B-tree

From Wikipedia, the free encyclopedia

In computer science, a **B-tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children (Comer 1979, p. 123). Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. It is commonly used in databases and filesystems.

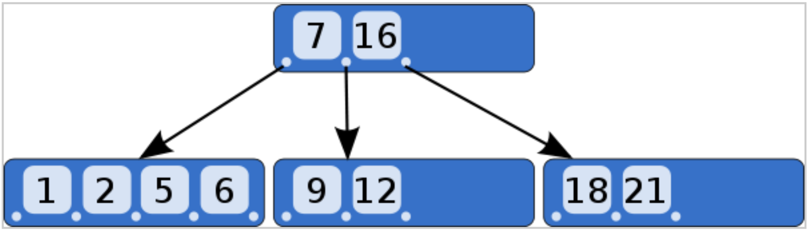
B-tree

Type	Tree	
Invented	1972	
Invented by	Rudolf Bayer, Edward M. McCreight	
Time complexity in big O notation		
	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)

Contents

- 1 Overview
  - 1.1 Variants
  - 1.2 Etymology
- 2 B-tree usage in databases
  - 2.1 Time to search a sorted file
  - 2.2 An index speeds the search
  - 2.3 Insertions and deletions
  - 2.4 Advantages of B-tree usage for databases
  - 2.5 Disadvantages of B-trees
- 3 Technical description
  - 3.1 Terminology
  - 3.2 Definition
- 4 Best case and worst case heights
- 5 Algorithms
  - 5.1 Search
  - 5.2 Insertion
  - 5.3 Deletion
    - 5.3.1 Deletion from a leaf node
    - 5.3.2 Deletion from an internal node
    - 5.3.3 Rebalancing after deletion
  - 5.4 Sequential access
  - 5.5 Initial construction
- 6 In filesystems
- 7 Variations
  - 7.1 Access concurrency
- 8 See also
- 9 Notes
- 10 References
  - 10.1 Original papers
- 11 External links

Overview



A B-tree (Bayer & McCreight 1972) of order 5 (Knuth 1998).

In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

Each internal node of a B-tree will contain a number of keys. The keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

Usually, the number of keys is chosen to vary between d and $2d$, where d is the minimum number of keys, and $d+1$ is the minimum degree or branching factor of the tree. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has $2d$ keys, then adding a key to that node can be accomplished by splitting the hypothetical $2d+1$ key node into two d key nodes and moving the key that would have been in the middle to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbor each have d keys, then a key may be deleted from the internal node by combining it with its neighbor. Deleting the key would make the internal node have $d-1$ keys; joining the neighbor would add d keys plus one more key brought down from the neighbor's parent. The result is an entirely full node of $2d$ keys.

The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-tree is sometimes described with the parameters $(d+1)$ — $(2d+1)$ or simply with the highest branching order, $(2d+1)$.

A B-tree is kept balanced by requiring that all leaf nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node farther away from the root.

B-trees have substantial advantages over alternative implementations when the time to access the data of a node greatly exceeds the time spent processing that data, because then the cost of accessing the node may be amortized over multiple operations within the node. This usually occurs when the node data are in secondary storage such as disk drives. By maximizing the number of keys within each internal node, the height of the tree decreases and the number of expensive node accesses is reduced. In addition, rebalancing of the tree occurs less often. The maximum number of child nodes depends on the information that must be stored for each child node and the size of a full disk block or an analogous size in secondary storage. While 2-3 B-trees are easier to explain, practical B-trees using secondary storage need a large number of child nodes to improve performance.

Variants

The term **B-tree** may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B+ tree and the B* tree.

- In the B+ tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access (Comer 1979, p. 129).
- The B* tree balances more neighboring internal nodes to keep the internal nodes more densely packed (Comer 1979, p. 129). This variant requires non-root nodes to be at least 2/3 full instead of 1/2 (Knuth 1998, p. 488). To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it. When both nodes are full, then the two nodes are split into three. Deleting nodes is somewhat more complex than inserting however.
- B-trees can be turned into order statistic trees to allow rapid searches for the Nth record in key order, or counting the number of records between any two records, and various other related operations.[1]

Etymology

Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972), but they did not explain what, if anything, the *B* stands for. Douglas Comer explains:

> The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees. (Comer 1979, p. 123 footnote 1)

