# Searching Versus Sorting—Exploring Computer Architecture and Alternative Models of Computation

Andrew Krapivin

March 4, 2024

## 1 Motivating Example—Sorting vs Searching

For those of you that have taken computer architecture, some of the following may seem "obvious," but I do not think computer architecture stresses enough the impact of the factors I will talk about, so I think this can still be useful. Additionally, after the exposition, assuming we have time, we will cover new algorithms that you most likely have not seen.

I cannot really remember exactly what I was thinking when I was first learning about data structures, but I remember that I had a thought/question along the lines of:

> Sorting with a search tree takes time $O(n \log n)$, and sorting with dedicated sorting algorithms also takes time $O(n \log n)$. Search trees are certainly "more powerful—"they maintain a sorted list dynamically. Since the runtimes are the same, what is the point of sorting algorithms; why not just exclusively use search trees, and in the context of sorting just mention one achieves that optimally?

What are you answers to this question? It's pretty open ended, and there are many directions you can take this.

Here are some of my answers to that question:

1. Sorting algorithms are "easier" to teach. Certainly, talking about rotations in a RB-tree or even just the basic concept of binary search trees or things like that is somewhat more complicated than the comparatively simpler idea of divide and conquer behind quicksort and mergesort.

2. Constant factors—This is likely even related to the previous point. The complexity of search trees and balancing them certainly seems like it would add some more instructions to the program. Thus, probably the hidden constant factors are better for sorting?

3. But are there other reasons? Are there things "more fundamental" in an algorithmic sense?

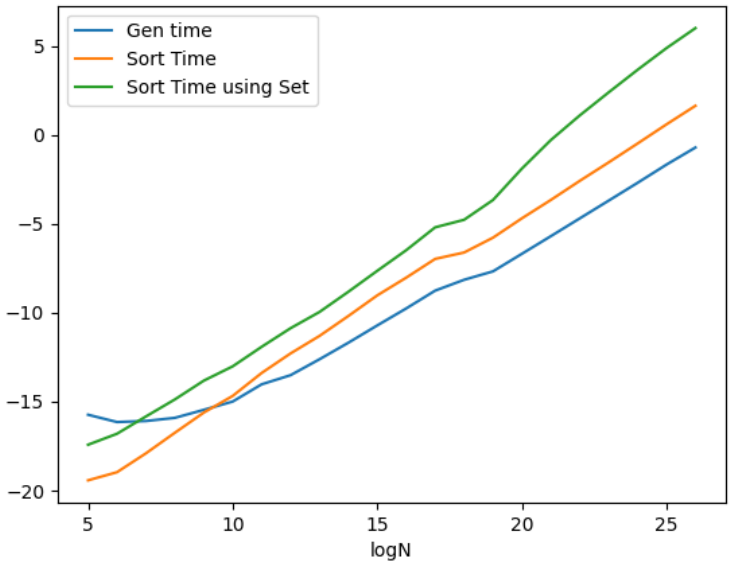
### 1.1 Evaluating Performance

With such a leading question, I am clearly egging on that it is not just "constant" factors at play, but let's actually take a look at how search trees and sorting compare in performance at different sizes of $n$. It certainly seems like a constant factor, so we should just see "relatively" constant ratio between the time to sort for a binary search tree and the time to sort for a dedicated sorting algorithm.

Let us actually test that theory in practice by sorting data in two ways. To do this, we first pick an array size $N$ and generate an array $A$ of $N$ random numbers. Then we sort $A$ with std::sort. We sort a copy of $A$ by inserting every element of $A$ into a std::multiset (since $A$ might contain duplicated items), and then retrieve the sorted list by iterating over the elements of $A$ (note that this second step is an $O(n)$ operation, so the time should be dominated by the part that inserts into $A$). Very simple code that does exactly this is available here.

