Paper Name: Data Structures
Unit No : VI
Lesson Name: M-Way Trees And B-Trees

Lesson Developer: Nidhi Arora, Assistant Professor
College/ Department: Kalindi College , University of Delhi.

Table of Contents

Chapter 2: M-Way Trees and B-Trees

The full form of M-way search tree is Multi-way trees. Actually binary trees are also multi way tree. Isn't it strange? But yes it is true. Let's now find the link between the two.

2.1 Binary Trees: A Special case of Multi Way Trees

A general definition of tree says:

A Tree is either a empty structure or a structure whose children are disjoint trees t_1 , t_2 ………..t_K .

According to this definition each node of this kind of tree can have any number of children. This is more general definition of the tree. Such kinds of trees are known as M-way trees or Multi way tree. These trees are so called as when we are on one node, and then there are multiple ways or multiple paths to different sub trees (as a node can have more than 2 children) from that node where we can proceed. But you have studied only those trees in which a node contains a single key and points to two sub trees. Such a tree is called binary tree. So now we can easily say that a binary tree is a multi way tree with a restriction of two children per node. A Multi way tree and a Binary tree are shown in figure 2.1 (a) and 2.1 (b) respectively.

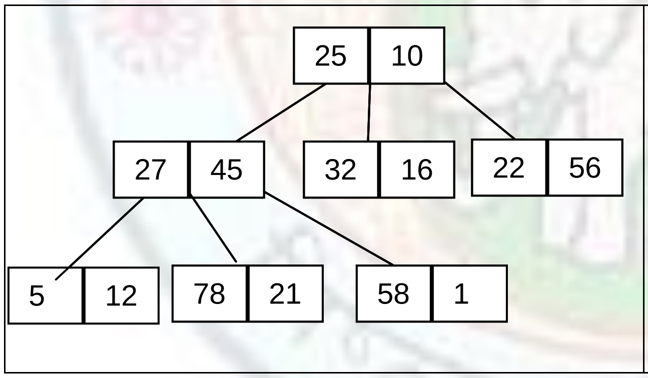


Fig 2.1 (a) Multiway tree

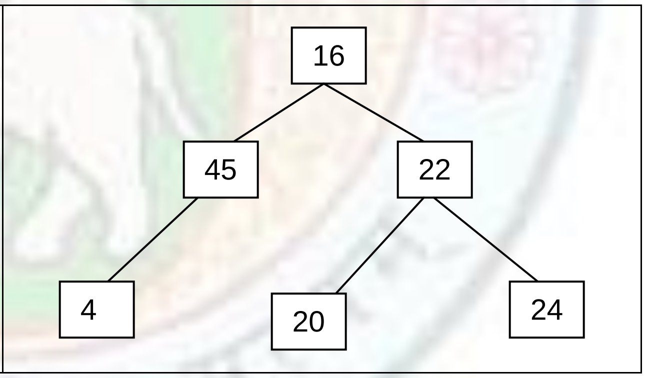


fig 2.1(b) Binary Tree

The maximum number of children that a node in a tree can have is known as the order of the tree. Multiway trees have order m. We can also call them m-way trees in short where m is the order of the Multiway tree. Binary tree is a Multiway tree of order 2, so they can also be called 2-way trees.

2.1.1 Interesting property of Multiway trees

Till now we have talked only about the number of children in a multiway tree. But there is one very interesting property of M-way trees related to the number of keys that it can hold in a node. In any multi way tree of order n, the numbers of children are at most n and the numbers of keys in any node are at most n-1. For example in a 3-way tree the maximum number of permissible children per node is 3(less is also allowed) and the maximum number of keys per node is 3-1=2. The numbers that we have given states the upper limit. So a node in such a tree can also have 2 children. Such a node must be having at least 1 key value in it, that is 1 less than the number of children but it can have keys up to n-1 values, as Null Child is also allowed. So we can state that:

If a node in a m-way tree of order m has k Children where k<=m then the node should have number of keys between k-1 and m-1 , where k-1 should be greater than or equal to 1.

Now we can apply this definition to binary trees as they are also M-way trees of order 2 and see that in a Binary tree maximum children per node is 2 and maximum values per node is 1 which is perfectly in synchronization with the above definition.

A 3-way tree is shown in figure 2.2.

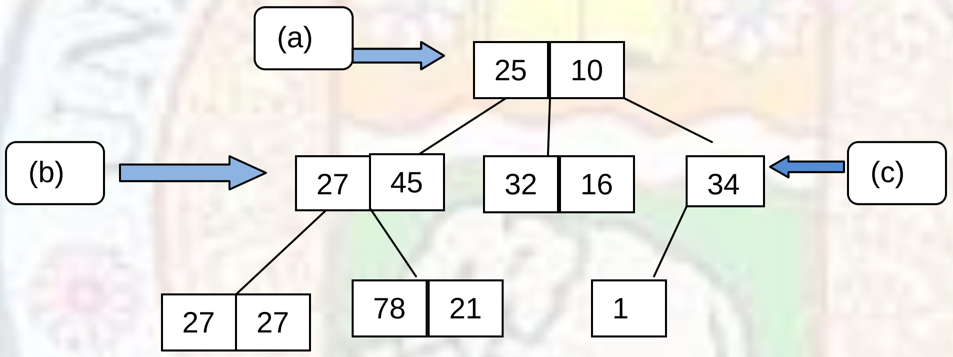


Fig 2.2 A 3-Way Tree

Node labelled (a) has 3 children and exactly 3-1=2 key values. Node labelled (b) has 2 children and 2 key values which is greater than 1(1 key value is at least expected in this node as it has two children). Node labelled (c) has one child and 1 key value which should be the least number of keys permissible in any node as a node can't have 0 number of keys.

Value Addition: Did you know?
List of different types of trees.

Binary trees

- Binary tree
- Binary search tree
- Self-balancing binary search tree
- Randomized binary search tree
- Weight-balanced tree
- Threaded binary tree
- AVL tree
- Red-black tree
- AA tree
- Scapegoat tree
- Splay tree
- T-tree
- Rope
- Top Trees
- Tango Trees
- Cartesian tree
- Van Emde Boas tree
- Treap

B-trees

- B-tree
- B+ tree
- B*-tree
- B sharp tree
- Dancing tree
- 2-3 tree
- 2-3-4 tree
- Queaps
- Fusion tree
- Bx-tree

Multiway trees

- Ternary search tree
- And–or tree
- (a,b)-tree
- Link/cut tree
- SPQR-tree
- Spaghetti stack
- Disjoint-set data structure
- Fusion tree
- Enfilade
- Exponential tree
- Fenwick tree

Heaps

- Heap
- Binary heap
- Binomial heap
- Fibonacci heap
- 2-3 heap
- Soft heap
- Pairing heap
- Leftist heap
- Treap
- Beap
- Skew heap
- Ternary heap
- D-ary heap

Tries

In these data structures each tree node compares a bit slice of key values.

- Trie
- Radix tree
- Suffix tree
- Suffix array
- Compressed suffix array
- FM-index
- Generalised suffix tree
- B-trie
- Judy array

Other kinds are also possible.

Source : http://www.servinghistory.com/topics/List_of_data_structures::sub::Trees

2.1.2 M-Way Search Trees

In the preceding chapters we have also studied binary search trees as special case of binary trees where key values of the left child of a node is always less than the key value of the node itself and key values of right child of a node is always greater than the key value of the node itself. Such an arrangement maintains the order of the keys which is helpful in fast searching and updation operations.

Such a link also exists between M-way trees and M-way search trees. In M-way search trees an order is maintained for the key values of the node. M-way search trees are used also for their fast searching and updation operation. Now let us give a formal definition of M-way search trees.

A Multi way search tree of order m, is a tree in which

a. The nodes hold between 1 to m-1 distinct keys
b. The keys in each node are sorted
c. A node with k keys has k+1 sub trees, where the sub trees may be empty.
d. The i^th^ sub tree of a node [$v_1, \dots, v_k$], $0 < i < k$, may hold only values v in the range $v_i < v < v_{i+1}$ (v_0 is assumed to equal $-\infty$, and v_{k+1} is assumed to equal *infinity*).