Donald Knuth speculates on the etymology of B-trees in his May, 1980 lecture on the topic "CS144C classroom lecture about disk storage and B-trees", suggesting the "B" may have originated from Boeing or from Bayer's name.[2]

Ed McCreight answered a question on B-tree's name in 2013:

> Bayer and I were in a lunch time where we get to think a name. And we were, so, B, we were thinking… B is, you know… We were working for Boeing at the time, we couldn't use the name without talking to lawyers. So, there is a B. It has to do with balance, another B. Bayer was the senior author, who did have several years older than I am and had many more publications than I did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees."[3]

B-tree usage in databases

Time to search a sorted file

Usually, sorting and searching algorithms have been characterized by the number of comparison operations that must be performed using order notation. A binary search of a sorted table with N records, for example, can be done in roughly $\lceil \log_2 N \rceil$ comparisons. If the table had 1,000,000 records, then a specific record could be located with at most 20 comparisons: $\lceil \log_2 (1,000,000) \rceil = 20$.

Large databases have historically been kept on disk drives. The time to read a record on a disk drive far exceeds the time needed to compare keys once the record is available. The time to read a record from a disk drive involves a seek time and a rotational delay. The seek time may be 0 to 20 or more milliseconds, and the rotational delay averages about half the rotation period. For a 7200 RPM drive, the rotation period is 8.33 milliseconds. For a drive such as the Seagate ST3500320NS, the track-to-track seek time is 0.8 milliseconds and the average reading seek time is 8.5 milliseconds.[4] For simplicity, assume reading from disk takes about 10 milliseconds.

Naively, then, the time to locate one record out of a million would take 20 disk reads times 10 milliseconds per disk read, which is 0.2 seconds.

The time won't be that bad because individual records are grouped together in a disk **block**. A disk block might be 16 kilobytes. If each record is 160 bytes, then 100 records could be stored in each block. The disk read time above was actually for an entire block. Once the disk head is in position, one or more disk blocks can be read with little delay. With 100 records per block, the last 6 or so comparisons don't need to do any disk reads—the comparisons are all within the last disk block read.

To speed the search further, the first 13 to 14 comparisons (which each required a disk access) must be sped up.

An index speeds the search

A significant improvement can be made with an index. In the example above, initial disk reads narrowed the search range by a factor of two. That can be improved substantially by creating an auxiliary index that contains the first record in each disk block (sometimes called a sparse index). This auxiliary index would be 1% of the size of the original database, but it can be searched more quickly. Finding an entry in the auxiliary index would tell us which block to search in the main database; after searching the auxiliary index, we would have to search only that one block of the main database—at a cost of one more disk read. The index would hold 10,000 entries, so it would take at most 14 comparisons. Like the main database, the last 6 or so comparisons in the aux index would be on the same disk block. The index could be searched in about 8 disk reads, and the desired record could be accessed in 9 disk reads.

The trick of creating an auxiliary index can be repeated to make an auxiliary index to the auxiliary index. That would make an aux-aux index that would need only 100 entries and would fit in one disk block.

Instead of reading 14 disk blocks to find the desired record, we only need to read 3 blocks. Reading and searching the first (and only) block of the aux-aux index identifies the relevant block in aux-index. Reading and searching that aux-index block identifies the relevant block in the main database. Instead of 150 milliseconds, we need only 30 milliseconds to get the record.

The auxiliary indices have turned the search problem from a binary search requiring roughly $\log_2 N$ disk reads to one requiring only $\log_b N$ disk reads where b is the blocking factor (the number of entries per block: $b = 100$ entries per block; $\log_b 1{,}000{,}000 = 3$ reads).

In practice, if the main database is being frequently searched, the aux-aux index and much of the aux index may reside in a disk cache, so they would not incur a disk read.

Insertions and deletions

If the database does not change, then compiling the index is simple to do, and the index need never be changed. If there are changes, then managing the database and its index becomes more complicated.

Deleting records from a database is relatively easy. The index can stay the same, and the record can just be marked as deleted. The database stays in sorted order. If there is a large number of deletions, then the searching and storage become less efficient.

Insertions can be very slow in a sorted sequential file because room for the inserted record must be made. Inserting a record before the first record in the file requires shifting all of the records down one. Such an operation is just too expensive to be practical. One solution is to leave some space available to be used for insertions. Instead of densely storing all the records in a block, the block can have some free space to allow for subsequent insertions. Those records would be marked as if they were "deleted" records.