An example of how to run the code is in genplot_example.sh. Basically, compile main.cpp however you want. The code loops through different values of $N$, which for simplicity are strictly powers of 2, and reports the time to generate the random numbers (for reference), the time to sort with std::sort, and the time to sort with std::multiset. To run, the first argument is the minimum power of 2 to test (so 5 means $N = 2^5 = 32$), the second is the maximum power of 2, the third is the step size (how many powers of two to move up per run, an integer), and the last is a boolean (0 or 1) telling the code whether to give verbose output (with text). To generate a plot with plot.py, use 0 for verbose.

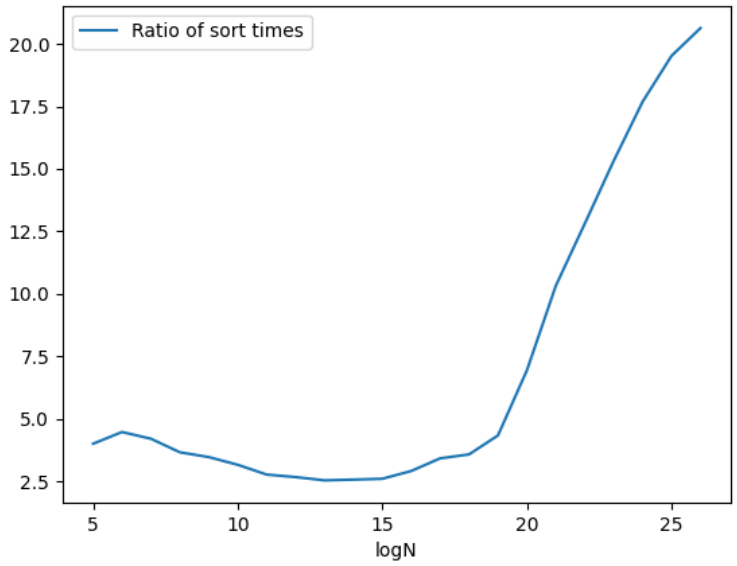
With all that aside, here is the output of the program. We graph $\log N$ versus log of the time (in seconds) to compute the result:




(Note: can you explain why theere is that brief flat region for sorting and searching and even to a lesser extent generating numbers somewhere around $\log N = 17$? I am actually not quite sure what is going on there. I have some guesses, but not in the least confident about any of them.

Also, why does gen time take longer for very small N? I am not sure, but I think the answer is that, to generate, I first get a seed by using random_device. This should ask for a hardware random number generator to give the seed, and these tend to be quite slow. On x86, maybe like 1500 cycles.)

If you notice, around $\log N = 20$, the difference between the time to sort using multiset and the time to sort using std::sort seems to go up, and remember that an arithmetic difference in logs represents a multiplicative distance in reality. The difference is actually growing! To really see how dramatic the difference is, here is a plot of the ratio of time to sort with std::multiset and the time to sort with std::sort:




It actually seems to go from almost 5.0 at $N = 6$ down to almost 2.5 at $N = 14$. That part of the graph seems normal. However, look at the sheer scale of the difference after that! After $N = 19$, the ratio skyrockets so that sorting with multiset takes 20 times longer!

*Note:* I wanted to also add a simulation of what would happend at really large values of $N$, something like $\log N = 35$, but I had some technical difficulties doing that (actually running $\log N = 35$ would take a completely infeasible amount of time!), but let me assure you of this: This ratio seems to be tapering off a little bit at the end of the graph and will even continue to do so going a little further. However, after jumping up a few more times (maybe around $\log N = 32$ on my computer), the ratio would probably jump to somewhere on the order of 1000 times. It depends on how you define constant factors, but I would strongly argue that these are *not* just constant factors at play!

### 1.2 Why?

We see this strange behavior, where, as $N$ goes up, our ratio does not seem to converge! Of course, it could be simply that we have not looked at large enough $N$, and we do have a constant ratio, so we just need to look at bigger $N$ (in some sense, one could actually argue this).

However, if we were to look at the number of operations being performed by each function, we would actually see a similar ratio (I have not verified this, and in the future hopefully will verify this using perf to count instructions). That is, even if we were to calculate constant factors, we would not actually be able to come up with an explanation for this difference! Therefore, even if it is constant, we should try and understand what is going on.