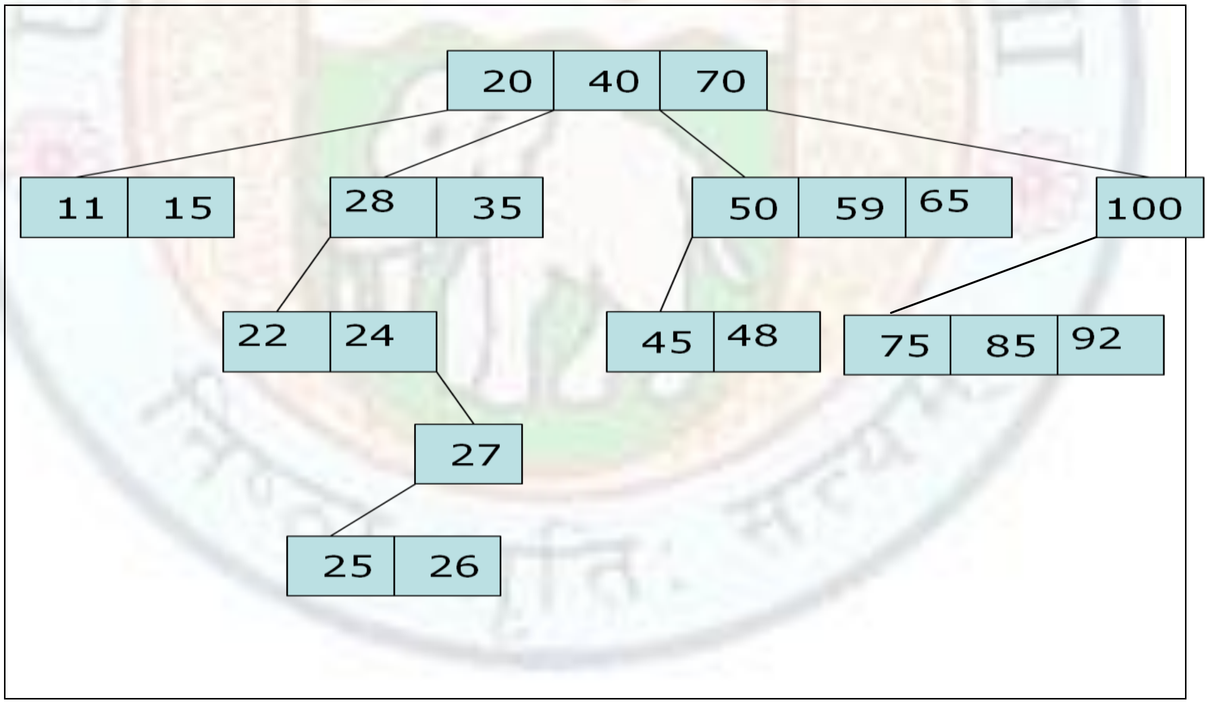


Fig 2.3 Multiway search tree of order 4.

Let's emphasize on the property (d) of the above definition. We will understand it with the following example.

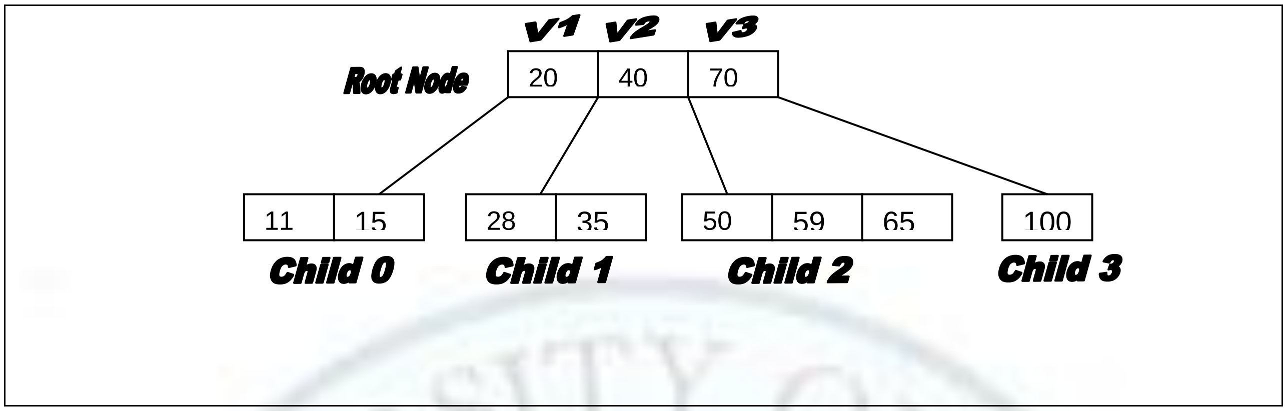


Fig 2.4 Parent child relationship in M-way search trees

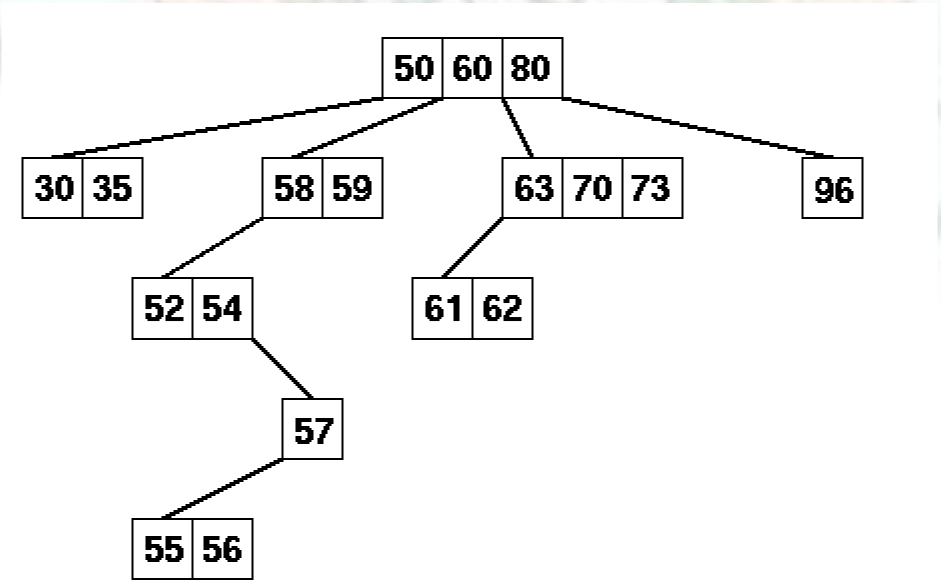
Let us consider figure 2.4 of a 4-way tree and consider the root node with values [20,40,70] arranged in ascending order. This node has 4 children numbered 0-3 say 0-[11,15] ,1-[28,35],2-[50,59,65] and 3-[100]. the first sub tree of the root node numbered 0 contains values between V0=infinity and V1=20 , the 2nd sub tree of the node numbered 1 contains the values between V1=20 and V2=40, the 3rd sub tree of the node numbered 2 contains values between v2 =40 and v3=70 and the 4th sub tree of the node numbered 3 contains values between V3=70 and V4=+infinity in this case. So a order is maintained in the key values of the node as well as the children of the node.

Note :An m-way tree of height h has between h and m^h - 1 key.

Value Addition: Did you know?

How to search a value in M-way search trees?

Consider the following M-way search tree :



The keys in the node are stored in ascending order V1 < V2 < ... < Vk

Searching in M-way Trees for X:

1. If X < V(1), recursively search in V(1)'s left sub tree.
2. If X > V(k), recursively search in V(k)'s right sub tree.
3. If X = V(i), for some i, X is found!
4. Else, for some i, V(i) < X < V(i+1); recursively search in sub tree between V(i) and V(i+1).

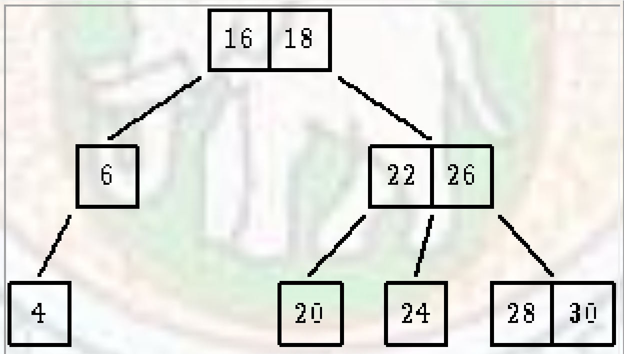
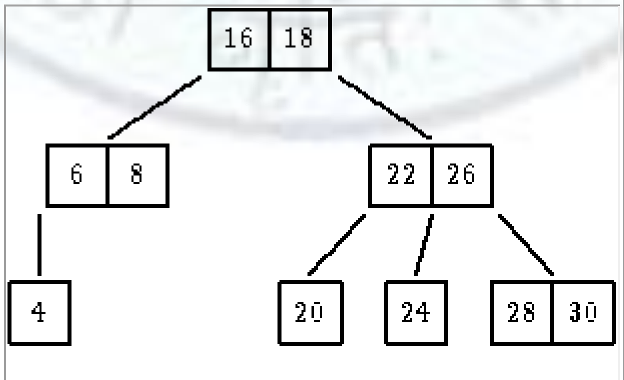
Try to search 57 in the above tree.

