Beauty In the Eye [1]

Robin Dawes

October 17, 2021

[1] ... is worth a bee in the hand!

RUDOLF BAYER spent a lot of time thinking about trees. We've studied his most famous creation (Red-Black Trees) already, and in these notes we will climb into one of his earlier designs: B-Trees.

Setting the Scene

SINCE THE EARLIEST DAYS of computing we have struggled with the problem of managing quantities of data that exceed the internal capacity of our computers. With hardware accelerations and ever-cheaper memory the issues have receded somewhat for the everyday user, but the optimal use of external storage is still important for very large data sets.

Let's suppose we have a very large collection of large data objects, each with an unique key. An illustration of this would be a set of medical records. For each patient there is an unique healthcard number, and the quantity of information for each patient might be very large, listing all of the patient's health history, including x-ray images etc.

The goal is to find a given patient's records quickly.

The key issue is access-time. In the very old days, external storage was largely tape-based. We've all seen images and movies featuring banks of huge tape drives with the tapes whirring back and forth. Slightly more recently, enormous hard-drives with stacks of disks in them became the storage medium of choice.

In the days before colourful screens, computers were mostly just immobile boxes with a few blinking lights - the tape drives were usually the only visibly active devices in a computer room. It's no surprise that photographers and film-makers focused on them.

With tape drives and mechanical disk drives, before we can read stored data the read/write head must be positioned over the requested data. Long moves or frequent moves of the r/w head can easily become the bottle-neck of an algorithm if the data is not carefully organized on the device.

Note that this is never a concern for data structures that are completely stored in internal memory (RAM). By definition, RAM provides constant time access to all memory locations.

When reading data from an external device, there is typically a hardware-dependent limit on the amount of data that can be transferred in a single operation. We can call this a BLOCK.

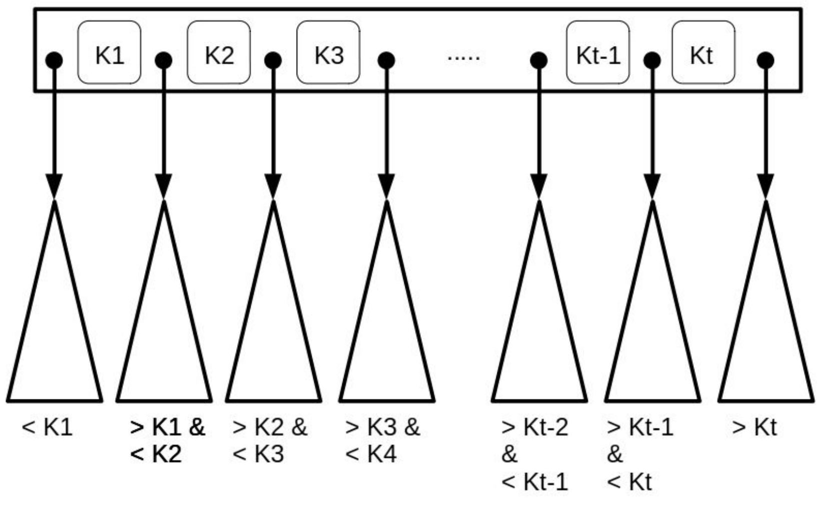
We will assume that our records are large enough that each block can only hold a few records. The question to be answered is: what is the best way to arrange the records in blocks so as to minimize the number of blocks we must access to find a particular record?

A B-Tree is an k-ary tree in which each vertex can have up to k children for some integer k. There are some more rules which we will introduce as we go along.

In a B-Tree, each vertex of the tree occupies a single block on the external device. This means that each vertex can contain several records. As a tree vertex, the block must also contain links to its children. We will assume that these links are identifiers for other blocks on the storage device, and that they are minimal in size (and therefore do not need to be accounted for in the allocation of space in the block). We will (as usual) represent links to other vertices as arrows.

A vertex of a B-Tree looks like this. A non-leaf vertex **always** has 1 more children than the number of records it contains. Leaf vertices contain records but have no children. The records contained in the vertex are stored in the block in sequential order from smallest to largest key. We can visualize the links to the children as alternating with the stored records - the first child link points to a subtree containing keys < the first stored record, the link between the first and second stored records points to a subtree containing all keys > the first stored record and < the second stored record, and so on. The final child link points to a subtree containing keys > the last stored record.

