COMS 4995-2: Advanced Algorithms (Spring'20) Apr 14, 2020

Lecture 21: Large Scale Models

Instructor: *Alex Andoni* Scribes: *Yotam Alexander*

1 Recap: I/O model (External Memory Model)

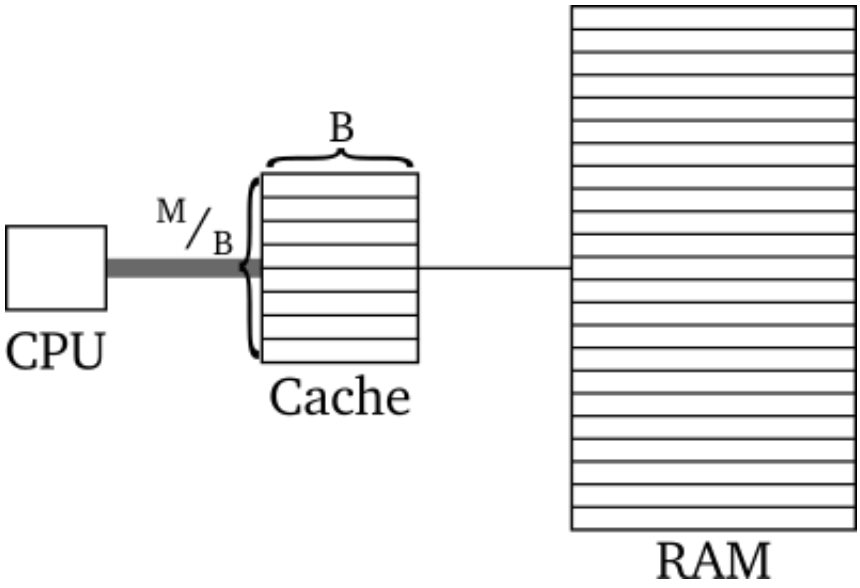

Figure 1: I/O Model

We briefly recall the I/O model:

- We have a cache of size M, and an external memory unit which has unbounded storage space.
- Both are composed of lines of length B.
- To access a location in external memory, we need to import its entire line to the cache.
- We define "Time"/"cost" of an algorithm to be the total number of lines brought into the cache.

Problem 0: Searching for x in an (unsorted) array A of size N via linear scan. This can be done in "time" $O(\frac{N}{B}+1)$, which is unsurprising because an array of size N occupies $\approx \frac{N}{B}$ cache lines.

Problem 1: Searching for x in a sorted array of size N via binary search. We know that in the standard (RAM) model, this can be done in $O(\log(N))$. In the I/O model the analysis proceeds as follows:

1. if $N \leq B$ then we need at most 2 CLs.

2. if $N >> B$: At the beginning of each iteration we search in the range $[l, r] \subset \{1, 2, ..., N\}$. As long as $r - l > B$, we need to bring another CL for each iteration/halving of the range. When $r - l \leq B$, the previous case applies.

Thus the overall cost is $O(\log(\frac{N}{B}) + 1)$.

Note that compared to the standard model, we achieved only an additive improvement ($O(\log(N) - \log(B) + 1)$ vs $O(\log(N))$, whereas in the unsorted search we achieved a multiplicative speedup ($O(\frac{N}{B})$ vs $O(N)$). Thus we would like to design an algorithm that achieves an analogous speedup for the sorted array problem as well.

Problem 2: Searching for x in a Binary Search Tree:

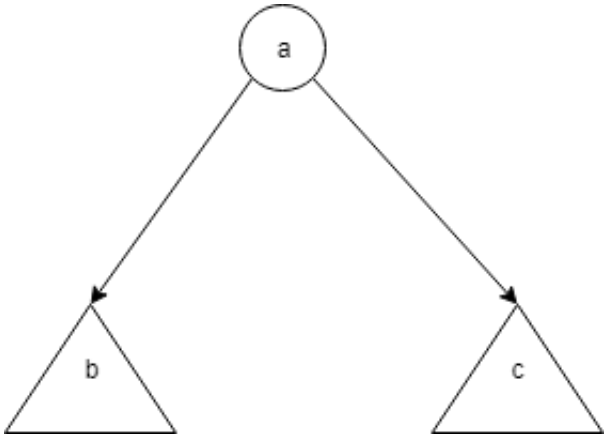

Figure 2: BST, with each node storing an element of A

Where a is the current node, and the binary tree is ordered in the sense that $x \leq a \leq y$ for any x, y in the subtrees b, c respectively.

Analysis:

- if $N \leq 2B$ then we need at most 3 CLs.
- if $N > 2B$: As long as the subtree rooted at the current node is of size $\geq 2B$, we need a new CL for each "split".

Thus overall again we require $O(log(\frac{N}{B}) + 1)$, which isn't an improvement...

Problem 3:

Idea: To get a multiplicative speedup, we re-organize the tree to utilize the CLs more efficiently.