Both insertions and deletions are fast as long as space is available on a block. If an insertion won't fit on the block, then some free space on some nearby block must be found and the auxiliary indices adjusted. The hope is that enough space is nearby such that a lot of blocks do not need to be reorganized. Alternatively, some out-of-sequence disk blocks may be used.

Advantages of B-tree usage for databases

The B-tree uses all of the ideas described above. In particular, a B-tree:

- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with a recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.

Disadvantages of B-trees

- maximum key length cannot be changed without completely rebuilding the database. This led to many database systems truncating full human names to 70 characters.

(Other implementations of associative array, such as a ternary search tree or a separate-chaining hash table, dynamically adapt to arbitrarily long key lengths).

Technical description

Terminology

The literature on B-trees is not uniform in its terminology (Folk & Zoellick 1992, p. 362).

Bayer & McCreight (1972), Comer (1979), and others define the **order** of B-tree as the minimum number of keys in a non-root node. Folk & Zoellick (1992) points out that terminology is ambiguous because the maximum number of keys is not clear. An order 3 B-tree might hold a maximum of 6 keys or a maximum of 7 keys. Knuth (1998, p. 483) avoids the problem by defining the **order** to be maximum number of children (which is one more than the maximum number of keys).

The term **leaf** is also inconsistent. Bayer & McCreight (1972) considered the leaf level to be the lowest level of keys, but Knuth considered the leaf level to be one level below the lowest keys (Folk & Zoellick 1992, p. 363). There are many possible implementation choices. In some designs, the leaves may hold the entire data record; in other designs, the leaves may only hold pointers to the data record. Those choices are not fundamental to the idea of a B-tree.[5]

There are also unfortunate choices like using the variable *k* to represent the number of children when *k* could be confused with the number of keys.

For simplicity, most authors assume there are a fixed number of keys that fit in a node. The basic assumption is the key size is fixed and the node size is fixed. In practice, variable length keys may be employed (Folk & Zoellick 1992, p. 379).

Definition

According to Knuth's definition, a B-tree of order *m* is a tree which satisfies the following properties:

1. Every node has at most *m* children.
2. Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with *k* children contains $k-1$ keys.
5. All leaves appear in the same level

Each internal node’s keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

Internal nodes

Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of *U* children and a **minimum** of *L* children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between $L-1$ and $U-1$). *U* must be either $2L$ or $2L-1$; therefore each internal node is at least half full. The relationship between *U* and *L* implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there’s room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

The root node

The root node’s number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than $L-1$ elements in the entire tree, the root will be the only node in the tree with no children at all.

Leaf nodes

Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers.

A B-tree of depth $n+1$ can hold about *U* times as many items as a B-tree of depth *n*, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

Some balanced trees store values only at leaf nodes, and use different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree, and may use the same structure for all nodes. However, since leaf nodes never have children, the B-trees benefit from improved performance if they use a specialized structure.

Best case and worst case heights

Let *h* be the height of the classic B-tree. Let $n > 0$ be the number of entries in the tree.[6] Let *m* be the maximum number of children a node can have. Each node can have at most $m-1$ keys.

It can be shown (by induction for example) that a B-tree of height h with all its nodes completely filled has $n=m^{h+1}-1$ entries. Hence, the best case height of a B-tree is:

$$\lceil \log_m(n+1) \rceil - 1$$

Let d be the minimum number of children an internal (non-root) node can have. For an ordinary B-tree, $d=\lceil m/2 \rceil$.

Comer (1979, p. 127) and Cormen et al. (2001, pp. 383–384) give the worst case height of a B-tree (where the root node is considered to have height 0) as

$$h \le \left\lfloor \log_d\left(\frac{n+1}{2}\right) \right\rfloor .$$

Algorithms

Search

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search reduces its field of view to the child pointer (subtree) whose range includes the search value. A subtree's range is defined by the values, or keys, contained in its parent node. These limiting values are also known as separation values.

Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.