Source : http://www.scribd.com/doc/20637188/M-WAY-TREES-PDF

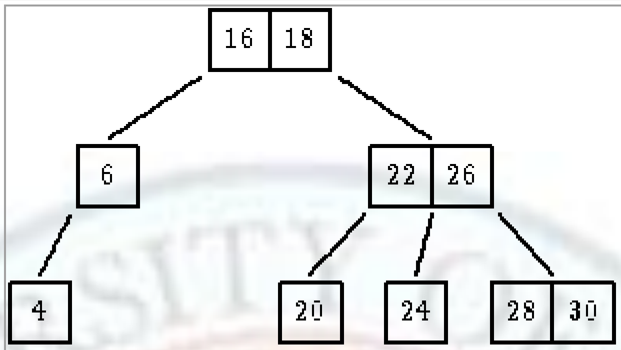
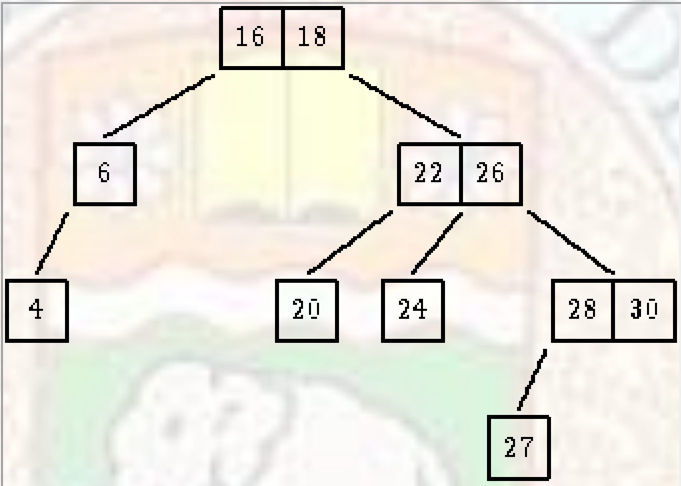
Value Addition: Did you know?

How to insert a value in M-way search trees?

- Search the key going down the tree until reaching an empty sub tree
- Insert the key to the parent of the empty sub tree, if there is room in the node.

	
insert 8	

- Insert the key as a new, if there is no room in its parent as shown below.

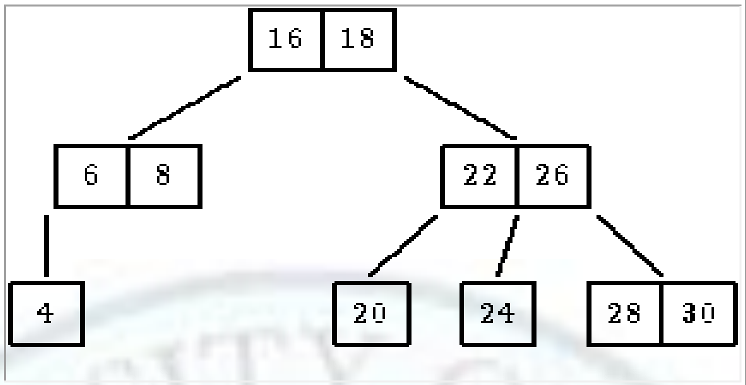
	
insert 27	

Source: http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch12.html

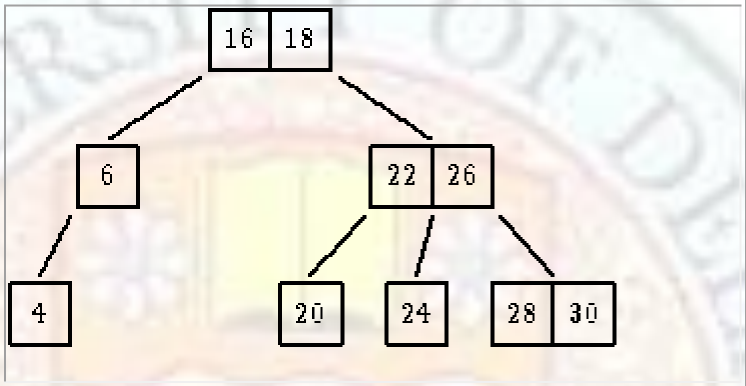
Value Addition: did you know?

How to delete a value in M-way search trees?

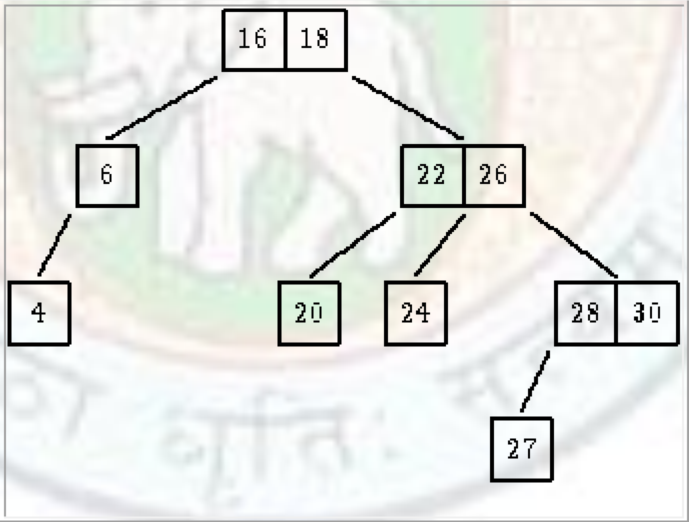
- If the key to be deleted is between two empty sub trees, delete it ,Remove the node if it becomes empty.

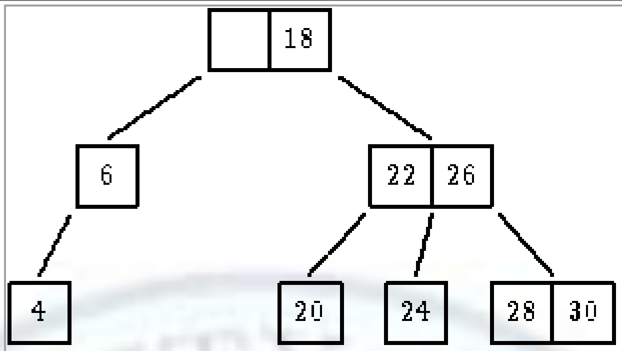
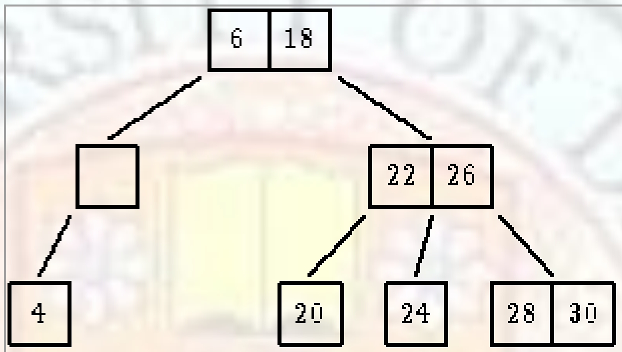
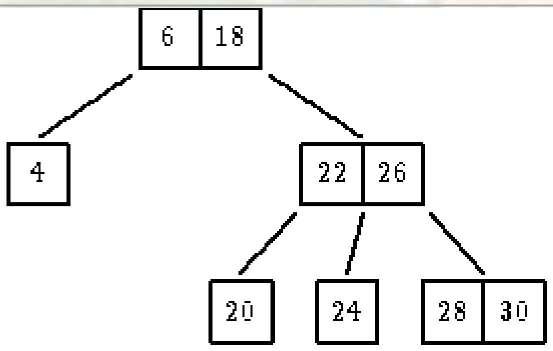


delete 8

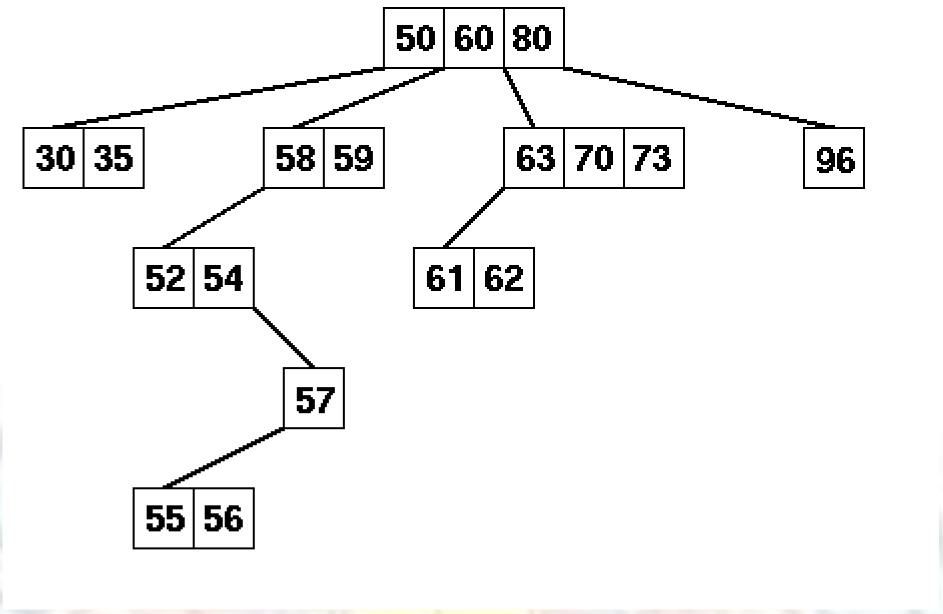


- If the key is adjacent to a nonempty sub tree, replace it with the largest key from the left sub tree or the smallest key from the right sub tree



remove 16	
	
	
Source: http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch12.html	

Now consider the following M-way Tree :



The draw backs of Multi way trees are also same as that of Binary search trees that they are unbalanced. A search for node 30 compares only 2 nodes while the search for 62 compares 5 nodes. So a need for making them balanced is there which is fulfilled by B- trees. This part is covered in subsequent part of this chapter.