## 2 The Cache Hierarchy

Well, if what I am claiming is right, and it is not that the ratio of number of instructions is not changing (significantly), then one cause could be that some instructions get slower (or faster, but, well, this is *mostly* not the case) for larger $N$. In fact, this is exactly what is going on—loading data from memory is *expensive*. The time between requesting a piece of data from main memory and receiving it on a modern desktop takes somewhere between 50 and 100 nanoseconds, a number that has not changed significantly for 20 years!

However, processors, the chips actually executing instructions, have gotten something like 10 times faster (single core, but we have also moved towards having many cores on a single chip, with 4-16 being commonplace, and in addition to hyperthreading, which takes this further the total multithreaded performance is more like 25-200 times faster) for a very rough number.

You may say, "but we still have to load memory when $N$ is small!" and indeed we do. However, to fix the problem of memory being rather slow, modern processors have small *cache* that can store a small amount of memory. This cache is really fast, so it allows the CPU to continue processing operations.

## 2.1 Block Sizes

So, it takes a long time to fetch data from main memory, whatever. How does that impact these sorting algorithms? Fetching memory is costly for both algorithms, but they both still do $O(n \log n)$ memory accesses. Now, that could change the constant factor, if the ratio of memory accesses is different to the rest of the instructions, but, well, it should not be too drastically different—there is one memory load for every comparison and then another one for the pointer, but there are more stores in sorting, and so on. [Actually, in reality (we will discuss this later) even under this model it is possible to have sorting (asymptotically, under certain constraints) use fewer memory operations than searching.]

One real reason is that main memory cannot be fetched at such a fine level, and this is actually because of optimizations that hardware makers employ. On a rough first level approximation, consider if you are operating on some object or data structure. It might have a couple fields, and these are all stored together in memory. If you fetch one field, you're probably going to fetch the next field. Or imagine you are operating on the array. If you fetch one element, you are very likely to then fetch the next, and then the next, and so on.

If the CPU had to wait for each of these accesses, despite the likely location of the next access being quite obvious, we would have a very slow program. Therefore, instead of fetching one memory location at a time, the CPU makes memory requests in *cachelines*, typically of 64B each, to exploit the fact that one memory access is typically right next to another. 64B is not actually that big, but it does mean that if you fetch just one 8B pointer or something along those lines, you are wasting most (87.5%) of your memory bandwidth!

Now, I have been saying that main memory is slow, but accessing your hard drive or SSD, what can actually store your data when you poweroff your computer, is much, much slower. The delay between a request to get a piece of data from an SSD and actually getting it is between 50 microseconds (so 50,000 nanoseconds) and 200 microseconds, three orders of magnitude higher than RAM. If you want to fetch a random location from a hard drive, that's yet two more orders of magnitude, up to 5 milliseconds.

Therefore, the situation is even more extreme with these data devices, which are typically just called "external memory" in theory. Here, we refer to the basic unit of memory that we fetch as the *block size*. Block sizes are typically 4KB, although, in the case of hard drives, the real granularity at which you get the most performance at is often something like 4MB. Especially back in the days of hard drives, but even now with SSDs (also, hard drives are still common and cheaper than SSDs per gigabyte), the bottleneck of the external memory was (is) often so huge that it was (is) worth considering only how many times you access external memory and ignore all computation costs. This resultant "external memory model" is what we will talk about in the next section, but the rest of this section gives various details that somewhat complicate the picture in reality.

## 2.2 Bandwidth Versus Latency

We talked about how it is inneficient to wait for main memory (RAM)/external memory to return a value, so we fetch a large block of values at once. Now, this only makes sense if the external memory is actually able to fetch all those values at once. In fact, it can, and typically when people talk of the

"speed" of RAM/external memory, they talk about not the latency (time between requesting data and receiving it) but the bandwidth (amortized amount of data you can fetch at once).