Example: If the vertex contains 7 records, it must have 8 children.



When we search the tree for a particular record we use a generalized version of the binary search algorithm. We first look at the root vertex (i.e. we load that block) and check the records in that block to see if one of them is the one we want. If not, the record we want must be in one of the subtrees so we use the child link to determine which block to load next. We know which block to load by comparing the key for the desired record to the key values in the records stored in this block. We repeat the search process on the newly loaded block.

Like Red-Black Trees, B-Trees are balanced and therefore have $O(\log n)$ height. However, the balancing method is completely different. To describe this aspect of the data structure, we need to introduce some more restrictions on the B-Tree vertices:

Let m be the maximum number of records a vertex can hold.

Remember, this is dictated by the size of the records and the size of a storage block.

1. Every non-leaf vertex except the root must contain at least $\left\lfloor \frac{m}{2} \right\rfloor$ records and no more than m records.

2. The root can contain between 1 and m records.

B-Tree Insertion

The algorithm for adding a new record to a B-Tree starts out in familiar fashion. We follow the branches of the tree until we find the proper leaf that the new record should be added to.

If the leaf contains fewer than m records, we add the new record in its proper place in the vertex, and we are done.

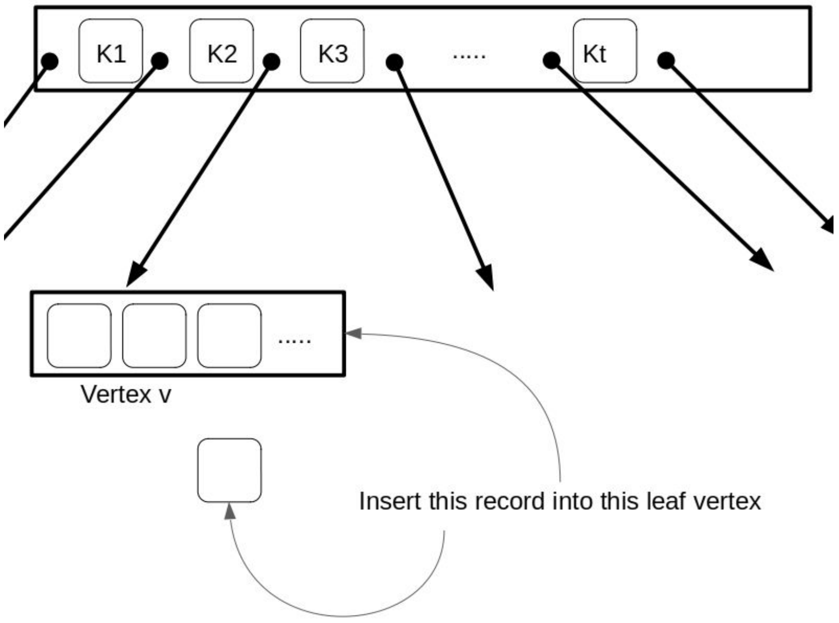
However, if the leaf already contains m records, it has no room to add another record. In this case:

1. Let $\{R_1, R_2, \ldots R_m, R_{m+1}\}$ be the set of records already in the leaf, plus the new record being added. Let these records be numbered in order by key (i.e. R_1.key < R_2.key, etc.)
2. Let $mid = \left\lceil \frac{m+1}{2} \right\rceil$
3. Create a new leaf vertex and move $\{R_1, \ldots R_{mid-1}\}$ to the new leaf
4. Keep $\{R_{mid+1}, \ldots, R_{m+1}\}$ in the original leaf
5. Add R_{mid} to the parent of the leaf, and add the new leaf as a new child of that parent.

This finishes the job unless the parent was also "full"[2]. In this case we apply exactly the same splitting operation on this vertex and push a record up to the level above. This split-and-push-one-record-up sequence can propagate all the way back to the root. If we have to split the root we do that, then create a new root containing just the "pushed-up" record from the old root.