But now you must be wondering what is the actual use of these trees is or what is the benefit that we are getting by storing Multi Way search trees with respect to Binary Search Trees. Let's see in the next section.

2.1.3 History behind M-way Trees

Let us see in which scenarios M-way trees or M-way search trees are required. Think about a scenario where the data on which we are operating is very large to be brought in the main memory. In this case we arrange the data in the disk storage.

Data on the disks is arranged in blocks. Block is the basic unit of read and write operations on the disk, which means at a time, one read operation reads one block and one write operation writes one block on the disk. These block sizes however can differ for different disks. Typical block sizes are between 512 bytes and 4096 bytes.

The typical disk access time is 1-10 ms, whereas the typical main memory access time is 10-100 ns. Let's take one example:

The disk is arranged in the form of multiple circular tracks and sectors of bytes known as block. Each time a information is requested on the disk the following set of operations are performed.

First it has to be found on which track the information is present.

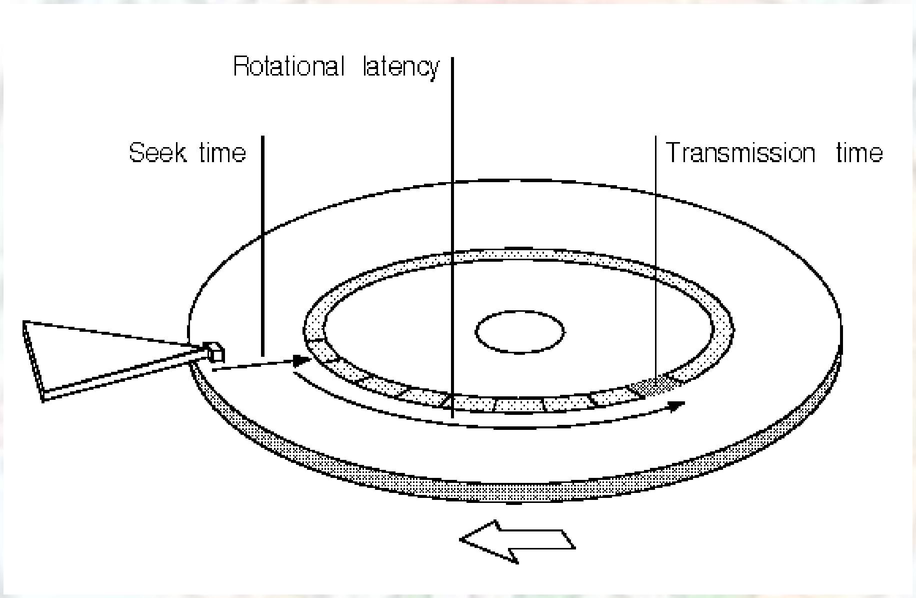
1. Then the head is moved to that particular track

2. After positioning the head on a particular track, the track is moved in circular manner so to position the track under the head.
3. Then the disk has to spin so that the entire block passes underneath the head to be transferred to memory.

Value Addition: Did you know?

Layout of hard disk

A hard disk's performance is characterized by its access time; this is the sum of the seek time taken by the head to move to the correct track, the rotational latency while the disk rotates until the head is over the desired block, and the transmission time needed to transfer data to or from the block as shown in ``The performance characteristics of a hard disk''.



These activities take a finite time to complete and are usually quoted as average figures. The peak transfer rate, which is also commonly quoted as a measure of disk performance, depends directly upon these activities. However, peak transfer rates are unusual, especially on disks which are used close to their maximum capacity. These suffer from greater fragmentation of data than emptier disks. If you have already purchased your disks, then the only use of the above values is in deciding which disks you should use to store your files.

Another factor that you can take into consideration if you know the geometry of your disks is the layout of file systems across the disk surface. Modern disks often have more blocks per track in the outer cylinders than in the inner tracks. When reading contiguous blocks, data from the outer tracks is transferred faster than from inner tracks. The innermost tracks, however, have lower latency times when reading relatively few blocks.

If the applications on your system access large sequential files, such as database journal logs, graphics images, fonts and PostScript, you may see a gain in disk

performance in putting these files in file systems on the outer tracks of your disks.
Source: http://osr507doc.sco.com/en/PERFORM/disk_IO_mech.html

So lot of delay is there before actual block starts to transfer in the main memory. The time taken in step 1 is known as seek time. The time taken in step 2 is known as latency time or rotational delay .the time taken in step 3 is transfer time. This means that there are several time components for the data access.

Access time= Seek Time + Rotational Delay + Transfer Time.

This time of accessing information from the disk is extremely slow as compared to transferring information within the memory.

2.1.3.1 Example of disk access

Latency is the time required to position the head on the correct block and on an average it is the time needed to make one half of the revolution. The time to transfer 5 KB (kilobytes) from a disk requiring 40 ms(milliseconds) to locate a track, making 3000 revolutions per minute and with a data transfer rate of 1000 KB per second is

Access time =40 ms+10ms +5ms=55ms

If 10 KB has to be transferred then it will take:

Access Time = 40 ms +10 ms +10 ms. = 60 ms.

Now see the case if this 10KB is spreaded in two different blocks of 5KB each then 2 disk accesses has to be made which will put more delay on the total access time.

Access Time = 2 * (40 ms+10ms +5ms) =110 ms.

So we should try to make most when we are accessing one block by arranging maximum information in one block rather than spreading the information in multiple blocks containing small data each.

Now consider the case of binary search trees where each node can only have single key. If we store such trees on disks then the nodes can be spreaded over different blocks. Different block accesses has to be done to either retrieve, insert or delete any element which will make such a good Data Structure (if it would have resided on main memory) a slow performer and will subsequently reduce the overall application’s performance.

The overall performance will be increased by using M-way Trees.

In the figure 2.5 we are showing the memory allocation to M-way trees on the Hard-Disk.

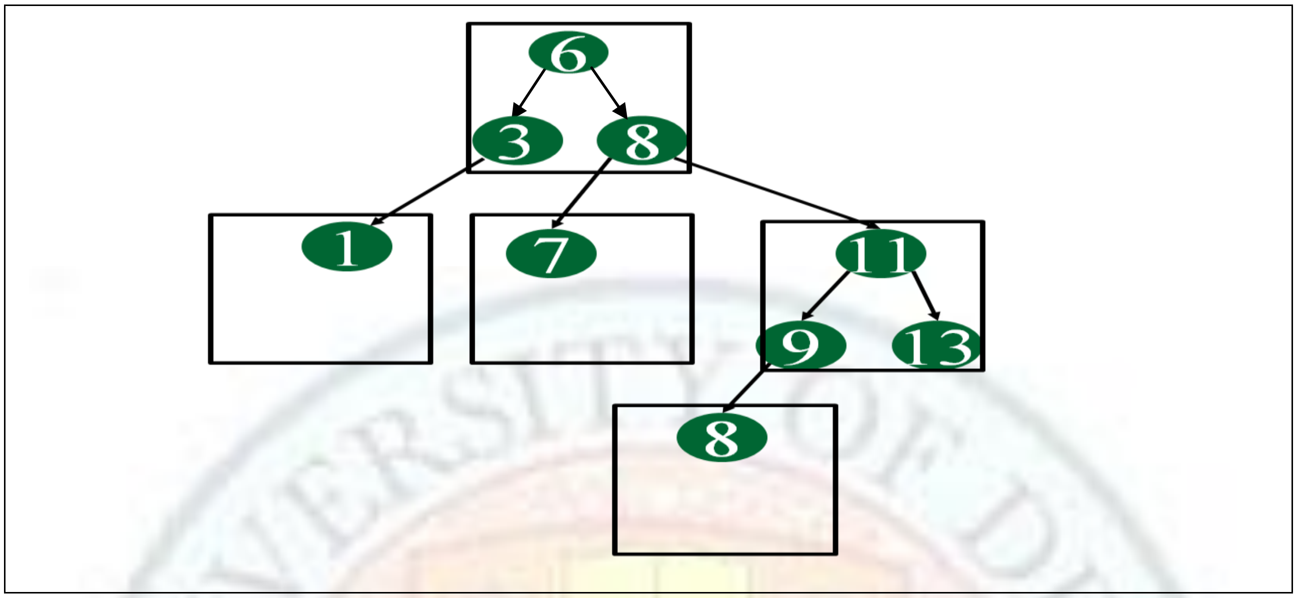


Fig 2.5 Block wise storage

In the above figure 2.5, multiple nodes are allocated to one single block so as to minimize the number of different disk accesses rather than spreading the information in different blocks as is done in binary search trees.