Especially for external memory, the CPU can quickly fetch lots of data from

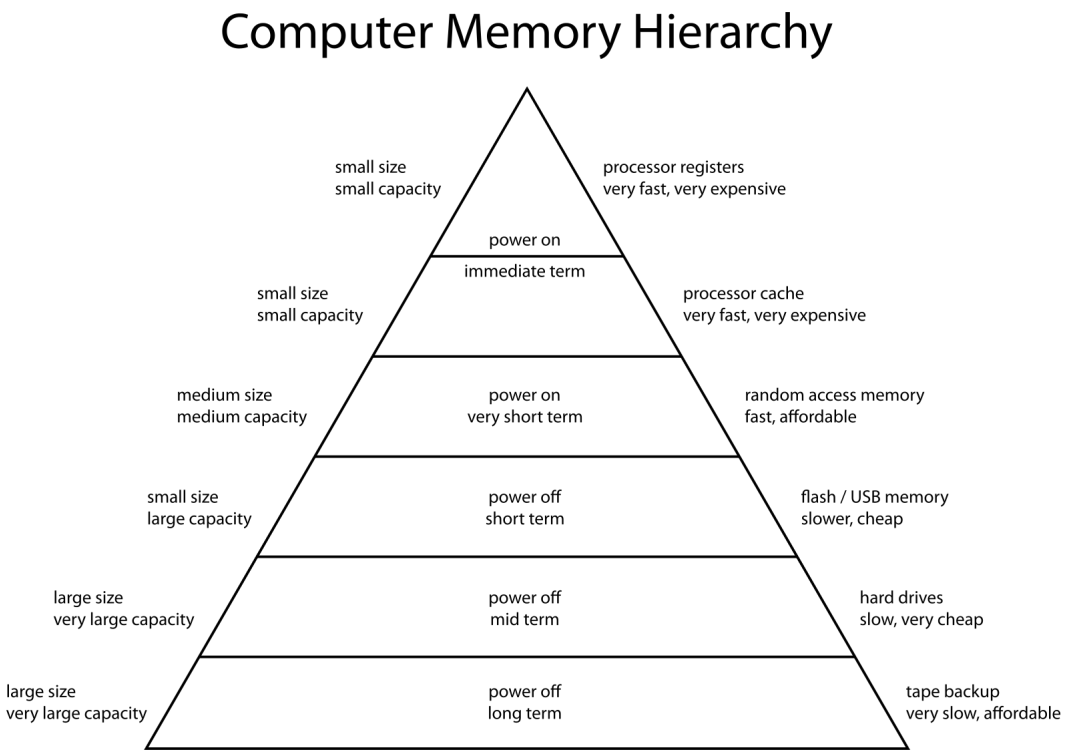
## 2.3 Caches all the Way Down

In fact, processors (CPUs) these days do not just have one cache, but multiple levels. You could already view the main memory (RAM) as a cache for the external memory, and the cache on the actual CPU as a cache for RAM. However, modern CPUs are really complicated and actually have several levels of cache on the CPU.

There is:

