# Sorting & Searching

Orphi the AweKid

12th December 2008

**Abstract**

Computers are renowned for their ability to store vast quantities of data and rapidly retrieve it again. However, finding data fast is about more than just raw speed; it requires clever techniques. This document provides a layman's overview of some of these techniques.

## 1 Introduction

When you walk up to a cashpoint, you take it for granted that the bank's computer can instantly find *your* account and check your balance. But think about this: How many accounts does a large highstreet bank actually have? Thousands? *Millions?* And yet, it can find your account amoung all the other accounts it has in a matter of seconds. How?

Of course, the modern electronic computer is an extremely high-speed device. Most people would probably assume that this alone is the sole explanation. But it is not. Actually, if the bank's computer were to search for your account in the most obvious way, it would take an absurd amount of time to find it.

In order to find information in a reasonable amount of time, modern computers employ a range of interesting techniques to speed up processing. In this document, you'll see what some of those techniques are.

## 2 Searching

### 2.1 Introduction

The problem we are concerned with is *searching*. Given a huge stack of items, how can we find the one we want quickly?

To put this into perspective, suppose I have you a stack of ten million invoices, and ask you to find invoice #3,768,242. Assuming the invoices are all muddled up in no particular order, the only way you can find the right invoice is to just search through the whole pile, one invoice at a time. That's going to take you... quite a while. (!)

For a sufficiently large number of invoices, that's going to take quite a long time even for a computer. Clearly we need something better.

### 2.2 Sorting & Searching

If you have a pile of invoices in no particular order, the invoice you're looking for could be *anywhere* in that pile. Thus, to find your invoice you

will have to look *everywhere*. There is no way around it. This method of searching is typically called a *full scan*.

If, on the other hand, the invoices are arranged in some sort of *pattern*, it is possible you might be able to figure out where the invoice you want is based on its serial number. The most common way to do this is by *sorting* the pile.

If all the invoices are in *sorted* order, you can usually find what you're looking for much more quickly. Thus, sorting and searching are often studied together, because they are related problems.

## 2.3 Binary chop

How *exactly* does having the invoices sorted help you find things more quickly? Well, there are several ways a human being might use this information, but computers are more precise and methodical. One of the ways a computer might be set up to approach the program is a method called *binary chop*.

The method is very simple. Take your pile of invoices, and look at the one in the middle. (It doesn't matter if it's *exactly* the middle, but ideally as close as possible.) Look at the number of that invoice. If the number you're looking for is higher than that, the invoice is in the top half of the pile, otherwise it's in the bottom half of the pile.

We have now cut the pile in half. We know that our invoice is in one half, so we don't even need to *look* at the invoices in the other half — a huge saving of work. But wait! We can take our new, smaller pile and split it in half again in exactly the same way. In fact, we can continue to split it into smaller and smaller piles until we find what we're looking for.

## 2.4 A speed comparison

Suppose we have ten million invoices. How long will it take to find one of them by a full scan? Well, once we *find* the one we want, we can stop looking. On average, we'll need to look at roughly half of the invoices before we find the one we want. (Sometimes it will be more, sometimes it will be less, but it averages out as half.) So that's five million invoices we need to inspect, on average.

How about with binary chop? Well, in that case we start out with a pile of ten million invoices. Then we cut it in half, and ignore one half. So we now have only five million invoices to worry about. Then we cut *that* pile in half, and so forth. In summary:

| | Invoices examined | Invoices remaining |
|---|---|---|
| Initially: | 0 | 10,000,000 |
| After 1 step: | 1 | 5,000,000 |
| After 2 steps: | 2 | 2,500,000 |
| After 3 steps: | 3 | 1,250,000 |
| After 4 steps: | 4 | 625,000 |
| After 5 steps: | 5 | 312,500 |
| After 6 steps: | 6 | 156,250 |
| After 7 steps: | 7 | 78,125 |
| After 8 steps: | 8 | 39,063 |
| After 9 steps: | 9 | 19,532 |
| After 10 steps: | 10 | 9,766 |
| After 11 steps: | 11 | 4,883 |
| After 12 steps: | 12 | 2,442 |
| After 13 steps: | 13 | 1,221 |
| After 14 steps: | 14 | 610 |
| After 15 steps: | 15 | 305 |
| After 16 steps: | 16 | 153 |
| After 17 steps: | 17 | 77 |
| After 18 steps: | 18 | 39 |
| After 19 steps: | 19 | 20 |
| After 20 steps: | 20 | 10 |