Value Addition: Interesting Fact
Benefit of large value of M for M-way Tree
By choosing a suitably large value for *M*, we can arrange that one node of an *M*-way search tree occupies an entire disk block. If every internal node in the *M*-way search tree has exactly *M* children, we can use Theorem to determine the height of the tree: Theorem: consider an M-ary tree T of height h>=0. The maximum number of internal nodes in T is given by $n=M^{h+1}-1/M-1$ Therefore h>= | $\log_M$ ((M-1)n+1)|-1 where n is the number of internal nodes. A node in M-way search tree that has *M* children contains exactly *M*-1 keys. Therefore, altogether there are *K*= (*M*-1)*n* keys and above Equation becomes $h \geq \lceil \log_M (K+1) \rceil - 1$. Ideally the search tree is well balanced and the inequality becomes equality. For example, consider a search tree which contains $K = 2\,097\,151$keys. Suppose the size of a disk block is such that we can fit a node of size *M*=128 in it. Since each node contains at most 127 keys, at least 16513 nodes are required. In the best case, the height of the *M*-way search tree is only two and at most three disk accesses are required to retrieve any key! This is a significant improvement over a binary tree, the height of which is at least 20.

Source: Data Structures and Algorithms with Object-Oriented Design Patterns in C++, Bruno R. Preiss

2.2 B-Trees

We have already discussed when data bases are stored on the Hard disk (as they are very large to be kept in the main memory), their access is very time consuming and slow as compared to the data which is stored on the main memory. To reduce this time penalty for locating and accessing the data on the hard disk blocks we need proper data structures.

M-way search Trees are one such data structure we have already discussed. But they are not used in their raw form as we have discussed. The main drawback of M-way trees is that they are not balanced. In real scenarios if we can get to use some balanced M-way tree the data structure will be really beneficial. And one such balanced M-way tree is called B-Tree.

The B-tree (not to be confused with Binary Tree) Data Structure was given by Bayer and McCreight in 1972. It is a Balanced M-way tree.

Value Addition: Did You know?

Meaning of 'B' in B-Tree

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)

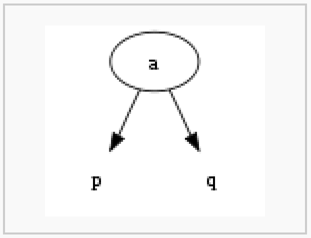
Source: http://en.wikipedia.org/wiki/B-tree

Before going into the intricacies of B-Trees let us first give a brief insight into the very well known Indexed Sequential access Method (ISAM) for large dictionaries which are kept on external storage devices .

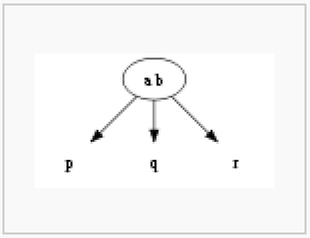
Value Addition: Did You know?

What are 2-3 Trees and 2-3-4 Trees?

A **2-3 tree** in computer science is a type of data structure, a B-tree where every node with children (internal node) has either two children and one data element (2-nodes) or three children and two data elements (3-nodes). Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements.



2 node



3 node

2-3 trees are an isometric of AA trees, meaning that they are equivalent data structures. In other words, for every 2-3 tree, there exists at least one AA tree with data elements in the same order. 2-3 trees are balanced, meaning that each right, center, and left subtree contains the same or close to the same amount of data.

Properties

- Every non-leaf is a 2-node or a 3-node. A 2-node contains one data item and has two children. A 3-node contains two data items and has 3 children.
- All leaves are at the same level (the bottom level)
- All data is kept in sorted order
- Every non-leaf node will contain 1 or 2 fields

2-3-4 Tree

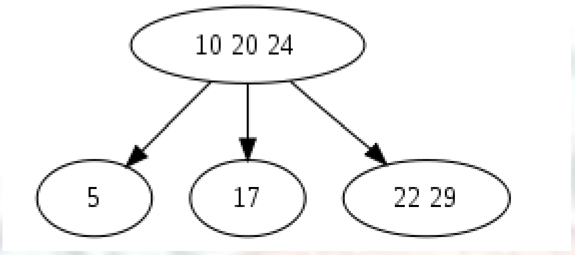
A **2-3-4 tree** (also called a **2-4 tree**), in computer science, is a self-balancing data structure that is commonly used to implement dictionaries. 2-3-4 trees are B-trees of order 4; like B-trees in general, they can search, insert and delete in O (log *n*) time. One property of a 2-3-4 tree is that all external nodes are at the same depth.

2-3-4 trees are an isometric of red-black trees, meaning that they are equivalent data structures. In other words, for every 2-3-4 tree, there exists at least one red-black tree with data elements in the same order. Moreover, insertion and deletion operations on 2-3-4 trees that cause node expansions, splits and merges are equivalent to the color-flipping and rotations in red-black trees. Introductions to red-black trees usually introduce 2-3-4 trees first, because they are conceptually simpler. 2-3-4 trees, however, can be difficult to implement in most programming languages because of the large number of special cases involved in operations on the tree. Red-black trees are simpler to implement, so tend to be used instead.

A **2-3-4 tree** (also called a **2-4 tree**), in computer science, is a self-balancing data structure that is commonly used to implement dictionaries. 2-3-4 trees are B-trees of order 4; like B-trees in general, they can search, insert and delete in O(log *n*) time. One property of a 2-3-4 tree is that all external nodes are at the same depth.

2-3-4 trees are an isometric of red-black trees, meaning that they are equivalent data structures. In other words, for every 2-3-4 tree, there exists at least one red-black tree with data elements in the same order. Moreover, insertion and deletion

operations on 2-3-4 trees that cause node expansions, splits and merges are equivalent to the color-flipping and rotations in red-black trees. Introductions to red-black trees usually introduce 2-3-4 trees first, because they are conceptually simpler. 2-3-4 trees, however, can be difficult to implement in most programming languages because of the large number of special cases involved in operations on the tree. Red-black trees are simpler to implement, so tend to be used instead.



Source: http://en.wikipedia.org/wiki/2-3_tree

2.2.1 Indexed *Sequential Access Method(ISAM)*

ISAM provides an efficient sequential and random Access(via an index) for the data bases which are stored on the hard disk. The data organized on hard disk is in sequential manner. Hard disks are semi sequential storage devices which mean if we want to search some record; the disk head is placed on the track and reads the data along the track sequentially. In order to read a specific section, the head has to move to a track containing that sector.. As there will be so many block accesses the time to access such data is dependent upon the seek time and latency time. Typically a block is one track long and can be input and outputted by single seek and latency delay.

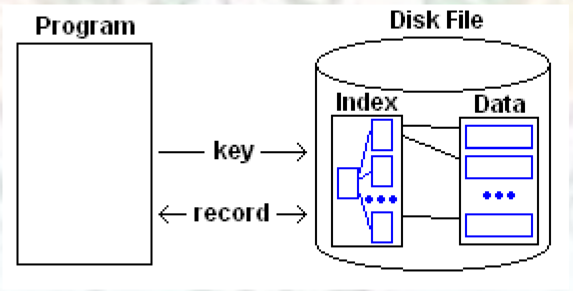


Fig 2.6 ISAM
www.comsci.us/fs/notes/ch10.html

2.2.1.1 Sorting: Speed up the searching process

If we arrange our records in sorted order then the searching time will be benefited and reduced by the order of two. Binary search algorithm when executed on sorted file of N records can search a particular record in $\text{Log}_2 n$ time.

So if there are 1000000 records in file in sorted order , then a specific element can be searched by doing $\log_2 1000000=19.93 \approx 20$ comparisons by using binary search which is much better than the sequential search of any unsorted file. .
If we talk about the situation in which these records are stored on the hard disks in case of large databases then naively the time to locate one record out of a million would take 20 disks reads times .each disk read will take time as per seek time and rotational time delays . If say each disk read takes 10 milliseconds (which accounts it's seek time and rotational time), then it will take 0.2 second to locate a record on such a disk.

Statistics reveal than nearly last 6 comparisons will be done with in the single block as blocks contain many records say 100 records per block. So search is reduced to nearly 14 disk read operations in the above scenario.

We still need to speed up this search.

2.2.1.2 Use index for faster searching

A significant improvement can be made with an index. Index contains the first record of each disk block which will make the index size nearly 1% of the original database. Finding an entry in such an index would tell us which block to search in the main database; after searching the index also known as auxiliary index, we would have to search only that one block of the main database—at a cost of one more disk read. If there are 100 records per disk block and there are total 1000000 records then the index would hold up to 10,000 entries, so it would take at most 14 comparisons. Like the main database, the last 6 or so comparisons in the 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.

So in ISAM ,on index for finding a particular block on the disk and then the block is sequentially searched for a particular record.

The trick of creating an auxiliary index can be repeated to make another auxiliary index to the first 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.

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.

2.2.1.3 Trouble of 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 doesn't cause much trouble. The index can stay the same, and the record can just be marked as deleted. The database stays in sorted order. If there are a lot of deletions, then the searching and storage becomes less efficient.

Insertions are a disaster 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.

A trick is to leave some vacant space 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.

Now, 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.

2.2.1.4 The B-Tree

The B-tree uses all the above ideas. It keeps the records in sorted order so they may be sequentially traversed. It uses a hierarchical index to minimize the number of disk reads. The index is elegantly adjusted with a recursive algorithm. The B-tree uses partially full blocks to speed insertions and deletions. In addition, a B-tree minimizes waste by making sure the interior nodes are at least 1/2 full. A B-tree can handle an arbitrary number of insertions and deletions.