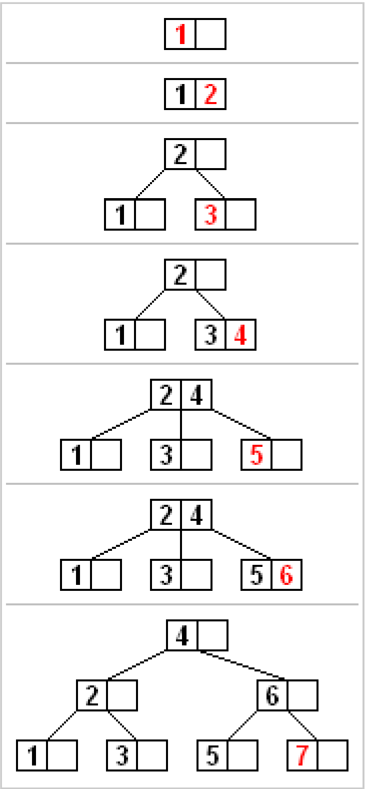
Insertion

All insertions start at a leaf node. To insert a new element, search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1. If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
2. Otherwise the node is full, evenly split it into two nodes so:
    1. A single median is chosen from among the leaf's elements and the new element.
    2. Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
    3. The separation value is inserted in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is $U-1$. When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number $U-1$ of elements into two legal nodes. If this number is odd, then $U=2L$ and one of the new nodes contains $(U-2)/2 = L-1$ elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If $U-1$ is even, then $U=2L-1$, so there are $2L-2$ elements in the node. Half of this number is $L-1$, which is the minimum number of elements allowed per node.

An improved algorithm supports a single pass down the tree from the root to the node where the insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage. However, to use this improved algorithm, we must be able to send one element to the parent and split the remaining $U-2$ elements into two legal nodes, without adding a new element. This requires $U = 2L$ rather than $U = 2L-1$, which accounts for why some textbooks impose this requirement in defining B-trees.



A B Tree insertion example with each iteration. The nodes of this B tree have at most 3 children (Knuth order 3).

Deletion

There are two popular strategies for deletion from a B-tree.

1. Locate and delete the item, then restructure the tree to retain its invariants, **OR**
2. Do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

The algorithm below uses the former strategy.

There are two special cases to consider when deleting an element:

1. The element in an internal node is a separator for its child nodes
2. Deleting an element may put its node under the minimum number of elements and children

The procedures for these cases are in order below.

Deletion from a leaf node

1. Search for the value to delete.
2. If the value is in a leaf node, simply delete it from the node.
3. If underflow happens, rebalance the tree as described in section "Rebalancing after deletion" below.

Deletion from an internal node

Each element in an internal node acts as a separation value for two subtrees, therefore we need to find a replacement for separation. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is still greater than the separator. Both of those elements are in leaf nodes, and either one can be the new separator for the two subtrees. Algorithmically described below:

1. Choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
2. The previous step deleted an element (the new separator) from a leaf node. If that leaf node is now deficient (has fewer than the required number of nodes), then rebalance the tree starting from the leaf node.

Rebalancing after deletion

Rebalancing starts from a leaf and proceeds toward the root until the tree is balanced. If deleting an element from a node has brought it under the minimum size, then some elements must be redistributed to bring all nodes up to the minimum. Usually, the redistribution involves moving an element from a sibling node that has more than the minimum number of nodes. That redistribution operation is called a **rotation**. If no sibling can spare an element, then the deficient node must be **merged** with a sibling. The merge causes the parent to lose a separator element, so the parent may become deficient and need rebalancing. The merging and rebalancing may continue all the way to the root. Since the minimum element count doesn't apply to the root, making the root be the only deficient node is not a problem. The algorithm to rebalance the tree is as follows:

- If the deficient node's right sibling exists and has more than the minimum number of elements, then rotate left
    1. Copy the separator from the parent to the end of the deficient node (the separator moves down; the deficient node now has the minimum number of elements)
    2. Replace the separator in the parent with the first element of the right sibling (right sibling loses one node but still has at least the minimum number of elements)
    3. The tree is now balanced
- Otherwise, if the deficient node's left sibling exists and has more than the minimum number of elements, then rotate right
    1. Copy the separator from the parent to the start of the deficient node (the separator moves down; deficient node now has the minimum number of elements)
    2. Replace the separator in the parent with the last element of the left sibling (left sibling loses one node but still has at least the minimum number of elements)
    3. The tree is now balanced
- Otherwise, if both immediate siblings have only the minimum number of elements, then merge with a sibling sandwiching their separator taken off from their parent
    1. Copy the separator to the end of the left node (the left node may be the deficient node or it may be the sibling with the minimum number of elements)
    2. Move all elements from the right node to the left node (the left node now has the maximum number of elements, and the right node – empty)
    3. Remove the separator from the parent along with its empty right child (the parent loses an element)
        - If the parent is the root and now has no elements, then free it and make the merged node the new root (tree becomes shallower)
        - Otherwise, if the parent has fewer than the required number of elements, then rebalance the parent