If we continue the table a little further, it basically works out that we need to examine roughly 25 invoices — as opposed to 5,000,000 invoices for a full scan. I now present a very simple calculation:

$$5{,}000{,}000 \div 25 = 200{,}000$$

In other words, for a list of ten million invoices, a binary chop search is *two hundred thousand times faster* than a simple full scan.

To put it another way, suppose you can check one invoice per second. How long will each method take?

| | |
|---|---|
| Binary chop: | 25 seconds. |
| Full scan: | 2 months. |

Note that the "2 months" assumes that you never stop work for a single second for two months solid. (No eating, no sleeping, no blinking. . . )

The results aren't much different for a computer. If we assume that ours can check a thousand invoices per second, we have

| | |
|---|---|
| Binary chop: | 0.025 seconds. |
| Full scan: | $83\frac{1}{3}$ minutes. |

So one way, the computer takes a split second, the other way it takes almost an hour and a half! Would *you* rather wait a split second or an hour for your bank balance?

Notice that the computer can perform 40 full scans per second at the speed indicated above. Sorting and searching isn't just about looking up financial records. Something like a spell-checker may need to look up a few hundred words in its dictionary in order to determine whether your 300-word assey is spelt correctly. You do *not* want to wait an hour for each word to be looked up!

(In case you're wondering, typical spell-checker dictionaries often contain tens of thousands of words at a minimum.)

If you *double* the number of items to search through, a full scan will take twice as long, but a binary chop will only take *one* extra step. This

is the key to its power. For really small lists, a full scan can actually be faster because it's simpler. But as the list gets bigger, the full scan slows down far more rapidly than binary chop.

# 3 Keeping it sorted

## 3.1 Introduction

A binary chop search is wildly faster than a full scan. It's not 10% faster or 20% faster, it's *thousands of times* faster. But there is a little snag: full scans always work, but binary chop requires the data to be stored in sorted order.

Aside from the problem of getting things into sorted order in the first place, keeping them that way is a problem. In our "pile of invoices" example, each time you add a new invoice, you can't just drop it on the top of the pile, you have to carefully insert it into the correct position.

## 3.2 Arrays

Inserting a paper invoice into a pile is just a matter of slotting it into the right position. For a computer things aren't so easy. Computers usually store data as an *array*, which is simply a row of numbered slots. Each slot contains a single item.

If the array is not sorted, each time you want to add a new item you can just put it in the next free slot at the end of the array. However, if you want to insert a new item into somewhere in the middle of the array, you have to move all the items in the right half of the array up one slot to make a space for the new item. If the array contains millions of items, moving half of them will be very slow.

In fact, inserting an item into a sorted array is as slow as performing a full-scan search on an unsorted array. (And that's assuming you *already* know exactly where the new item needs to be put; in reality, you need to perform a lookup operation first *as well*.)

In short, unsorted arrays have fast insert and slow lookup, while sorted arrays have fast lookup and slow insert. Neither of these options are very satisfying. (Remember, "fast" means split seconds, while "slow" means *hours*. How would you feel if it took an hour to add a new word to your spell checker's dictionary?)

## 3.3 Linked lists

Instead of using an array, another commonly-used computer storage structure is a *linked list*. With this method, each item also has a *pointer* stored with it. This pointer tells the computer where the next item in the list is.

The advantage of this is that the list elements can be *stored* anywhere you like; the pointers define the ordering of the elements, so you don't need to move elements around to "make space" for new ones. When you add a new item, store it wherever is convinient, and then adjust the pointers as necessary to add it to the correct position in the sequence.

So adding an item to a linked list is fast. Unfortunately, this doesn't help us though, because *accessing* a linked list is slow. Specifically, you can't jump directly to the middle of such a list; you have to start at the beginning and follow the individual pointers, one at a time, like a trail of breadcrumbs.