So to conclude B-tree is a Data Structure suitable for the indexes which reside on the hard disk.

Value Addition: Biography

Rudolf Bayer

Rudolf Bayer (born 7 May 1939) has been Professor (emeritus) of Informatics at the Technical University of Munich since 1972. He is famous for inventing two data sorting structures: the B-tree with Edward M. McCreight, and later the UB-tree with Volker Markl. He also invented red-black trees.

He is a recipient of 2001 ACM SIGMOD Edgar F. Codd Innovations Award

Rudolf Bayer

2001 SIGMOD Edgar F. Codd Innovations Award

Rudolf Bayer studied Mathematics in Munich and at the University of Illinois, where he received his Ph.D. in 1966. After working at Boeing Research Labs he became an Associate Professor at Purdue University. He is a Professor of Informatics at the Technische Universität München since 1972 and Head of the Research Group Knowledge Bases at the Bavarian Research Center for Knowledge Based Systems (FORWISS). He consulted for IBM, Siemens, Amdahl, DEC, Deutsche Telekom and was a Visiting Professor at IBM, XEROX PARC, and at several Universities in Japan, Australia, the US and Singapore. He is a cofounder of TransAction SW GmbH and holds two patents. Rudolf served as an Associate Editor of ACM TODS and of Informatik Forschung und Entwicklung. He organized and directed several national research projects in SW Engineering, Deductive Databases and Digital Libraries, served on the Präsidium der Gesellschaft für Informatik and is a consultant of the German Wissenschaftsrat and for various government institutions. He was awarded the Bundesverdienstkreuz of the Federal Republik of Germany and received the SIGMOD Innovations Award in 2001.

Source: http://www.sigmod.org/sigmod-awards/award-people/rudolf-bayer

Value Addition: Biography

Edward M. McCreight

Edward M. McCreight

He lives in Zürich, Switzerland. He grew up in Washington, PA (Pennsylvania, USA), and attended the College of Wooster ('66) and Carnegie-Mellon University. He had played with computers since 1963. He is retired. In the past he has worked at Boeing Aircraft, Xerox PARC, and Adobe Systems. He has been a guest professor at the University of Washington, Stanford University, the Technical University of Munich, and the Swiss Federal Institute of Technology in Zürich. Before moving to Zürich, he lived thirty-two years in Los Altos, CA (California, USA) (2008)

He regularly reads email addressed to: ed@mccreight.com

Source: http://www.mccreight.com/people/ed_mcc/index.htm

2.2.2 Definition of B-Tree

A **B-tree** is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic amortized time. It is a well suited data structure for large databases which are stored on Hard Disks.

B-trees can be tuned to reduce the time penalty introduced by Hard disk accesses and fasten the search .One important property of B-trees is that the size of each node here can be made as large as the block. The number of keys in the one node can vary depending upon the size of block and size of the keys.

Value Addition: Frequently Asked Question

What is the difference between B-Trees and Binary Trees?

The major difference between B-tree and binary trees is that B-tree is a external data structure and binary tree is a main memory data structure. The computational complexity of binary tree is counted by the number of comparison operations at each node, while the computational complexity of B-tree is determined by the disk I/O, that is, the number of node that will be loaded from disk to main memory. The comparison of the different values in one node is not counted.

Source: http://dev.fyicenter.com/Interview-Questions/CPP-2/What_are_the_advantages_and_disadvantages_of_B_s.html

A B-tree of order m is a Multiway search tree of the following properties.

1. The root has at least two sub trees unless it is a leaf.
2. Each non root and each non leaf node holds k-1 keys and k pointers to sub trees where k lies between m/2 and m.
3. Each non leaf node holds k-1 keys where k lies between m/2 and m.
4. All leaves are on the same level.

According to this definition a B-Tree is always half full, has few levels and is perfectly balanced.

Value Addition: Interesting Fact

Terminology related to B-Trees

The terminology used for B-trees is inconsistent in the literature:

Unfortunately, the literature on B-trees is not uniform in its use of terms relating to B-Trees. (Folk1992 page 362)

Bayer1972, Comer1979, and others define the **order** of B-tree as the minimum number of keys in a non-root node. Folk1992 points out that terminology are 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. Knuth1993b 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. Bayer1972 considered the leaf level to be the lowest level of keys, but Knuth1993b considered the leaf level to be one level below the lowest keys. (Folk1992 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. Bayer1972 avoided the issue by saying an index element is a (physcially adjacent) pair of where is the key, and 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. Bayer1972 states, "For this paper the associated information is of no further interest."

There are also unfortunate choices like using the variable to represent the number of children when 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. (Folk1992 pg 379.)

Source: http://www.absoluteastronomy.com/topics/B-tree

2.2.2.1 Structure of a Node in B-Tree

Usually m is very large (50-100) so that the information stored in one block of the secondary storage can fit into one block. As we have already discussed that the B-tree is used to store indexes of the records on the secondary storage. Here a key field is associated with each of the record which will act as an index field for that record. We are only interested in storing the key fields of the records in B-tree along with their addresses on the hard disk.

In the long run, it is better than keeping the whole records in the nodes as it will result in less keys per nodes and hence, more number of nodes per tree and hence deeper trees. And thus will make the search path longer resulting in more running time.

For simplicity and for the sake of further discussion we will portray our B-Trees containing only keys as shown in the following figure 2.7.

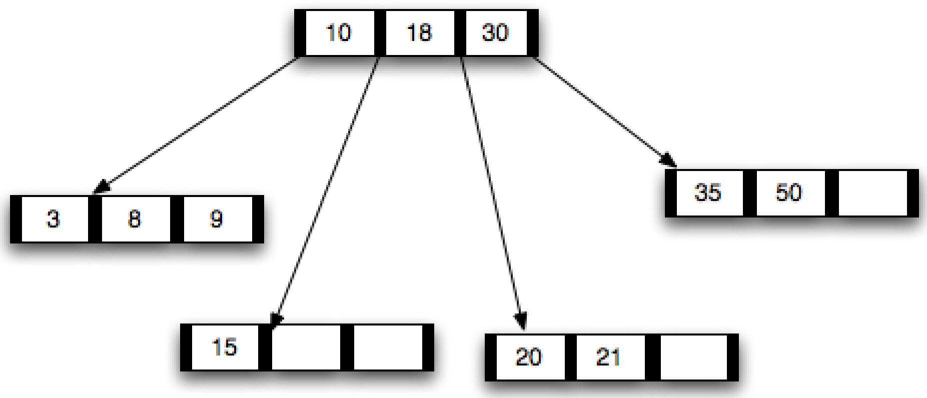


Fig 2.7 B-tree of order 4
http://scienceblogs.com/goodmath/btree-one-insert.png

Value Addition: Interesting Fact

B-trees in Relational Data Base Systems (RDBS)

B-trees as Primary Indexes

State of the art in RDBS is to use the primary key of a relation as the key for any variant of a B-tree. The data of a tuple is stored in the leaves of the B-tree, i.e. the complete table is stored in the B-tree. Alternatively, the tuple itself is stored in a separate data structure and the associated information in the B-tree is just a pointer to the tuple. In many RDBS applications, tuple access via the primary key is the most frequent retrieval operation and is made very fast by a B-tree index.

B-trees as Secondary Indexes

Arbitrary attributes (columns) of a relational table may also be indexed by a B-tree. In this case, the associated information is a set of primary key values or artificial internal identifiers, also called *database identifiers*, which are then used to access the tuple itself.

Both, primary and secondary indexes are ideal to answer SQL queries of the following forms quickly, where R is a relation, A is an attribute of R, and c and d are constants:

- *select * from R where A = c;*
- *select * from R where A is between c and d;*

B-trees as Join Indexes

Relational table joins result from queries like

- *select * from R1, R2 where R1.A = R2.B;*

Here, for a given tuple from R1 and known value for the attribute R1.A the matching tuples from the second relation R2 must be fetched. The access to R2 is obviously a point query and is very fast, if an index on the attribute B of R2 is available. Of course, this technique also works symmetrically with respect to R1 and R2.

The use of B-trees in RDBS is ubiquitous and manifold; their best use must be determined by the RDBS optimizer and is a complex task way beyond this survey.

Some relational queries can even be answered by the index itself without accessing the relation at all, e.g.

- *select count(*) from Employees;*

Parallel Processing with B-trees

In many applications, like banking, electronic shopping, Web or library queries, highly parallel processing within the databases is needed (Bayer and Schkolnick 1977), (Weikum and Vossen 2002). This must be compatible with parallel updates and in addition requires non-stop operation. B-trees are ideally suited for such DBapplications for the following two reasons:

1. The root must be visited by every search and update process, but even for most updates it only needs to be read. Thus, the roots – and maybe some of its children – are read hotspots, and therefore they are always cached in main memory by the standard caching techniques of the database system and the operating system. This allows extremely fast and highly parallel processing.
2. On the other hand, most updates and structural transformations of B-trees are limited to the leaves and the lower levels of the tree, where they interfere very little.