Note: The rebalancing operations are different for B+ trees (e.g., rotation is different because parent has copy of the key) and B*-tree (e.g., three siblings are merged into two siblings).

Sequential access

While freshly loaded databases tend to have good sequential behavior, this behavior becomes increasingly difficult to maintain as a database grows, resulting in more random I/O and performance challenges.[7]

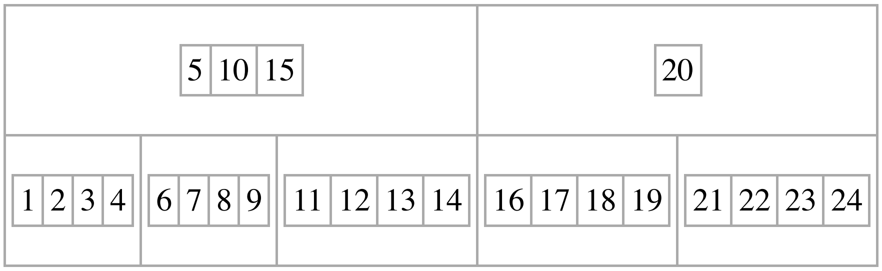
Initial construction

In applications, it is frequently useful to build a B-tree to represent a large existing collection of data and then update it incrementally using standard B-tree operations. In this case, the most efficient way to construct the initial B-tree is not to insert every element in the initial collection successively, but instead to construct the initial set of leaf nodes directly from the input, then build the internal nodes from these. This approach to B-tree construction is called bulkloading. Initially, every leaf but the last one has one extra element, which will be used to build the internal nodes.

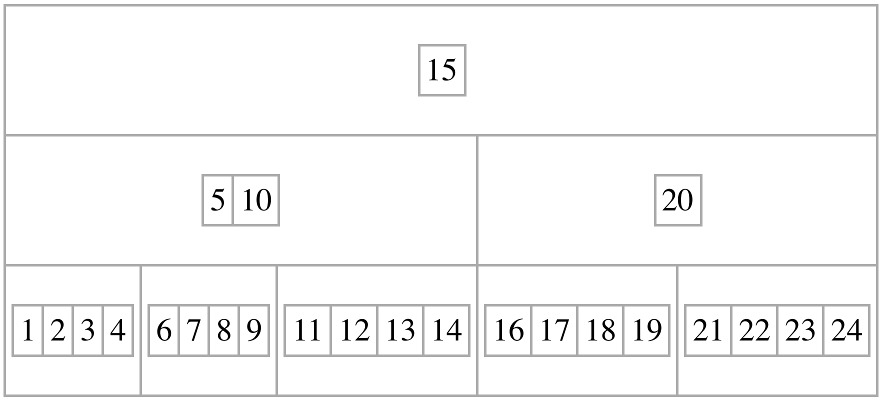
For example, if the leaf nodes have maximum size 4 and the initial collection is the integers 1 through 24, we would initially construct 4 leaf nodes containing 5 values each and 1 which contains 4 values:

1	2	3	4	5		6	7	8	9	10		11	12	13	14	15		16	17	18	19	20		21	22	23	24

We build the next level up from the leaves by taking the last element from each leaf node except the last one. Again, each node except the last will contain one extra value. In the example, suppose the internal nodes contain at most 2 values (3 child pointers). Then the next level up of internal nodes would be:



This process is continued until we reach a level with only one node and it is not overfilled. In the example only the root level remains:



In filesystems

Most modern filesystems use B-trees (or § Variants); alternatives such as extendible hashing are less common.[8]

In addition to its use in databases, the B-tree is also used in filesystems to allow quick random access to an arbitrary block in a particular file. The basic problem is turning the file block i address into a disk block (or perhaps to a cylinder-head-sector) address.