The net result is that a linked list is a bit like a tape. You have to "fast-foward" or "rewind" to get to the position you want; you can't just jump directly to (say) the 5,000th element. Unfortunately, to perform a binary chop, that's *exactly* the sort of thing we need to do!

Since inserting into a sorted pile requires finding the right spot first, what all this means is that a sorted linked list is slow for lookup *and* insert. We're even worse off than with an array!

## 3.4 Binary search trees

Linked lists are no help to us, but they do hint the way. By using a structure where pointers are used to indicate the relationships between items rather than their storage position, we avoid the need to relocate things when we want to insert a new item. But unfortunately, we also lose the ability to jump to an arbitrary position in the structure.

So linked lists can't help us. But that's because the pointers don't point to where we're trying to go; they point to the *next* item, but we want the *middle* item.

There is another structure based on pointers, known as a *tree*. The simplest kind is a *binary tree*, and it's ideally suited to binary chop. It works like this: We have a *root node*, which contains the "middle" item from our "pile". It also contains two pointers — one to the *left branch* (containing all the items that come "before" the middle one), and one to the *right branch* (containing all the items that come "after' the middle one).

Each branch has the same structure as the main tree; the pointers in the root node point to *child nodes* which contain the middle item from that half of the pile, and pointers to further child nodes, and so forth, until you get to the *leaf nodes* which each contain an item but no further pointers.

This structure exactly matches what we want to do with binary chop. To perform such a search using a binary tree, start from the root node. Compare the item there with the item you're looking for. If it's higher, take the right branch. If it's lower, take the left branch. Repeat until you find what you're looking for.

When a binary tree is used in this way, it's generally called a *binary search tree* ("BST"). The "binary" part refers to each node having two branches; it is possible to build trees with more than just two branches. (We'll see what that might be useful later.)

## 3.5 Indexes

An *index tree* (or just *index*) is essentially a search tree where the "items" are stored somewhere else rather than in the tree itself. (In other words, the tree contains pointers to the items rather than containing the items themselves.) The advantage of doing this is that the same heap of items can be indexed in multiple ways.

For example, if you have a pile of paper invoices, you can sort them by number, or by customer name. But you can't sort them both ways; you can only sort them one way or the other. If you regularly need to look up by number *and* by customer name, you've now got a problem. Even if you use a binary search tree, you still have a problem.

Indexes solve this problem. You store the invoices themselves in an unsorted array, and you have to seperate indexes that point to the invoices.

One index is ordered by invoice number, and the other by customer name. (And you can add others too if you wish.) Now to look up an invoice by number, you go through the number index and get a pointer into the array, but to look up by customer name, you look through the other index instead.

## 3.6 Balancing trees

Looking up information in a tree (whether a search tree or an index tree) is nice and fast, essentially using the normal binary chop method. But what about inserting new items? This involves doing a normal lookup operation to find the spot in the tree where the item "should" go, and then creating a new branch to put it in and attaching it to the rest of the tree. Thus an insert is nearly exactly the same speed as a lookup.

So here we have a structure that gives us fast lookups *and* fast inserts. There is, however, one *small* problem remaining. For best performance, the tree needs to be *balanced*.

Put simply, this means that each pair of branches should be (roughly) the same size. That way, each time you choose one branch rather than the other one, you are eliminating roughly half of the nodes from your search. If (say) one branch was 5 times bigger than the other, then picking the small branch would eliminate 4/5 of the remaining nodes, which is great. However, picking the large branch would only eleminate 1/5 of the remaining nodes, which isn't so good.

As a more extreme example, if lots of items are added to the tree that all happen to have values that fall within one particular branch, that branch might end up 200 times larger than its sibling. Then anybody choosing that branch (highly likely, given that it's so large) would only have saved themselves 1/200 work.

In the worst possible case, if the tree is severly unbalanced, each branch choice might only eleminate a handful of nodes from consideration. This can quickly make the tree slow down to almost the speed of a full scan — i.e., extremely slow!