- L1: Very small (somewhere around 32-96 KiB typical) and very fast—something like 4 cycles of latency, almost unrestricted bandwidth. This is often split into L1D and L1I cache, which are explicitly only for data (L1D) or program instructions (L1I). Private to a core.
- L2: Larger but still very small. Really hard to give a range, but it might be something like 1 MiB per core, have a 14 cycle clock latency, and have slightly restricted bandwidth for some extreme cases, but usually this can still be treated as having infinite bandwidth. This can be shared amongst all cores, and it can be private per core.
- L3: Not all CPUs have this, but many do. L3 cache is usually shared across all cores, so this is kind of the fastest "synchronization point" for chips. The layout here actually matters a lot, as maybe some cores have fast access to certain parts of the L3 that they are physically close to and slower access to parts of L3 they are futher away from. A typical figure for L3 latency is something like 50 cycles, but again its somewhat complicated here.
- L4: Not common for a CPU to have a fourth level of cache, but it happens. Historically, this has typically not been directly on the CPU die, but placed really close to it for fast access. There isn't any rule here other than that it has higher latency and lower bandwidth than L3. Here is an overview of a recent chip with L4 done by servethehome.
- (D)RAM/main memory: This is the slowest tier fo memory that is volatile, or wiped away when you shut off your computer. It is also what most programs and data is stored on while being run. It is very important to note that: *caching is (usually) transparent to the program.* That is, it is done automatically. Using some caching scheme, the CPU automatically copies data from RAM to cache.
- Optane: No longer produced, but was a pretty cool technology that Intel promised would bridge the gap between SSDs and RAM.
- SSD: fastest (typically available) nonvolatile storage, although it is worth noting that some expensive RAM sticks have batteries/capacitors on them that allow them enough time to copy their data to nonvolatile storage. Reasonably cheap.
- HDD: Cheapest storage mechanism typically used within a computer. These are mechanical devices, and thus have their own quirks. While even for the above storage mechanisms, it can still matter how you access data, here it matters a lot. The device can only read point point of data at a time, and the physical read head has to first find that data, which takes like 5 ms, an eternity for computers. Actual linear bandwidth (so find the location and then continue linearly from there) is still reasonable. One thing is that data is stored two dimensionally in disks and somewhat three dimensionally by stacking a few disks on top of each other.
- Tape: This is only really used for archiving/backups, and there its a linear process to find/write data. Not even close to random access.

The general hierarchy of storage for a computer is typically summarized by a diagram like so:




## 2.4 Prefetching and More Complicated Things

Despite the fact that we have these cachelines or blocks, with SSDs and RAM it doesn't make sense to be forced to wait for a request to come back before fetching another piece of memory. Fetching 64B every 100 ns is still not near enough to saturate bandwidth, even with multiple cores doing memory accesses. For example, even with 16 threads all fetching 64B of data every 100ns, we get 64B * 10 million accesses/(thread second) * 16 threads $\approx$ 10 gigabytes per second, while RAM bandwidth can be 50-100 gigabytes for a common 8 core, 16 thread machine, and oftentimes you may only be using 1 or a few cores, making the difference even greater. A similar picture paints itself with SSDs, but I will not talk about that too much, and the solutions are in many cases software-based.

One thing that you could easily do to fix this issue is to just make cachelines larger. If you fetch 4 KiB per memory access, like on SSDs, then you solve this issue trivially. However, the bigger you make a cacheline, the more you are forcing the user to structure their data. What if instead you have a graph algorithm that follows edges? Then that may be really difficult, or, in some cases, impossible, to structure in this way. Therefore, there is only so big you can make cachelines before it becomes somewhat useless to do so.

Instead, CPUs do hardware *prefetching*. This is a term for requesting memory before you actually need it. For example, imagine I have an instruction stream where there are a bunch of arithmetic operations, but, in 100 instructions, there is some load from memory. Well, the CPU will simply prefetch that load. That is, CPUs look at future instructions to see what to do.

The most important aspect of this is that CPUs are actually out-of-order processors. If a CPU stalls on a memory access, it will actually continue executing any instructions that do not depend on that memory access, up to some hardware-defined maximum size. It will also continue this for another memory load that blocks operations, and so on. Doing this is rather complicated, as the CPU has to

keep track of which load/store operations are in flight (requested, no answer yet), it has to keep track of which instructions depend on which to only execute those for which you have up-to-date data, and so on. If you look at a real CPU diagram, it gets pretty complicated, but the rough idea is that CPUs have some reorder buffer window for which they can move around things to make execution as fast as possible.

Here, we are really dealing with architecture-specific things. Almost all CPUs in your laptop or phone have advanced reordering and prefetching, but simpler chips designed to use very little energy (like, microcontroller energy; phones have moved past this stage) may have much more basic versions. For example, some chips may have no prefetching or reordering at all, known as in-order processors. Some chips may be able to execute past one, and exactly one, slow memory load, and so on.

In fact, doing all these things is really inneficient and one of the big reasons why GPUs and other specialized hardware is widely used. GPUs have many more really weak cores, in some sense, so they do not need to do all these complicated things to get the GPU as a whole to be really fast. In terms of total operations (such as adding two numbers) done, GPUs completely outclass CPUs. However, CPUs offer this very flexible programming model that can run basically anything at least decently.

