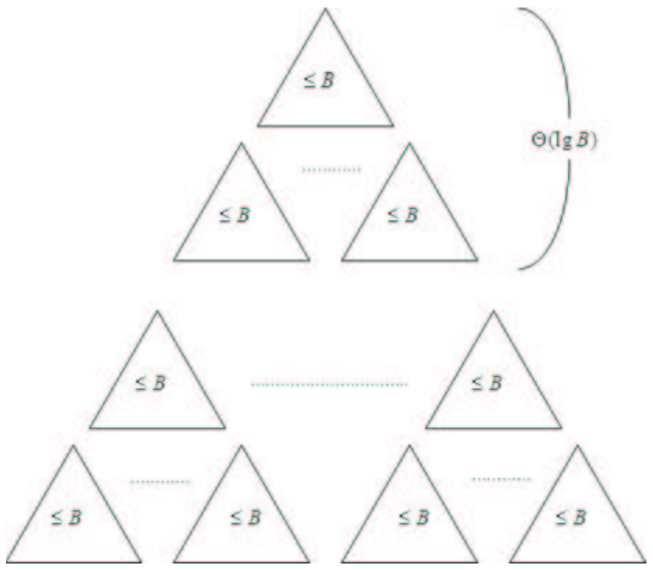
Solutions: A naïve approach would be to use binary search. Binary search takes $O(\lg N)$ time and $O(\lg(N/B))$ memory transfers. Note $O(\lg(N/B) = O(\lg N) - O(\lg B)$. We know that B-trees take $O(\log_B N) = O\left(\frac{\lg N}{\lg B}\right)$ memory transfers. We want to improve our static search trees to be as good as B-trees.

Van Emde Boas layout [P99, BDF00]

1. Arrange the N elements in-order in a complete binary tree on N nodes.
2. Split the tree at middle level.
3. On top, we will have one subtree with $\sqrt{N}$ nodes. On the bottom, we will have $\sqrt{N}$ subtrees each with $\sqrt{N}$ nodes. See figure below.
4. Recursively layout each subtree.
5. Concatenate these layouts together, and store the nodes in that order.



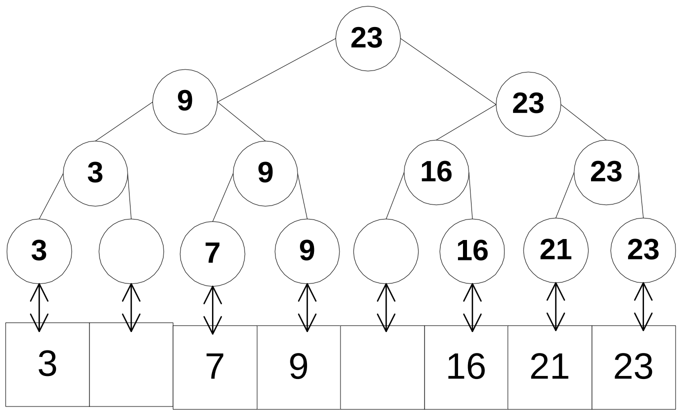
Analysis: We are recursively cutting the height of the search tree in half. We will "stop" the recursion when $\sqrt{B} \leq \sqrt{N}$ (size of subtrees) $\leq B$. Then we can have pieces with height between $\frac{1}{2} \lg B$ and $\lg B$ and containing between $\sqrt{B}$ and B nodes. The number of pieces visited on a root-to-leaf path is $\Theta\left(\frac{\lg N}{\lg B}\right) = \Theta(\log_B N)$. Each piece fits within a block, so occupies at most two blocks (depending on alignment). Therefore, the number of memory transfers in a search is $O(\log_B N)$.



Theorem: Being careful with constants: number of memory transfers in a search is $\leq 4 \log_B N + 2$. This theorem applies to all search trees with constant node degrees ≥ 2.

2.3 Dynamic B-Trees [BDF00, BDIW02, BFJ02]

Now we'll consider applying divide-and-conquer to the dynamic case. We can achieve a dynamic tree by using a static search tree and performing ordered-file maintenance on the leaves [BDIW02]. This organization creates gaps that will allow for the shuffling of elements. Using constant-size gaps, where the pointers between leaf nodes represent slots (see figure below), results in a cost of $O((\lg^2 N)/B)$ amortized memory transfers.



Invariant: Each node in the tree stores the maximum value of its children. Search can then use the value at left child to decide to travel down the left or right path.