The solution to all of this is to periodically *rebalance* the tree. There is a vast *ocean* of different methods for deciding when to do this, and how best to do it. The point is, no matter what kind of tree design you use, the tree *must* be rebalanced from time to time, and rebalancing it takes time (i.e., it slows things down).

Some methods involve rebalancing the tree every single time a new item is inserted. Others just rebalance the whole tree after every $x$ inserts. Other methods try to detect "how badly" the tree is unbalanced, and only rebalance it when a certain threshold is reached. There is a huge amount of research into the best ways to do this kind of thing, and it's still ongoing today.

## 3.7 Deletions

Notice that *deleting* items is quite similar to inserting new ones. We have to find the item before we can remove it. If we're using an array, we need to move everything to the right one slot left to fill in the gap thus created. (Unlike insertion, *this* problem applies whether the array is sorted or not!)

With a tree, deletion is almost as easy as insertion. You do a lookup to find the item of interest. If it's in a leaf node, just delete the whole node from the tree. Otherwise, you'll have to move data up from the lower

nodes to fill the now vacated "hole" when you do the deletion; this still doesn't take much work.

Like insertion, deletions can unbalance the tree. Again, you can rebalance at every single deletion, rebalance after every few deletions, or try to detect when the tree is losing performance and rebalance then.

### 3.8 Trees on disk

Search trees and index trees are fundamentally based on random access. When data is stored in a computer's memory, flitting from place to place is no problem. However, when data is stored on disk, it's (comparatively) slow to move the read/write heads back and forth to access data scattered around at random.

For this reason, trees stored in memory are typically *binary* trees (i.e., each node has two branches), but trees on disk typically have a lot of branches per node (e.g., 20). Each (non-leaf) node contains (pointers to) several items instead of just one, and the branch pointers are to the middle of the subranges delimited by these items.

This slight increase in complication means that you can find what you're looking far while examining fewer nodes. Each node has more data in it, so the theoretical total number of "operations" performed is roughly the same. However, since a whole node can be read off disk all one once, "in one go", reading fewer nodes results in better performance.

## 4 Hash tables

### 4.1 Introduction

Storing your data is *sorted* order is one way to speed up the process of finding it. But it isn't the only way. All that really matters is that data is stored according to some sort of pattern. It doesn't particularly matter what pattern, so long as there is one.

An alternative approach is to use a so-called *hash table*. This is similar to a sorted array. However, the problem with a *sorted* array is that the correct position for each item depends on what other items are there. Thus, when you add new items, you have to move the existing ones around, which is a lot of work.

### 4.2 Hashing

Instead of sorting the elements, a hash table uses a special calculation known as a *hash function* to decide where to put each item. Each item is stored and looked up via a *key*. For example, if you're working with invoices, the key might be the invoice number. You hash function could be something like "take the digits of the key, multiply each one by 7, and then add them all together". The code thus produced is called the *hash code* of the key.

So, to put an item into a hash table, you take the key, use a hash function to calculate a hash code from it, and then store the item in the slot with that number. (E.g., if the hash code is 137, store the item in the 137th slot in your array.) And to lookup an item, take the key you want (e.g., the invoice number) and put that through the hash function. This will calculate the same hash code as before, and tell you exactly which slot to look in to find your invoice.

In summary, both insert and lookup operations are extremely fast. The hash function tells you *exactly* where to put stuff, without you having to do any "searching" at all! It sounds like a hash table is truly the perfect data structure.

## 4.3 Collisions

There is a snag, however. Sometimes, it may happen that two different keys just happen to produce exactly the same hash code. When this happens, it's called a *collision*. A carefully designed hash function will minimise collections, but avoiding them is harder than it sounds. There are several things you can do when a collision happens — but that's just it. You have to *do* something, and doing things takes time.

If collisions never happened, the hash table would indeed be the perfect data structure. However, if *a lot* of collisions happen, hash tables end up being slower than almost any other kind of data structure. And whether or not you get collisions depends on your hash function, and your keys. It also depends on how full the table is. As the table fills up, collisions become more and more likely.