### 2.5 Cache-Oblivious Algorithms

We will not focus on these, but there is a class of algorithms that is "cache-oblivious," which in a rough sense means that it performs as well as possible, up to a (closer to "real") constant factor, at least with respect to accesses across different levels of cache. I actually myself am not that well versed on these [but maybe a future lecture could talk about these and some interesting things with them?] Cache-oblivious algorithms do not know the size of any of the caches, and they do not know the size of a block, but the basic idea is that they structure their accesses to "naturally" take advantage of caches. If you took computer architecture, you probably learned about these in the context of certain operations on matrices, such as transposing, and remember them better than I.

## 3 The External Memory Model—A Simplification

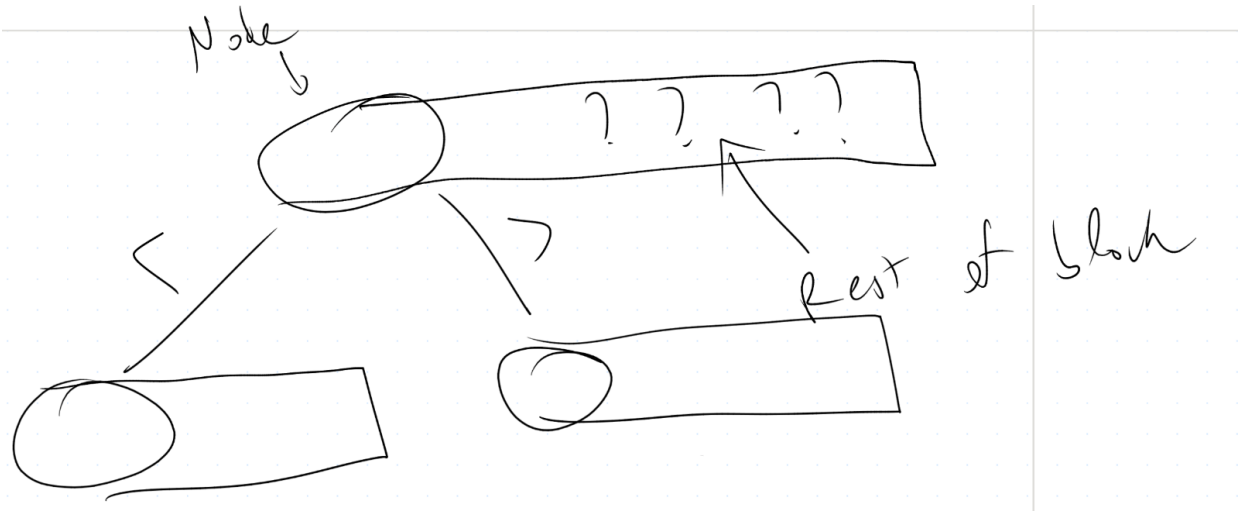
Recall that data is fetch from RAM or from external memory in a cachline or block. To match what people use in theory, we will use block for the rest of this section.

The external memory model is really simple. Instead of measuring the complexity of an algorithm by the number of instructions, we measure it by the number of block accesses made (either read access or write access). Data is fetched from external memory in blocks of size $B$, and we have a fast cache (for which memory accesses are considered to be free) of size $M$. An access of a block (of size $B$) has unit cost. That is, if we do 10000 operations and access 10 blocks of memory, our algorithm has total time 10.

Our focus here will be on large blocks like in SSDs, where the block size is $\geq$ 4KiB, but the analysis applies if we consider some extra details to RAM as well. The external memory really can be anything that is slow to access or do relative to other operations.

### 3.1 Binary Search Trees

To understand how external memory impacts us, let us consider binary search trees. How does this work? Apologies for the terrible picture, but here is an illustration:




When you fetch a node from memory, you don't fetch just the 8 bytes for the key to compare against plus 8 bytes for the left pointer plus 8 bytes for the right pointer, but also the rest of the block, which is thousands of bytes for SSDs. What is in the rest of the block? Other search nodes, but you have no idea if they are helpful to you.