2.3.1 Insert

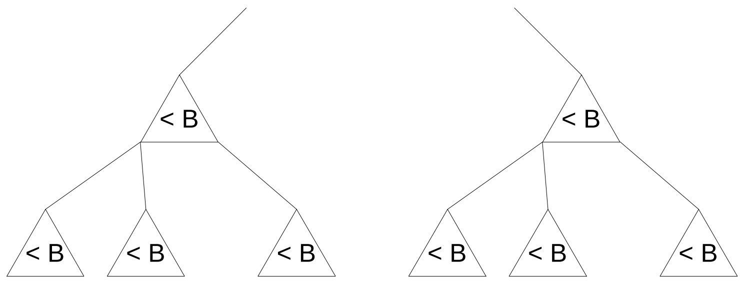
To insert an element in the dynamic B-tree, we proceed as follows:

1. We either search for the appropriate slot or suppose that it is given
2. Insert the element into the ordered-file maintenance structure, which takes $O((\lg^2 N)/B)$ amortized memory transfers.
3. Update the corresponding leaves of changed cells and propagate labels up the tree, in a post-order traversal of changed leaves and their ancestors.

Claim: $O(\lceil K/B \rceil + \lg_B N)$ memory transfers for K changed leaves.

Proof: As before, look at the level of detail where each subtree has between $\leq B$ and $> B$ nodes.

Consider the bottom two levels of subtrees at this level of detail:



Because we look at ≤ 2 triangles at once (a bottom subtree and its parent subtree), we use blocks optimally and update $\Theta(B)$ elements per block read, assuming $M \geq 2B$. Thus, we use $O(\lceil K/B \rceil)$ memory transfers in the bottom two levels.

Now we can afford $O(1)$ transfers for each of the remaining ancestors that we need to update, because there are at most K/B remaining elements above the bottom two levels, until we reduce to a single node to be updated. The path from this single node to the root is $O(\lg_B N)$ extra.

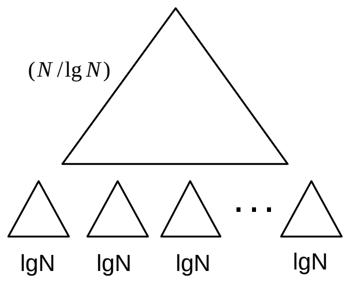
Corollary: $O(\lceil (\lg^2)/B \rceil + \lg_B N)$ amortized memory transfers for an insertion.

To summarize, so far we have obtained the following bounds:

- **Update:** $O(\lg^2 N/B + \lg_B N)$ amortized memory transfers.
- **Search:** $O(\lg_B N)$ memory transfers.
- **Traversal** of K consecutive elements: $O(\lceil K/B \rceil)$ memory transfers.

2.3.2 Indirection

Indirection can delay the number of updates to the ordered-file structure. Specifically, we cluster $\lg N$ consecutive elements:



The top tree of $\Theta(N/\lg N)$ elements is stored using the structure described above. Each bottom tree is stored arbitrarily in a contiguous region of memory of size $\Theta(\lg N)$. Touching a bottom tree (for update or query) involves, at worst, scanning the entire region of memory, for a cost of $O((\lg N)/B)$ memory transfers. Only when one of these bottom trees fills up or becomes too empty do we have to update the top tree. Thus, we effectively slow down the frequency of updates to the top tree by a factor of $\Theta(\lg N)$. More precisely:

Cost of updates:

- Top cost: $O(\frac{\lg N}{B} + \frac{1}{\lg B})$ amortized
- Bottom cost: $O(\frac{\lg N}{B})$
- Total cost: $O(\frac{\lg N}{B})$ amortized + search cost

Cost of search:

- Top cost (same as plain B-tree): $O(\lg_B N)$
- Bottom cost: $O(\frac{\lg N}{B})$
- Total cost: $O(\lg_B N)$

References

[BFPRT73] M. Blum, R.W. Floyd, V. Pratt, R. Rivest, and R. Tarjan, Time bounds for selection, J. Cornput. ystem Sci. (1973).

[P99] Prokop, Cache-Oblivious Algorithms, MIT M.S. Thesis (1999).

[BDF00] M. Bender, E. Demaine, M. Farach-Colton, Cache-Oblivious B-Trees, FOCS 399-409 (2000).

[BDIW02] Michael A. Bender, Z. Duan, J. Iacono, J. Wu, A locality-preserving cache-oblivious dynamic dictionary. SODA 29-38 (2002).

[BFJ02] G. Brodal, R. Fagerberg, R. Jacob, Cache oblivious search trees via binary trees of small height, In Proc. ACM-SIAM Symp. on Discrete Algorithms, 39-48 (2002).