The crucial problem is that a given hash function might work brilliantly for one set of keys, and might work really badly for a different set of keys. And it's almost impossible to know ahead of time. The result is that hash tables have "unpredictable" performance; sometimes they work very well, sometimes they work extremely badly.

When they work well, they out-perform any other data structure known to man. But when they work badly, they work very, *very* badly! Also, they tend not to be very space-efficient; collisions are lowest when the table is emptiest, but a huge empty table is a bit of a waste of space.

## 4.4 Probing

So how can we handle collisions? One possibility is called *probing*.

If the slot where we were going to store the item is already taken, store the item somewhere else. In the simplest case, use the next slot along. But what if *that* slot is taken as well? Then we use the next one along. In other words, we start from where the hash code says the item should go, and keep looking until we find an empty slot. This is called *linear probing*.

We have to go through the same process when looking something up. If it isn't where the hash function predicted, maybe it's in the next slot along? Or the next one? Only if we come to an empty slot can we conclude that the item we seek isn't in the table at all.

An "ideal" hash function should sprinkle the items evenly over the array, but in practise this doesn't always happen. Sometimes items get clusted together in one spot. For this reason, more complicated probing sequences have been invented. Rather than trying the *next* slot to see if it's free, some more complicated method is used for choosing the next slot to try. This is intended to overcome clumping, but it doesn't always work.

## 4.5 Cuckoo hashing

Another possibility is called *Cuckoo hashing*. Here you have *two* seperate hash functions, and thus for any given key there are two places where you

could store the thing. When you want to store something, you put it into it's A-slot. If that's already taken, you remove whatever is already there and move it to it's alternate slot. And if something is already *there* too, you move that... and you keep moving things around until finally you happen upon an empty slot.

Of course, in practice, almost all items fit into their A-slot first time. Occasionally you have to free the A-slot first. And very occasionally you have to do more work. And when looking up an item, you only ever need to look in two places (so it's very fast).

Assuming the table is less than 50% full, Cuckoo hashing usually works quite well. But as the table fills up, the "chains" of items needing to be moved get longer and longer until eventually you run out of free spaces all together. (How quickly this happens depends on how good the two hash functions are.)

### 4.6 Chaining

A final possibility is to make each "slot" in the array point to a "bucket" — a linked list that can store an unlimited number of items. This way, when a collision happens, you just add the new item to the end of the list, and during a lookup you just full scan the list.

Assuming you have 100 buckets and the hash function sprinkles items evenly amoung them, each bucket contains 100 times fewer items than if the whole lot was just one giant list. Thus it's about 100 times faster to find anything.

This method is called *chaining*. The main advantage is that as the table fills up, performance decreases *gradually*. By constrast, a hash table based on probing is faster initially, but after a certain point it suddenly slows down quite drastically.

Because of this, hash tables that use probing typically have to be kept mostly empty to prevent collisions, but hash tables using chaining can be significantly "overfilled" without too much loss of performance. So chained tables are more memory-efficient, but typically slightly slower.

Another advantage of chained hash tables is that it's easy to *delete* items. If you're using probing, you need to take some kind of action so that items that "should have" gone in the slot you just emptied can still be found. (Remember, searching usually stops the first time it hits an empty slot.) With chaining, you just delete the item from a linked list — which is easy.

## 5 Bulk sorting

### 5.1 Introduction

We've talked about how storing things in sorted order makes finding stuff faster. But we haven't talked yet about the actual sorting process itself. Let's look at that now.

Suppose you have a stack of a thousand invoices sat in front of you, and you need to sort them into numerical order. No matter which way you do it, it's going to take you a while. However, some ways will take vastly longer than others.

## 5.2 Insert-sort

One method you might choose is this: You start with a full "in" pile and an empty "out" pile. You pick up the top item from the in-pile, and insert it into the correct position in the out-pile. In this way, the in-pile gradually empties, and the out-pile gradually fills.

This method is called an *insert-sort*. For small numbers of items, it's fairly fast. However, as the in-pile fills up, it becomes increasingly slow to insert new items into the correct place. For this reason, this is one of the lowest sorting methods.

