Analysis of B-tree data structure and its usage in computer forensics

Conference Paper · January 2010

CITATIONS
10

READS
7,852

2 authors:

Petra Grd
University of Zagreb
31 PUBLICATIONS **145** CITATIONS
SEE PROFILE

Miroslav Bača
University of Zagreb
108 PUBLICATIONS **484** CITATIONS
SEE PROFILE

Analysis of B-tree data structure and its usage in computer forensics

Petra Koruga, Miroslav Bača
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
{petra.koruga, miroslav.baca}@foi.hr

Abstract. *The idea behind this article is to give an overview of B-tree data structure and show the connection between B-tree indexing technique and computer forensics. B-tree is a fast data indexing method that organizes indexes into a multi-level set of nodes, where each node contains indexed data. This technique is most commonly used in databases and file systems where it is important to retrieve records stored in a file when data is to large to fit in main memory. In that case, B-trees are used to reduce the number of disk accesses.*

Keywords. B-tree, computer forensic, indexing, filesystem

1 Introduction

When data is to large to fit in main memory, number of disk accesses is important. The time required to access and retrieve a word from high-speed memory is a few microseconds at most and the time required to locate a particular record on a disk is measured in milliseconds. Hence the time required for a single access is thousands of times greater for external retrieval than for internal retrieval [1]. The goal in external searching is to minimize the number of disk accesses, since each access takes so long compared to internal computation. The idea is to make the height of the tree as small as possible. That is where B-trees become useful. They were named by R. Bayer and E. McCreight, who were the first to consider the use of multiway balanced trees for external searching [2]. It allows to keep both primary data records, and search tree structure, out on disk. Only a few nodes from the tree and a single data record ever need be in primary memory [3].

2 B-tree

B-trees, which are balanced search trees specifically designed to be stored on magnetic disks. Because magnetic disks operate much more slowly than random access memory, we measure the performance of B-trees not only by how much computing time the dynamic-set operations consume but also by how many disk accesses are performed. For each B-tree operation, the number of disk accesses increases with the height of the B-tree, which is kept low by the B-tree operations[5].

2.1 Basics of B-tree

The B-tree was created by Rudolf Bayer and Ed McCreight while they were working at Boeing Research Lab. They haven't explained what B stands for, the most common belief is that B stands for balanced, other beliefs are that it stands for Bayer, Boeing, broad or bushy[4]. In computer science, a B-tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic amortized time. The B-tree is a generalization of a binary search tree in that more than two paths diverge from a single node[4]. A B-tree of order m is an m-way search tree in which[1]:

- All leaves are on the same level

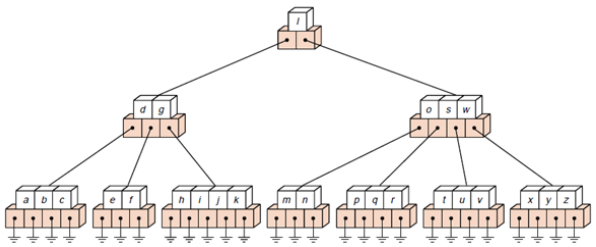


Figure 1: Example of a B-tree[1]

- All internal nodes except the root have at most m nonempty children, and at least m/2 nonempty children
- The number of keys in each internal node is one less than the number of its nonempty children, and these keys partition the keys in the children in the fashion of a search tree
- The root has at most m children, but may have as few as 2 if it is not a leaf, or none if the tree consists of the root alone

2.1.1 Height

The question of height of a b-tree is important because maximum height of a b-tree gives an upper bound on number of disk accesses[5]. If a B-tree has height h, the root contains at least one key and all other nodes contain at least t - 1 keys. So there are at least 2 nodes at depth 1, at least 2t nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h there are at least $2t^{h-1}$ nodes. Thus, the number n of keys satisfies the inequality $t^h \leq (n + 1)/2$. Taking a base-t algorithm of both sides gives: $h \leq \log_t ((n+1)/2)$.