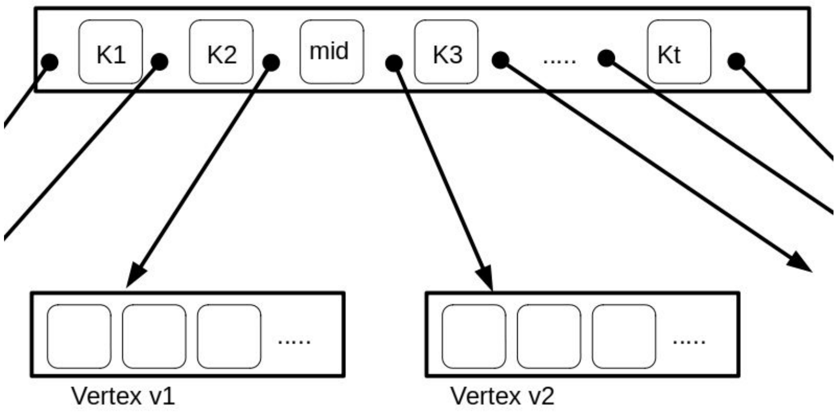
[2] i.e. it already contained *m* records

Repeating the above, but with figures!



If Vertex v has room for another record, we simply insert the new record into the vertex.

But if v already contains m records, we put the $m + 1$ records that belong in vertex v (the m already there, plus the new one) into ascending order, and let *mid* refer to the one in the middle. We promote *mid* to the parent, and split the remaining m records into two leaves, as children of the parent. The result is this:



The new record is either in vertex v1, or in vertex v2, or it is mid and new resides in the parent of these vertices. Each of v1 and v2 contains a legal number of records.

If the vertex to which we have just added mid is now over-full, we perform exactly the same split-and-push-one-record-up operation on this record. If the split-and-push-one-record-up sequence propagates all the way back to the root, the tree ends up being one level taller than before because the root gets split in two and a new root is added above it. This is the **only** way a B-tree can get taller.

B-Tree Deletion

The algorithm for removing a record is messier than the insertion algorithm. There are multiple cases and complicated sequences of operations, and we won't go into details here. The basic idea is that if removing a record makes a vertex too small[3] then we attempt to move records around between the vertex and its children to restore the tree. If that is not possible, we combine small vertices into proper sized vertices. As with the insertion algorithm these changes can propagate up the tree, potentially reaching the root and reducing the height of the tree.

[3] i.e. it now contains fewer than $\left\lfloor \frac{m}{2} \right\rfloor$ records

Performance of B-Trees

Let n be the number of records in a B-tree T. Because each internal vertex (except the root) must have at least $\left\lfloor \frac{m}{2} \right\rfloor$ records, and therefore at least $\left\lfloor \frac{m}{2} \right\rfloor + 1$ children, each level of the tree (except the level just below the root) contains at least $\left\lfloor \frac{m}{2} \right\rfloor + 1$ times as many vertices as the level above it, and each of those vertices contains at least $\left\lfloor \frac{m}{2} \right\rfloor$ records.

Since every vertex except the root must contain at least $\left\lfloor \frac{m}{2} \right\rfloor$ records, we can see that the number of vertices cannot exceed $\approx \frac{n}{\frac{m}{2}} = \frac{2n}{m}$

Let k be the height of the B-tree. There is 1 vertex at the top level (the root), at least 2 at the second level, at least $2 * (\left\lfloor \frac{m}{2} \right\rfloor + 1)$ at the third level, and then at least $2 * (\left\lfloor \frac{m}{2} \right\rfloor + 1)^{i-2}$ vertices at level i thereafter. So the number of vertices is at least

$$3 + 2 * \sum_{i=3}^{k} \left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right)^{i-2}$$

Combining these bounds on the number of vertices gives

$$3 + 2 * \sum_{i=3}^{k} \left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right)^{i-2} \quad \leq \quad \frac{2n}{m}$$

Let's ignore the "3" as an irrelevant constant[4]. This gives

[4] How convenient! It's a good thing I'm not writing a math textbook! But we know we are heading towards a O classification in which constant terms are discarded, so I'm really just dropping it early rather than late.

$$\sum_{i=3}^{k} \left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right)^{i-2} \quad \leq \quad \frac{n}{m}$$

Note that the lhs is $\geq \left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right)^{k-2}$ so we have

$$\left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right)^{k-2} \quad \leq \quad \frac{n}{m}$$

Take the log of both sides in base $\left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right)$

$$k - 2 \quad \leq \quad \log_{\left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right)} \left(\frac{n}{m} \right)$$

In other words, $k \in O\left(\log \frac{n}{m} \right)$ and since m is a constant (for this

B-tree), we conclude (finally!!!) that B-trees are guaranteed to have height in $O(\log n)$, as claimed.