In this case, you have to fetch a block for every single node. We can reasonably expect that the upper nodes, which are accessed more frequently, will be in cache. Therefore, the first $O(\log M)$ nodes we can expect to be free. After that, however, we incur a unit cost per node. There is a total depth of $O(\log N)$, so we expect to have $O(\log N - \log M) = O(\log N/M)$ memory accesses per search.

To sort with a binary search tree, we thus spend $O(N \log N/M)$ time.

This may seem not too bad, but remember that we can store many keys in one node, and we are wasting a massive amount of bandwidth if we just access data about one key (remember, in the normal RAM model, ONE external memory access costs $O(B)$!). As we will see in the next subsection, if we can amortize one memory access across a lot of keys, we can drastically speed things up.

## 3.2 Merging

Now, let us look at a simple operation that gives us a glimpse of the power of external memory algorithm analysis, merging. Merging is a key component of sorting, and it turns out we can do it rather fast.

Consider if we have two sorted arrays, $A$ and $B$ stored on external memory, each of size $n$ (for simplicity, let $n = Bk$). Each array is physically divided up into blocks of size $B$ on external memory. Each block is its own mini little sorted subarray, and let us call the $i$th block of $A$, $A_i$ (and similarly for $B$).

Now, let us think about what we need for merging. What do we need to know to get the first element of the merged sorted array $C$? We need to know the minimal element of $A$ and the minimal element of $B$ to compare them, as any other elements in $A$ are bigger than the minimal element in $A$ and ditto for $B$. But what about the first $B$ elements of $C$, since that is what we really care about?

## 3.3 Sorting

Total time(IOs) to sort: $O\left(\frac{N}{B} \log_M N\right)$—really close to linear for realistic sizes of memory vs external memory!

### 3.4 Searching—B-Trees

Total time(IOs) per query/insetion: $O(\log_B N)$!!
Total time(IOs) to sort: $O(N \log_B N)$—waaaaaay worse!

## 4 A New Hope for Search Trees—Insetion-Optimization

### 4.1 Log-Structured Merge Trees

### 4.2 B$^\varepsilon$-Trees—Achieving Optimality (Under comparison model)

Time for query changes by only a constant factor, but time for insert: $O\left(\frac{\log_B n}{B^{1-\varepsilon}}\right)$—less than one for typical "external memory!"
Note that can "stream" off a B$^\varepsilon$ tree in linear time, so the query performance is not so important here. Time to sort is thus $O\left(\frac{N}{B^{1-\varepsilon}} \log_B n\right)$—still significantly worse, but not too bad!

These are actually optimal for search trees, and so we cannot get any better.

### 4.3 State of the Art—SplinterDB and the Mapped Size-Tiered B$^\varepsilon$ Trees

Now, let's shed our chains of sorting completely and talk about search in general. As you mostly likely know well, there are two types of search data structures—hash tables and search trees. Turns out that, if you want to insert-optimize hash tables at all, you have to have similar tradeoffs to search trees. However, you can get slightly better bounds than for search trees, and, more importantly, you can achieve a fast hash table simultaenously with a pretty fast search tree in one structure! That is what the state of the art and probably not too far from the end of the line (although interesting ideas will always abound, and the question of making these structures more general certainly exists), SplinterDB achieves.

### 4.4 Note on Many Computational Models in General

For the external memory model, we identified a really slow part of our computation, fetching external memory, and then focused on optimizing that. While it has this significance in computer hardware, there is nothing really special about optimizing external memory accesses.

## 5 Another Interesting Model—The Word-RAM

We can also expand our computational models in different directions. You probably learned that sorting must take time $\Omega(n \log n)$, but this only really applies to *comparison-based* sorting algorithms. What if we do something clever?

The idea is that CPUs do not just operate on abstract numbers. Integers (and floating point numbers too!) are stored as strings of 1s and 0s, and you can compare them the same way you compare the ordering of words in a dictionary—compare the first bit on which they differ. This leads to a variety of algorithms that resemble(or directly use) a hash table, as well as some other algorithms that do cute little bit tricks.