2.1.2 Example

The following example is of a B-tree of order 5. This means that all internal nodes have at least ceil(5 / 2) = ceil(2.5) = 3 children (and at least 2 keys). Of course, the maximum number of children that a node can have is 5 (4 is the maximum number of keys). Each leaf node must contain at least 2 keys. In practice B-trees usually have orders a lot bigger than 5[1].

2.2 Operations on B-trees

2.2.1 Initial Construction

First operation on a B-tree is its initial construction. That construction will be explained with pseudocode [5]:

```
B-Tree-Create(T)
  x <- Allocate-Node()
  leaf[x] <- TRUE
  n[x] <- 0
  Disk-Write(x)
  root[T] <- x
```

The B-Tree-Create(T) creates an empty b-tree by allocating a new root node that has no keys and is a leaf node.

2.2.2 Insertion

B-trees grow at the root, they are not allowed to grow at their leaves because all leaves must be at the same level. General insertion method (pseudocode) [5]:

```
B-TREE-SPLIT-CHILD(x, i, y)
  z <- ALLOCATE-NODE()
  leaf[z] <- leaf[y]
  n[z] <- t - 1
  for j <- 1 to t - 1
do keyj[z]<- keyj+t[y]
  if not leaf [y]
  then for j<- 1 to t
do cj[z] <- cj+t[y]
  n[y] <- t - 1
  for j<- n[x] + 1 downto i + 1
do cj+1[x] <- cj [x]
  ci+1[x] <- z
  for j<- n[x] downto i
do keyj+1[x] <- keyj[x]
  keyi[x] <- keyt[y]
  n[x] <- n[x] + 1
  DISK-WRITE(y)
  DISK-WRITE(z)
  DISK-WRITE(x)
```

2.2.3 Search-data retrieval

Searching in B-trees is actualy deciding between n[x] +1 choices. Pseudocode[5]:

```
B-TREE-SEARCH(x, k)
  i <- 1
  while i <= n[x] and k > keyi[x]
```

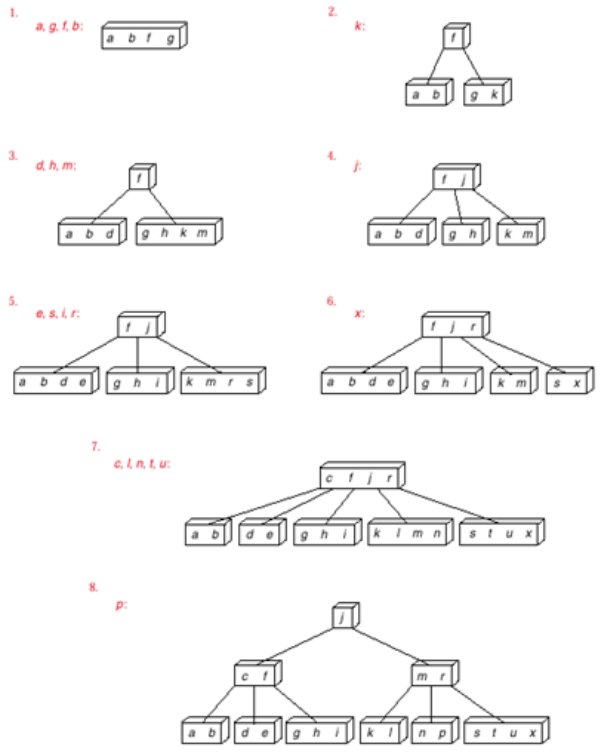


Figure 2: Insertion into a B-tree[1]

do i <- i + 1
if i <= n[x] and k = keyi[x]
then return (x, i)
if leaf [x]
then return NIL
else DISK-READ(ci[x])
return B-TREE-SEARCH(ci[x], k)

After finding the value greater than or equal to the desired value, the child pointer to the left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. The search is terminated when the desired node is found [1].