Some operating systems require the user to allocate the maximum size of the file when the file is created. The file can then be allocated as contiguous disk blocks. When converting to a disk block the operating system just adds the file block address to the starting disk block of the file. The scheme is simple, but the file cannot exceed its created size.

Other operating systems allow a file to grow. The resulting disk blocks may not be contiguous, so mapping logical blocks to physical blocks is more involved.

MS-DOS, for example, used a simple File Allocation Table (FAT). The FAT has an entry for each disk block,[note 1] and that entry identifies whether its block is used by a file and if so, which block (if any) is the next disk block of the same file. So, the allocation of each file is represented as a linked list in the table. In order to find the disk address of file block i, the operating system (or disk utility) must sequentially follow the file's linked list in the FAT. Worse, to find a free disk block, it must sequentially scan the FAT. For MS-DOS, that was not a huge penalty because the disks and files were small and the FAT had few entries and relatively short file chains. In the FAT12 filesystem (used on floppy disks and early hard disks), there were no more than 4,080 [note 2] entries, and the FAT would usually be resident in memory. As disks got bigger, the FAT architecture began to confront penalties. On a large disk using FAT, it may be necessary to perform disk reads to learn the disk location of a file block to be read or written.

TOPS-20 (and possibly TENEX) used a 0 to 2 level tree that has similarities to a B-tree. A disk block was 512 36-bit words. If the file fit in a 512 (2^9) word block, then the file directory would point to that physical disk block. If the file fit in 2^{18} words, then the directory would point to an aux index; the 512 words of that index would either be NULL (the block isn't allocated) or point to the physical address of the block. If the file fit in 2^{27} words, then the directory would point to a block holding an aux-aux index; each entry would either be NULL or point to an aux index. Consequently, the physical disk block for a 2^{27} word file could be located in two disk reads and read on the third.

Apple's filesystem HFS+, Microsoft's NTFS,[9] AIX (jfs2) and some Linux filesystems, such as btrfs and Ext4, use B-trees.

B^*-trees are used in the HFS and Reiser4 file systems.

Variations

Access concurrency

Lehman and Yao[10] showed that all the read locks could be avoided (and thus concurrent access greatly improved) by linking the tree blocks at each level together with a "next" pointer. This results in a tree structure where both insertion and search operations descend from the root to the leaf. Write locks are only required as a tree block is modified. This maximizes access concurrency by multiple users, an important consideration for databases and/or other B-tree based ISAM storage methods. The cost associated with this improvement is that empty pages cannot be removed from the btree during normal operations. (However, see [11] for various strategies to implement node merging, and source code at.[12])

United States Patent 5283894, granted in 1994, appears to show a way to use a 'Meta Access Method' [13] to allow concurrent B+ tree access and modification without locks. The technique accesses the tree 'upwards' for both searches and updates by means of additional in-memory indexes that point at the blocks in each level in the block cache. No reorganization for deletes is needed and there are no 'next' pointers in each block as in Lehman and Yao.

See also

- B+tree
- R-tree
- 2–3 tree
- 2–3–4 tree

Notes

1. For FAT, what is called a "disk block" here is what the FAT documentation calls a "cluster", which is fixed-size group of one or more contiguous whole physical disk sectors. For the purposes of this discussion, a cluster has no significant difference from a physical sector.
2. Two of these were reserved for special purposes, so only 4078 could actually represent disk blocks (clusters).