In the word RAM model, we consider that the CPU operations on a finite number $w = \Omega(\log n)$ (since to access element $i$ of array $A$ of size $N$, we actually need to have enough bits to identify $i$, although there are some methods to go around this) of bits. In some sense, machine instructions operate in parallel on all these bits, and we can extract some of this parallelism for faster searching and sorting.

## 5.1 Radix Sort/Bucket Sort

The simplest non-comparison based sorting algorithm is probably bucket sort. Imagine that we have a list of $N$ numbers, all between 1 and $N$. Well, then we can simply just put all the 1s in the front, then put all the 2s after them, and so on. To do this, we can simply count the number of 1s, number of 2s, and so on in the table. That is, we start with an array $A$ of size $n$ that we want to sort and create an array $B$ of size $n$ initialized to zero. For each $i$, we have $B[A[i]]++$. After we run through $A$ once, we then simply plot $B[1]$ 1s in the front, followed by $B[2]$ 2s, and so on. Running through $A$ like this takes $O(n)$ time, and, since the largest number in $A$ is $N$, running through $B$ also takes $O(N)$ time. This algorithm runs in $O(n)$ time.

Radix sort then takes this idea and applies it recursively. Since numbers on computers are simply strings of digits ("binary" but we can group them to make bigger "digits"), we can sort by the most significant digit, then the second most, and so on. Counterintuitively, many practical radix sort algorithms go the other way and sort by the least significant digit going to the most significant digit. If your number is $b$ digits, sorting takes time $O(nb)$. Since as per the bucket sort algorithm, we can make each digit have values from 1 to $N$, we can have each digit be $O(\log n)$ bits, meaning that $b = w/\log n$. We can thus sort in something like $O(Nw/\log N)$ time. If the size of your numbers is polynomial in $N$, ie. $A[i] \leq N^c$ for all $i$ for some constant $c$, then $w = O(\log N^c) = O(c \log N) = O(\log N)$, so sorting is $O(n \log N/\log N) = O(n)$!

## 5.2 vEB Trees and Fusion Trees

What if your word size is bigger, though? What if like $w = O(n)$, in an extreme case? Then radix sort would be way slower than just normal comparison-based sort? Well, there are more advanced algorithms, that actually are based on search trees (radix sort is also treeifyable) that let you do better in this case.

Two really interesting (and either actually useful or almost useful) algorithms for arbitrary number sizes (so not just $w = O(\log n)$). vEB trees actually enable searching/inserting in $O(\log(w))$ time per key, meaning that they are most effective when $w$ is "small." Fusion trees enable searching/inserting in $O(\log_w(n))$, meaning that they are fast for "large" word sizes. Combining these two algorithms, we get $O(n\sqrt{\log n})$ in the worst case over the range of possible $w$.

I think these are the fastest (known) trees with guaranteed worst case performance, but I am not sure. If we expand to allow amortized time rather than worst case time per operation, there is a search tree that enables $O(n\sqrt{\log n/\log\log n})$, but that is more complicated. This has been proven to be optimal. We cannot do better than $\Omega(n\sqrt{\log n/\log\log n})$.

## 5.3 Sorting in Linear Time?

There is no such lower bound for sorting. In fact, there is one really interesting sorting algorithm that achieves $O(n)$ sorting when $w = \Omega(\log^{2+\varepsilon} n)$, so the keys we are sorting are "large" in some sense, which may seem a bit surprising at first—why would sorting big numbers be easier? Again, remember that the CPU in some sense enables parallelism on these bits, so you can do some pretty interesting things. For general $w$, the fastest achieved is $O(n\sqrt{\log\log n})$, but I do not recommend you try reading that paper. I think the paper that does the linear time sorting for large word sizes is pretty interesting.

Obviously, to read all the numbers we need time $\Omega(n)$, so we need at least that much time to sort. The big obvious question here is: Can we sort in linear time (for all $w$)?