2.2.4 Deletion

Deletion from a B-tree is a little bit more complicated then insertion. B-tree after deletion must have the same properties that define a B-tree. Especially property that the path to where the key is to be deleted has the minimum number of keys. Deletion can have several cases [5]:

- If the key k is in node x and x is a leaf, key k from x has to be deleted
- If the key k is in node x and x is an internal node:
  - If the child y that precedes k in node x has at least t keys, then the predecessor k' of k in the subtree rooted at y has to be found. Recursively k' has to be deleted, and replaced by k' in x
  - Symmetrically, if the child z that follows k in node x has at least t keys, then the successor k' of k in the subtree rooted at z has to be found. Recursively k has to be deleted, and replaced by k' in x
  - Otherwise, if both y and z have only t - 1 keys, k and all of z have to be merged into y, so that x loses both k and the pointer to z, and y now contains 2t - 1 keys. Then, z has to be freed and k recursively deleted from y
- If the key k is not present in internal node x, the root ci[x] of the appropriate subtree that must contain k has to be determined, if k is in the tree at all. If ci[x] has only t - 1 keys, step 1 or 2 is executed as necessary. Then recursing on the appropriate child of x has to be done.

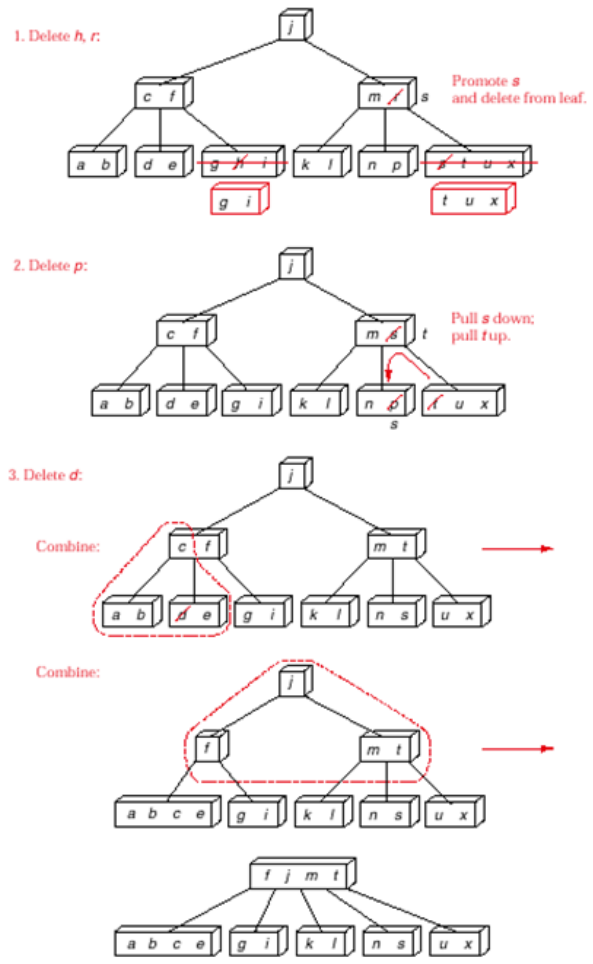


Figure 3: Deletion in a B-tree[1]

- If ci[x] has only t - 1 keys but has an immediate sibling with at least t keys, ci[x] is given an extra key by moving a key from x down into ci[x], moving a key from ci[x]'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into ci[x]

- If ci[x] and both of ci[x]'s immediate siblings have t - 1 keys, ci[x] is merged with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node