B-Tree

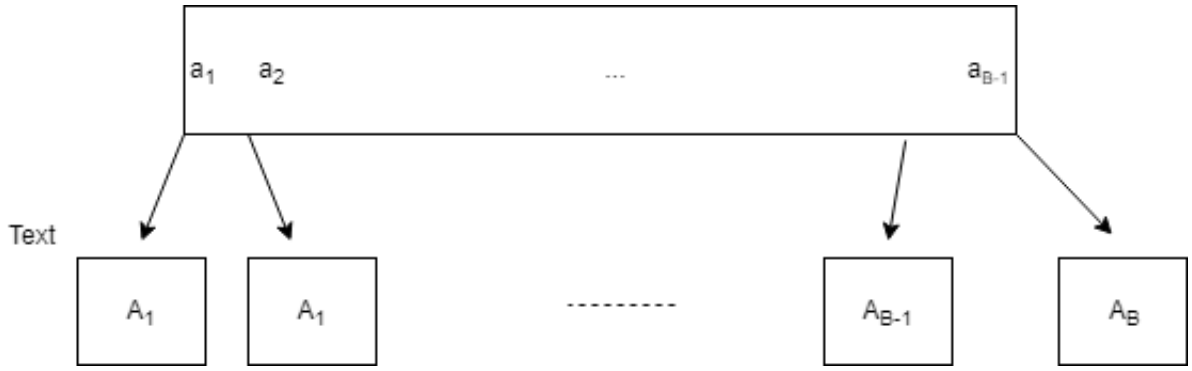

Figure 3: B-Tree

Each nodes now stores $B-1$ elements of A, and has B children, and the tree satisfies the sortedness property:

$$A_1 \leq a_1 \leq A_2 \leq a_2 \leq ... \leq a_{B-1} \leq A_B$$

Note the if B=1 we recover the definition of a BST.

The search proceeds as before.

Analysis: Each node requires at most 2 CLs (we store $B-1$ numbers and B pointers to its children). This implies that the number of CLs we need to bring is $\leq 2\times$ (height of the tree). The height of the tree is $\lceil \log_B(N) \rceil$, so overall we require $O(\log_B(N+1) = O(\frac{Log(N)}{Log(B)} + 1)$ (where as before the constant is required to deal with the case that $N \leq B$).

Example: If $N = 2^{20}$ and $B = 2^{10}$, using a binary tree would require $20 - 10 = 10$ CLs, whereas a B tree would require only $\frac{20}{10} = 2$.

Remark 1: The runtime (number of CPU operations), once we take into account operations within each CL, is $\log_B(N) \times B$ (using a naive scan of the elements in each node), but using Binary Search instead reduces this to $\log_B(N) \times \log(B) = \log(N)$).

Remark 2: Sorting can be implemented at cost $O(\frac{N}{B} \times \log_{\frac{M}{B}}(\frac{N}{B}))$, using an appraoch analogous to a $\frac{M}{B}$ -way merge-sort.

2 Cache Oblivious Model

In the I/O model, using the above algorithms requires knowledge of B,M. This is problematic for the following reasons:

1. We would like to design algorithms that are hardware independent.
2. The value of of the parameters B, M could change over time, e.g because multiple processors use the same cache.

3. In realistic applications there are multiple caches with different parameters.

Thus we would like to design efficient algorithms that don't require knowledge of B,M.

Theorem 1. *We can achieve the same performance, i.e* $O(\frac{\log(N)}{\log(B)})$*, in the cache oblivious model.*

First Approach- Heaps: A heap is a (balanced) BST stored in an array, where if a node is stored in the i-th cell, its children are stored in the cells $2i+1$ and $2i+2$. This approach doesn't work however, because searching for x in this heap requires:

- A single CL for the first B elements, which correspond to depth $Log(B)$ from the root.
- Another CL for each additional "layer", of which $\log(N) - \log(B)$ remain.

And thus $\log(N) - \log(B) + 1$ CLs overall.

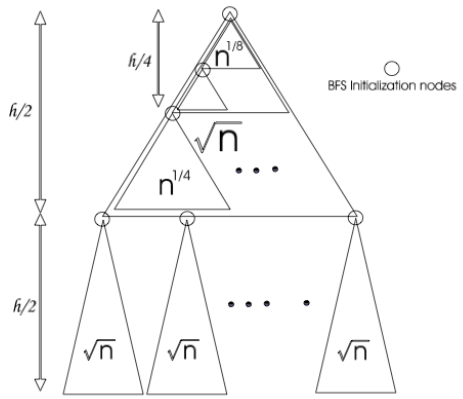

Figure 4: VEB Layout

New Data Structure: We store the BST using the Van Emde Boas layout. To store a tree T with N nodes, we proceed as follows:

- If $N \leq 16$ we store T in an arbitrary way.
- Otherwise, we store $T_0, T_1, ..., T_{\sqrt{N}}$ consecutively, where T_0 consists of the upper half of T (namely all nodes of distance $\leq \frac{n}{2}$ from the root), and $T_0, T_1, ..., T_{\sqrt{N}}$ are the subtrees rooted at the leaves of T_0. Each one of the trees $T_0, T_1, ..., T_{\sqrt{N}}$ is stored recursively.

We will prove that using binary search on a BST stored in this way costs $O(\frac{\log(N)}{\log(B)})$ in the next lecture.