To account for this processing scenario a variety of specialized synchronization techniques were developed for read- and update-transactions. They work with different types of locks, mostly on the node granularity, and follow locking protocols which take advantage of the special properties of B-trees. The first such synchronization protocol was developed in (Bayer and Schkolnick 1977) for System R of IBM Research. (Weikum and Vossen 2002) devotes an entire chapter to this topic.

http://www.scholarpedia.org/article/B-tree_and_UB-tree

2.2.2.2 Height of the B-Tree

Height of the B-tree will give us an upper bound on the running time of search operation. As we know that the search operation in Binary search tree takes time O(h) in worst case where h is the height of the BST , similarly the search operation in B-tree also takes O(h) time in the worst case. So now let’s try to find out what is this h i.e. height of a B-tree.

Let us imagine a situation when the B-tree has smallest allowable number of pointers per non root node i.e. q=m/2, and the search has to reach a leaf. In this situation the height of the tree is maximum and searching in such a tree will result in worst case running time.

Let us find out the total number of keys in such a tree.

One important thing to note in this situation is that if a node has q pointers to its children then it has q-1 keys as per the definition of B-Trees as shown in fig 2.8.

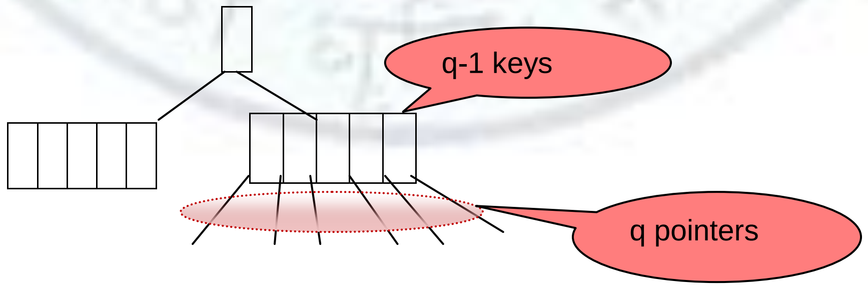


Fig 2.8 Relation between number of keys and pointers in a node

Level 1 = Root will contain at least 1 key.
Level 2 = There will be 2 non root nodes at this level => each non root node will have q pointers => resulting in q-1 keys per node => total 2(q-1) keys
Level 3 = There are q pointers emanating from each non root node at its previous level=> 2q such pointers => each pointer is linked to one node at this level => so there are 2q nodes at this level => each node contains q-1 keys => so there are 2q(q-1) keys at this level
Level 4 = There are 2q nodes at previous level and q pointers per node=> so there are $2q^2$ Nodes at this level with q-1 keys per node=> $2q^2(q-1)$ keys at this level

If height is h, the total number of keys in such a B-tree is :

$$\underset{\text{Root}}{\underset{\downarrow}{1}} + \underset{\text{second level}}{\underset{\downarrow}{2(q-1)}} + \underset{\text{third level}}{\underset{\downarrow}{2q\,(q-1)}} + \underset{\text{fourth level}}{\underset{\downarrow}{2q^2(q-1)}} + \ldots\ldots \underset{\text{level h}}{\underset{\downarrow}{2q^{h-2}(q-1)}}$$

$$=1+ \left(\sum_{i=0}^{h=2} 2q^i\right)(q-1) \text{ keys in the B-Tree} \qquad \text{(i)}$$

We know that, $\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}$

Substituting in (i) we get total number of keys n=

$$1+2(q-1)\left(\sum_{i=0}^{h-2} q^i\right) = 1+2(q-1)\left(\frac{q^{h-1}-1}{q-1}\right) = -1+2q^{h-1}$$

So $n \geq -1+2q^{h-1}$

Hence $h \leq \log_q \frac{n+1}{2} + 1$. So h= O(log n)

This means that for very large number of keys and large m, the height is small. For example if m=200 and n=2,000,000 then h<=4. i.e. finding a key in the worst case require only four seeks.

Value Addition: Frequently Asked Question

What is the height of the B-tree in Best case and Worst Case?

The best case height of a B-Tree is:

$$\log_m n.$$

The worst case height of a B-Tree is:

$$\log_{m/2} n$$

Where *m* is the maximum number of children a node can have.
Source: http://en.wikipedia.org/wiki/B-tree

2.3 Operations on B-Trees

2.3.1 B-Tree Search

The search operation on a b-tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b-tree search must make an n-way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree, *B-Tree-Search* is *O (log n)*.

Key k needs to be searched. A node contains total i keys from 0 to i-1.following is the algorithm to search.

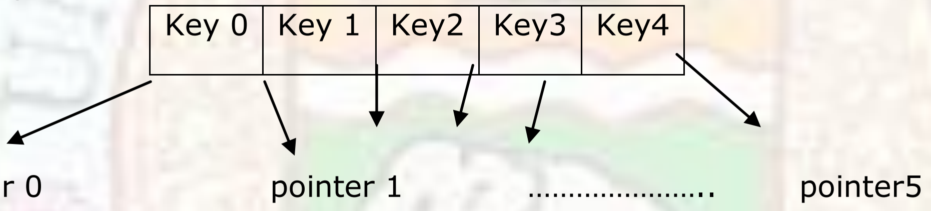


Fig 2.9A Node with 4 Keys in B-Tree.

Suppose there are 5 keys in a node numbered 0 to 4 then keynum value which signifies the number of keys in a node will become 5**.** This node is linearly searched for a key k starting from 0 to 4. If any key is found greater than k say key 1 is found greater than k than pointer 1 is followed which is left pointer of key 1. If the key is greater than the last key of the node then the right most pointer of that node is followed for the search. This process is repeated until the key is found or we reach the dead end.

B-Tree-Search (key k, node * node)

```
{

if (node !=0)
{
   for (i=1; i<= node-> keynum && node ->keys[i-1] <k;i++);
   if (i> node->keynum || node-> keys[i-1]>k)
       return B-Tree-Search(k,node->pointers[i-1]);
   else return node;
}
else return 0;}
```

Value Addition: Frequently Asked Question
B-trees indexes and Hash indexes
It depends on which storage engine you're using. For most, BTREE is the default so specifying it doesn't really change anything. For storage engines such as MEMORY/HEAP and NDB, the default is to use HASH indexes by default. More information can be found here. Whether or not a B-tree or a HASH index is advantageous for you from a performance perspective depends on the data and how you're accessing it. If you know you're queries are going to target exactly one row or scattered individual rows, then a HASH index may be useful. Anything other than that, I generally prefer a BTREE index as the data is sorted and thus makes range queries and those that return multi-rows more efficient. First off, depending on the Storage Engine used, you may just not have a choice (InnoDB for example is exclusively using BTREE for its index). Also, BTREE is the default index type for most storage engines. Now... There are cases, when using alternative index types may result in improved performance. There are (relatively rare case) when a HASH index may help. Note that when a HASH index is created, a BTREE index is also produced. That's in part due to the fact that hash indexes can only resolve equality predicates. (a condition such as WHERE Price > 12.0 could not be handled by a hash index). In short: Keep using BTREE, whether implicitly (if BTREE is the default for the Storage used), or explicitly. Learn about the other types of indexes so that you know about them would the need arise.
Source: http://stackoverflow.com/questions/1687910/advantage-of-btree

2.3.2 B-Tree Insert

In B-Trees, the tree is build from bottom up fashion in contrast to top down fashion of binary trees. This is done to maintain one very important property of B-Trees and that is we have to keep all the leaves at the same level. First a search for the proper location is made and then insertion is done at that position. Every new key is first inserted in the leaf, and then adjustments are done to upper levels according so as to grow the tree in bottom to up fashion.

The strategy here is:

The strategy here is:

1. Insert the new key into the appropriate leaf.
2. If after insertion of the new key the size of leaf node becomes more than what is allowed then perform the following action:

    a. Break the leaf node into two so that now we have two leaves.
    b. Promote one key out of the broken leaves to a node above one level and insert it there.
    c. If after this insertion the node is more than full, step 2 is followed again till we reach root node and a new root is formed.

Three common situations are encountered while inserting. Let's see

Situation 1: Inserting a key into a non full leaf.

See insertion of 12 in fig 2.10(a) to make it to the one like 2.10(b).

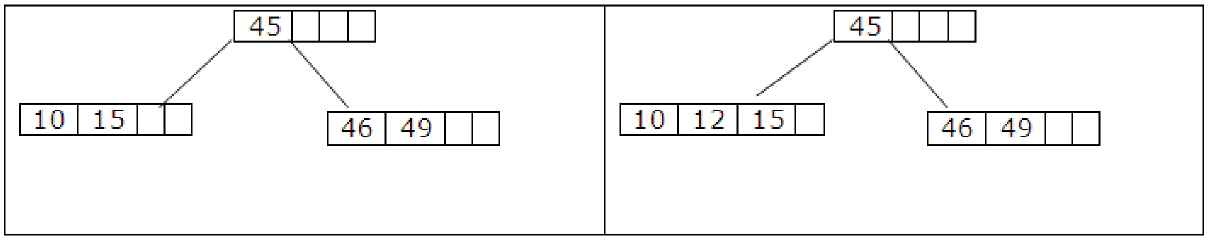


Fig 2.10 (a) Before Insertion Fig 2.10(b) After Insertion

Situation 2: Inserting a key into a full leaf. Insert 15 in a full leaf.