2.3 Usage of B-trees

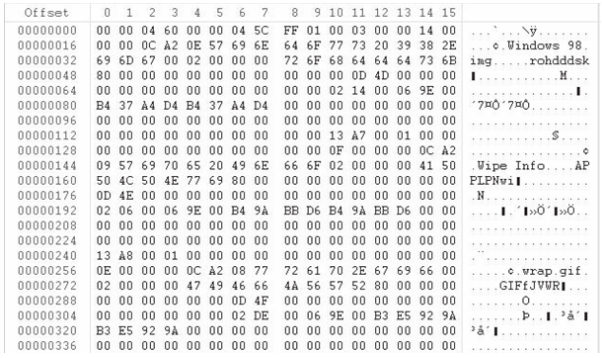
2.3.1 Databases

A database is a collection of data, organized so that supports updating, retrieving, and managing the data. The data can be anything, from names, addresses, pictures, and numbers. Databases are used everyday, a University might maintain a database of students and their grades, university employees or similar.

To be useful and usable, databases must support operations, such as retrieval, deletion and insertion of data. Databases are usually large and cannot be maintained entirely in memory, b-trees are often used to index the data and to provide fast access. Searching an unindexed and unsorted database containing n key values will have a worst case running time of O(n). If the same data is indexed with a B-Tree, the same search operation will run in O(log n) [6]. To perform a search for a single key on a set of one milion keys (1,000,000), a linear search will require at most 1,000,000 comparisons. If the same data is indexed with a b-tree of minimum degree 10, 114 comparisons will be required in the worst case[6].

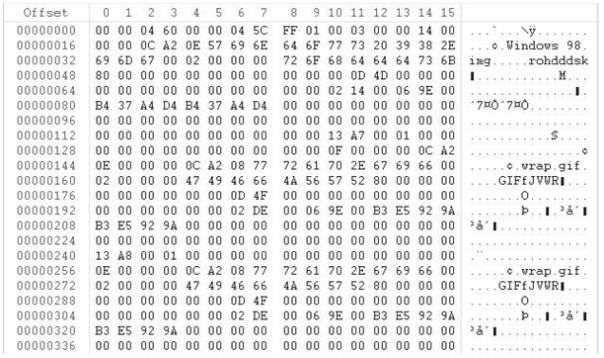
2.3.2 Filesystem

B-tree structures are also used in file systems. Directories in NTFS are indexed to make finding a specific entry in them faster. They are stored in a B-tree in alphabetical order. That takes a little more time when adding files to an NTFS directory, however it takes less time when using a directory. There are two NTFS system attributes that describe the B-Tree contents: INDEX_ROOT and INDEX_ALLOCATION. The INDEX_ROOT value is one or more "Index Entry" structures that each describe a file or directory. The "Index Entry" structure contains a copy of the FILE_NAME attribute for the file or sub-directory. For small directories, INDEX_ALLOCATION attribute will not exist and all information will be saved in the INDEX_ROOT structure. The content of this attribute is one or more "Index Buffers". Each "Index Buffer" contains one or more "Index Entry" structures, which are the same ones found in the INDEX_ROOT[7]. The benefit of using B-tree structures is evident when NTFS enumerates files in a large folder. The



Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
00000000	00	00	04	60	00	00	04	5C	FF	01	00	03	00	00	14	00	...`...\ÿ.....
00000016	00	00	0C	A2	0E	57	69	6E	64	6F	77	73	20	39	38	2E	...¢.Windows 98.
00000032	69	6D	67	00	02	00	00	00	72	6F	68	64	64	64	73	6B	img.....rohdddsk
00000048	80	00	00	00	00	00	00	00	00	00	00	0D	4D	00	00	00	▌...........M...
00000064	00	00	00	00	00	00	00	00	00	00	02	14	00	06	9E	00	..............▌.
00000080	B4	37	A4	D4	B4	37	A4	D4	00	00	00	00	00	00	00	00	´7¤Ô´7¤Ô........
00000096	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000112	00	00	00	00	00	00	00	00	00	00	13	A7	00	01	00	00	...........$....
00000128	00	00	00	00	00	00	00	00	00	00	0F	00	00	00	0C	A2	..............¢
00000144	09	57	69	70	65	20	49	6E	66	6F	02	00	00	00	41	50	.Wipe Info....AP
00000160	50	4C	50	4E	77	69	80	00	00	00	00	00	00	00	00	00	PLPNwi▌.........
00000176	0D	4E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.N..............
00000192	02	06	00	06	9E	00	B4	9A	BB	D6	B4	9A	BB	D6	00	00	....▌.´▌»Ö´▌»Ö..
00000208	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000224	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000240	13	A8	00	01	00	00	00	00	00	00	00	00	00	00	00	00	.¨..............
00000256	0E	00	00	00	0C	A2	08	77	72	61	70	2E	67	69	66	00	.....¢.wrap.gif.
00000272	02	00	00	00	47	49	46	66	4A	56	57	52	80	00	00	00	....GIFfJVWR▌...
00000288	00	00	00	00	00	00	0D	4F	00	00	00	00	00	00	00	00	.......O........
00000304	00	00	00	00	00	00	02	DE	00	06	9E	00	B3	E5	92	9A	.......Þ..▌.³å´▌
00000320	B3	E5	92	9A	00	00	00	00	00	00	00	00	00	00	00	00	³å´▌............
00000336	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Figure 4: Catalog B-tree leaf node containing file records[9]



Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
00000000	00	00	04	60	00	00	04	5C	FF	01	00	03	00	00	14	00	...`...\ÿ.....
00000016	00	00	0C	A2	0E	57	69	6E	64	6F	77	73	20	39	38	2E	...¢.Windows 98.
00000032	69	6D	67	00	02	00	00	00	72	6F	68	64	64	64	73	6B	img.....rohdddsk
00000048	80	00	00	00	00	00	00	00	00	00	00	0D	4D	00	00	00	▌...........M...
00000064	00	00	00	00	00	00	00	00	00	00	02	14	00	06	9E	00	..............▌.
00000080	B4	37	A4	D4	B4	37	A4	D4	00	00	00	00	00	00	00	00	´7¤Ô´7¤Ô........
00000096	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000112	00	00	00	00	00	00	00	00	00	00	13	A7	00	01	00	00	...........$....
00000128	00	00	00	00	00	00	00	00	00	00	0F	00	00	00	0C	A2	..............¢
00000144	0E	00	00	00	0C	A2	08	77	72	61	70	2E	67	69	66	00	.....¢.wrap.gif.
00000160	02	00	00	00	47	49	46	66	4A	56	57	52	80	00	00	00	....GIFfJVWR▌...
00000176	00	00	00	00	00	00	0D	4F	00	00	00	00	00	00	00	00	.......O........
00000192	00	00	00	00	00	00	02	DE	00	06	9E	00	B3	E5	92	9A	.......Þ..▌.³å´▌
00000208	B3	E5	92	9A	00	00	00	00	00	00	00	00	00	00	00	00	³å´▌............
00000224	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000240	13	A8	00	01	00	00	00	00	00	00	00	00	00	00	00	00	.¨..............
00000256	0E	00	00	00	0C	A2	08	77	72	61	70	2E	67	69	66	00	.....¢.wrap.gif.
00000272	02	00	00	00	47	49	46	66	4A	56	57	52	80	00	00	00	....GIFfJVWR▌...
00000288	00	00	00	00	00	00	0D	4F	00	00	00	00	00	00	00	00	.......O........
00000304	00	00	00	00	00	00	02	DE	00	06	9E	00	B3	E5	92	9A	.......Þ..▌.³å´▌
00000320	B3	E5	92	9A	00	00	00	00	00	00	00	00	00	00	00	00	³å´▌............
00000336	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Figure 5: catalog B-tree leaf node containing file records after deletion of Wipe Info[9]

B-tree structure allows NTFS to group, or index, similar file names and then search only the group that contains the file, minimizing the number of disk accesses needed to find a particular file, especially for large folders. Because of the B-tree structure, NTFS outperforms FAT for large folders because FAT must scan all file names in a large folder before listing all of the files[8].