References

1. Counted B-Trees (http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html), retrieved 2010-01-25
2. Knuth's video lectures from Stanford (http://scpd.stanford.edu/knuth/index.jsp)
3. Video of the talk (http://vimeo.com/73481096) at CPM 2013 (24th Annual Symposium on Combinatorial Pattern Matching, Bad Herrenalb, Germany, June 17–19, 2013), retrieved 2014-01-17; see question asked by Martin Farach-Colton
4. Seagate Technology LLC, Product Manual: Barracuda ES.2 Serial ATA, Rev. F., publication 100468393, 2008 [1] (http://www.seagate.com/staticfiles/support/disc/manuals/NL35%20Series%20&%20BC%20ES%20Series/Barracuda%20ES.2%20Series/100468393f.pdf), page 6
5. Bayer & McCreight (1972) avoided the issue by saying an index element is a (physically adjacent) pair of (x, a) where x is the key, and a is some associated information. The associated information might be a pointer to a record or records in a random access, but what it was didn't really matter. Bayer & McCreight (1972) states, "For this paper the associated information is of no further interest."
6. If n is zero, then no root node is needed, so the height of an empty tree is not well defined.
7. "Cache Oblivious B-trees". State University of New York (SUNY) at Stony Brook. Retrieved 2011-01-17.
8. Mikuláš Patocka. "Design and Implementation of the Spad Filesystem" (http://artax.karlin.mff.cuni.cz/~mikulas/spadfs/THESIS.PDF). "Table 4.1: Directory organization in filesystems". 2006.
9. Mark Russinovich. "Inside Win2K NTFS, Part 1". Microsoft Developer Network. Archived from the original on 13 April 2008. Retrieved 2008-04-18.
10. "Efficient locking for concurrent operations on B-trees". Portal.acm.org. doi:10.1145/319628.319663. Retrieved 2012-06-28.
11. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA232287&Location=U2&doc=GetTRDoc.pdf
12. "Downloads - high-concurrency-btree - High Concurrency B-Tree code in C - GitHub Project Hosting". Retrieved 2014-01-27.
13. Lockless Concurrent B+Tree (http://www.freepatentsonline.com/5283894.html)

General

- Bayer, R.; McCreight, E. (1972), "Organization and Maintenance of Large Ordered Indexes" (PDF), *Acta Informatica*, **1** (3): 173–189, doi:10.1007/bf00288683
- Comer, Douglas (June 1979), "The Ubiquitous B-Tree", *Computing Surveys*, **11** (2): 123–137, doi:10.1145/356770.356776, ISSN 0360-0300.
- Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2001), *Introduction to Algorithms* (Second ed.), MIT Press and McGraw-Hill, pp. 434–454, ISBN 0-262-03293-7. Chapter 18: B-Trees.
- Folk, Michael J.; Zoellick, Bill (1992), *File Structures* (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4
- Knuth, Donald (1998), *Sorting and Searching*, The Art of Computer Programming, Volume 3 (Second ed.), Addison-Wesley, ISBN 0-201-89685-0. Section 6.2.4: Multiway Trees, pp. 481–491. Also, pp. 476–477 of section 6.2.3 (Balanced Trees) discusses 2-3 trees.

Original papers

- Bayer, Rudolf; McCreight, E. (July 1970), *Organization and Maintenance of Large Ordered Indices*, Mathematical and Information Sciences Report No. 20, Boeing Scientific Research Laboratories.
- Bayer, Rudolf (1971), *Binary B-Trees for Virtual Memory*, Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California.

External links

- B-tree lecture (http://www.youtube.com/watch?v=I22wEC1tTGo) by David Scot Taylor, SJSU
- B-Tree animation applet (http://slady.net/java/bt/view.php) by slady
- B-tree and UB-tree on Scholarpedia (http://www.scholarpedia.org/article/B-tree_and_UB-tree) Curator: Dr Rudolf Bayer

- B-Trees: Balanced Tree Data Structures (http://www.bluerwhite.org/btree)
- NIST's Dictionary of Algorithms and Data Structures: B-tree (https://xlinux.nist.gov/dads/HTML/btree.html)
- B-Tree Tutorial (http://cis.stvincent.edu/html/tutorials/swd/btree/btree.html)
- The InfinityDB BTree implementation (http://www.boilerbay.com/infinitydb/TheDesignOfTheInfinityDatabaseEngine.htm)
- Cache Oblivious B(+)-trees (http://supertech.csail.mit.edu/cacheObliviousBTree.html)
- Dictionary of Algorithms and Data Structures entry for B*-tree (https://xlinux.nist.gov/dads/HTML/bstartree.html)
- Open Data Structures - Section 14.2 - B-Trees (http://opendatastructures.org/versions/edition-0.1g/ods-python/14_2_B_Trees.html)
- Counted B-Trees (http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html)
- B-Tree .Net, a modern, virtualized RAM & Disk implementation (http://sop.codeplex.com)

Retrieved from "https://en.wikipedia.org/w/index.php?title=B-tree&oldid=747461265"

Categories: 1971 introductions | B-tree | Database index techniques

---

- This page was last modified on 2 November 2016, at 14:29.
-