(It is especially slow for a computer, since inserting into an array requires moving the elements after the insertion point, and using a linked list prevents binary chop from being used to find the insertion point.)

## 5.3 Merge-sort

When using an insert-sort, doubling the number of items *quadruples* the time required to sort them. As a result, small quantities can be sorted rather quickly, but for even slightly larger quantities, speed decreases rapidly.

But here's an interesting thing: if doubling the size quadriples the time, then logically enough halfing the size should quarter the time. Weirdly, that means that if you split the pile into two halves, and sort each pile independently, sorting each one takes $\frac{1}{4}$ the time, yielding $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ time in total.

We're not *quite* finished, however. We still need to "merge" the two piles together to produce a final, sorted list. But we can do that by just comparing the top-most item on each pile, and choosing the appropriate one to add to the final out-pile. So we only need to make one pass over both the small piles. Compared to the time to *sort* the small piles, this is almost negligable.

So here we have a method where it takes (slightly more than) half the normal time to do the sorting. But we can of course repeat the trick: We take the original pile, split it in half, split the halves into halves, and just keep splitting until we end up with (a huge number of) really tiny piles. These tiny piles are then easy to sort, and we then reverse the process, merging the tiny piles into bigger and bigger piles until we have just one giant, fully-sorted pile at the end.

Sorting this way is called a *merge-sort*. Like full scan vs binary chop, for large amounts of information a merge-sort turns out to be *vastly* faster than an insert-sort.

## 5.4 Quick-sort

Another method is called *quick-sort*. It works in a very similar way to a merge-sort. First, we pick a *pivot value*. We then go through the in-pile and put all the items below the pivot value into one pile, and all the values above the pivot value into another pile. We then process the two new piles in the same way, until eventually we get down to piles small enough to sort directly.

The merge-sort splits piles in half, sorts them (by further splitting and processing) and then carefully merges them back together. Quick-sort does it the other way round: piles are carefully split apart, and once they've been sorted (by further splitting, etc.) they can just be plonked

one on top of the other without any further processing. (Remember, one pile contains "high" items and the other contains "low" items.)

In summary, merge-sort does the work during merging, and quick-sort does it during splitting. *Theoretically* both methods are equally fast. However, due to particular characteristics of quick-sort that I won't go into, quick-sort is usually slightly faster than merge-sort when run by a computer.

There is a catch, however. Merge-sort is *always* fast. But quick-sort fundamentally relies on choosing the right pivot value at each step. If you choose a good pivot value, you'll split the pile roughly in half. If you choose a bad pivot value, you could end up doing a whole heap of work and ending up with one tiny pile (which is no problem) and one giant pile that isn't much smaller than when you started (which is a waste of effort). In the worst case, quick-sort can end up *slower* than a plain insert-sort!

## 5.5 Other sorting methods

There are still other sorting methods available. For example, a *heap-sort* involves using a kind of tree called a *heap tree*. It's similar to a search tree, but simpler to manipulate. Losely speaking, a heap-sort involves putting all of your data into a heap tree (which partially sorts it) and then taking the data out again in sorted order.

Another simple method is a *bucket-sort*. This involves setting up a number of "buckets", each covering a particular value range, and then putting all of the items from the unsorted pile into the appropriate bucket. The bucket contents can then be sorted somehow. In other words, a bucket-sort isn't a complete method, it's part of a complete method. (In a way, a quick-sort is like an extended, repeating bucket-sort.)

A bucket sort is often used as the first stage of sorting very large amounts of data stored on disk. Disks are fastest to access *sequentially*, but sorting generally requires *random* access. So typically the information is split (possibly in multiple stages) into buckets small enough that the contents of a single bucket can be held in the computer's random-access memory for final sorting.

Yet another method is a *shell-sort*, which involves a strange kind of shuffling. Stranger still is a *bubble-sort*, which is very easy to program into a computer, but very inefficient.

Theoretically, bubble-sort and insert-sort are both nearly the same speed, and merge-sort, quick-sort, heap-sort and shell-sort are all nearly the same speed. In reality, different kinds of sorting algorithms have slight benefits and drawbacks in a particular situation.