3 Application in computer forensics

On a FAT32 file system, files are deleted by replacing the first character of the filename with the hex byte E5. The file's clusters are flagged as available, and the E5 entry in the directory still contains the (changed) name, attribute flags, date and time stamps, and logical size. On an NTFS system, the entry is un-indexed from the MFT[9].

Filenames and information on type and creator codes are visible on Figure 4[9]:

- x00 ulong fLink - Forward link to next node on the same level
- x04 ulong bLink - Backward link to previous node
- x08 uchar node - Type FF=leaf node, 00=index node, 01=B-tree Header node, 02=2nd VBM
- x09 char level - Level of this node (1=leaf)
- x0A uint numRecs - Number of records in this node

It is visible from the Figure 5 that the forward link to the next node is 00 00 04 60 (x00-x03), the backward link to the previous node is 00 00 04 5c (x04-x07), node type is FF (x08) which means that the node in question is leaf node, that the node is at level one (x09). At the position x0A is visible that there are three records in current node. At the right side of the picture names of these files are visible: Windows 98.img, Wipe Info, wrap.gif. After deletion of the file Wipe Info the number of records (x0A) is decreased from 3 to 2, and the second entry (Wipe Info) is replaced by the third entry which is now the second entry.

Although the leaf node entry may be physically overwritten, other instances of the node data may still exist in unallocated space, in index nodes, and in nodes that have been removed from the tree at a higher level. If all the files in a node are deleted because their common parent (directory) has been deleted, it is not unusual to see "pruned" nodes with all of the records intact[9]. Within each file entry in the B-tree are numerous bit fields, pointers, keys, and data values that include important things like creation and modification dates, file ID numbers and locations for the data blocks that make up the file[9]. Small folders reside whole within the MFT record, while large folders have a b-tree index structure to other data blocks. A person could search unallocated space for file entries in leaf nodes, but this presumes that a person knows what he/she is looking for, as in a filename or attribute

data. If that is not the case, places that are likely to have what wants to be found, need to be searched and such places are leaf nodes in the B-tree[9].

4 Conclusion

Small folders reside whole within the MFT record, while large folders have a b-tree index structure to other data blocks. A person could search unallocated space for file entries in leaf nodes, but this presumes that a person knows what he/she is looking for, as in a filename or attribute data. If that is not the case, places that are likely to have what wants to be found, need to be searched and such places are leaf nodes in the B-tree[9]. B-trees could be used in data retrieval, but the main problem is that the B-tree is reorganized after files or directories are deleted, overwriting the entries for the deleted items. There seems to be a temporary intermediate state where the entry is rendered invalid but is still present in the B-tree, but this seems to only be the case on live systems. It may be possible to find the data for deleted entries after the end of a list in a B-tree node or in unused nodes. Future research will be oriented on data retrieval using B-trees in live computer forensics and live computer forensic in general.

References

[1] Kruse R. L., Ryba A. J. : Data Structures and Program Design in C++, Prentice-Hall Inc., New Jersey, USA, 2000.

[2] Sedgewick, R. : Algorithms, Addison-Wesley Publishing company,Massachusetts, USA, 1983.

[3] Grey, N. : A Beginners C++

[4] Comer, D. : The Ubiquitous B-Tree, Computing Surveys 11

[5] Cormen T. H., Leiserson C. E., Rivest R. L. : Introduction to algorithms, McGraw-Hill Book Company, New York, USA, 2000.

[6] Bayer R., Schkolnick M.: Concurrency of Operations on B-Trees. In Readings in Database Systems, 1994.

[7] Kristian Dreher: NTFS, Master's thesis

[8] Microsoft: How NTFS works available at `http://technet.microsoft.com/en-us/library/cc781134%28WS.10%29.aspx`, accessed: 20th April 2010.

[9] Philipp A., Cowen D., Davis C.: Hacking exposed Computer forensics, McGraw-Hill Book Company, New York, USA, 2010.