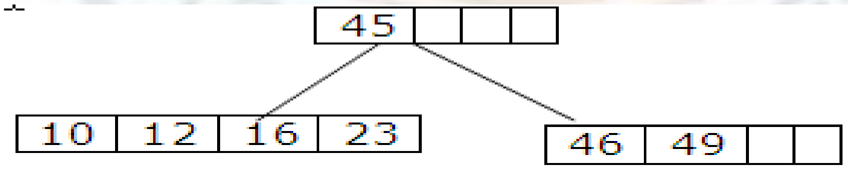
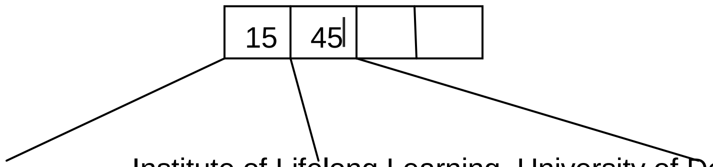


Fig 2.11 (a) Before Insertion



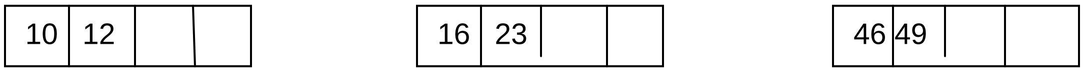


Fig 2.11(b) After Insertion

Here first a leaf where 15 can be inserted is found out which is already full. Never mind the key 15 is inserted and the leaf is broken into two to form two leaves out of one. , taking the right most key value of the old leaf to the parent i.e the middle element. This procedure is repeated for parent node as well. Each such split adds one more node to the B-Tree.

Situation 3: A special case when root also becomes full due to the keys coming up from bottom. This case arises when the B-Tree is already full and a new key comes.

Let's see inserting 59 in the following figure.

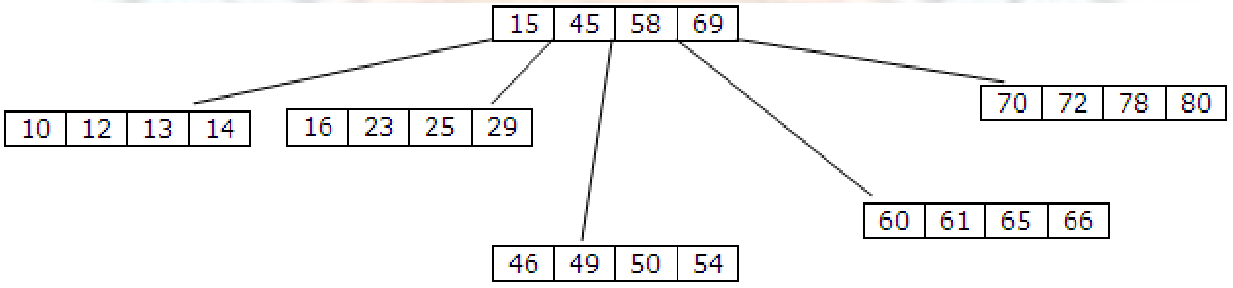


Fig 2.12(a)Before Insertion

In the above figure the key 59 will be inserted in leaf 4 which is already full. So leaf is broken in to two as shown in the following figure.

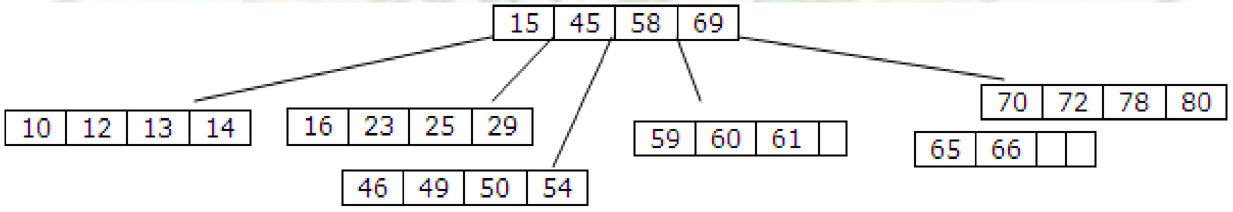


Fig 2.12(b)After Insertion stage 1

After breaking the leaf into two the right most element of the older leaf is taken to its parent. Here parent is the root which is also full. So now root is also broken to form two nodes and the middle element of the two nodes i.e. the rightmost element of the first node is taken above one level to form new root of the B-Tree as shown in the following figure..

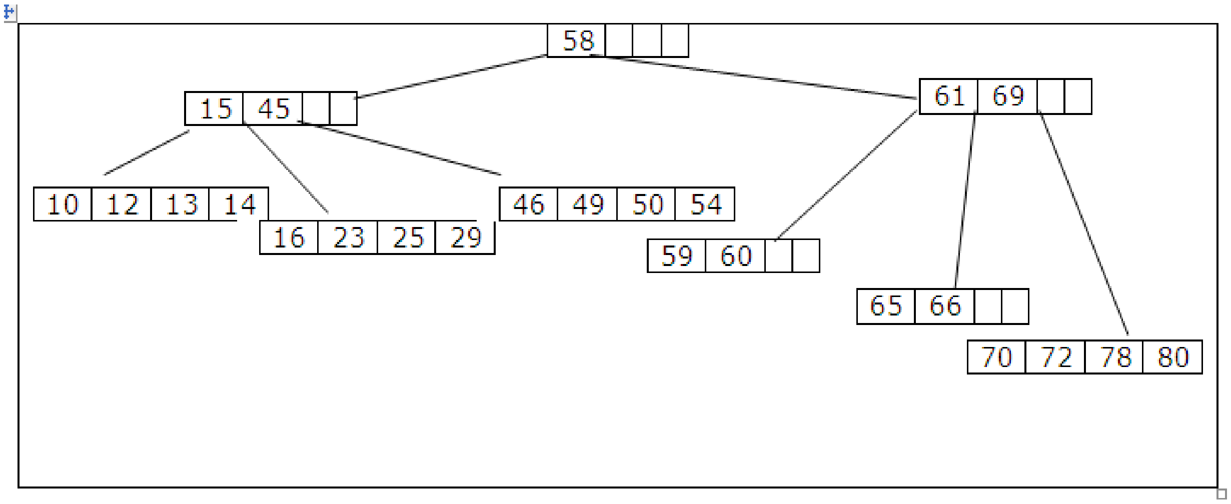
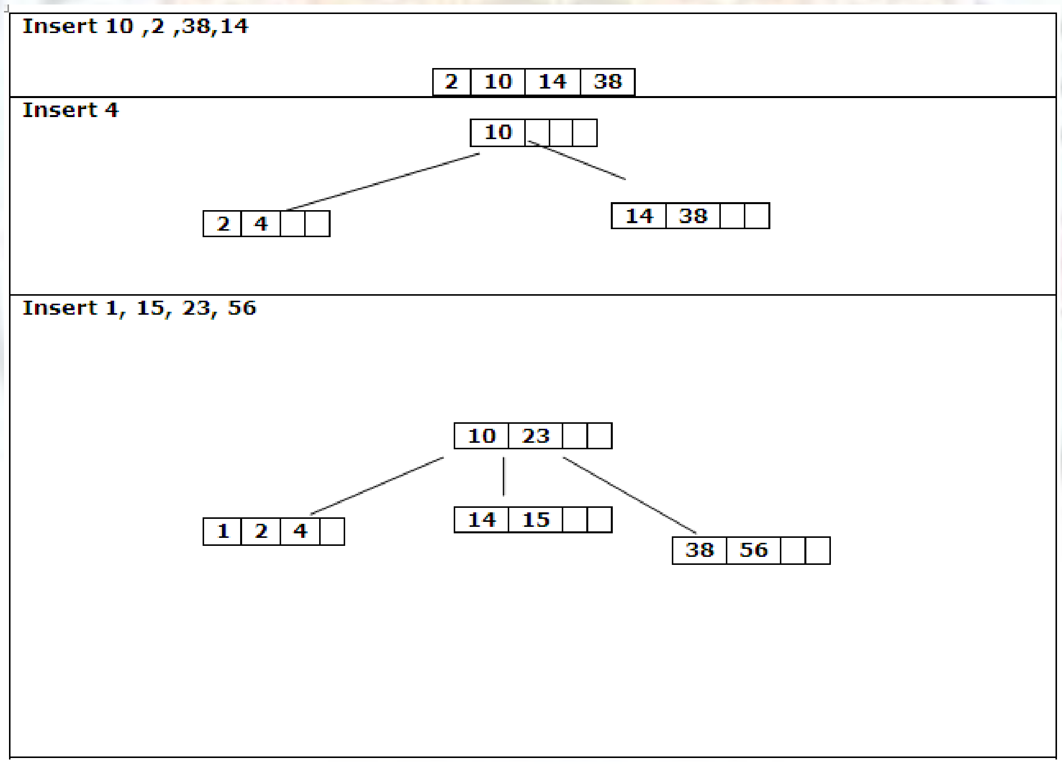
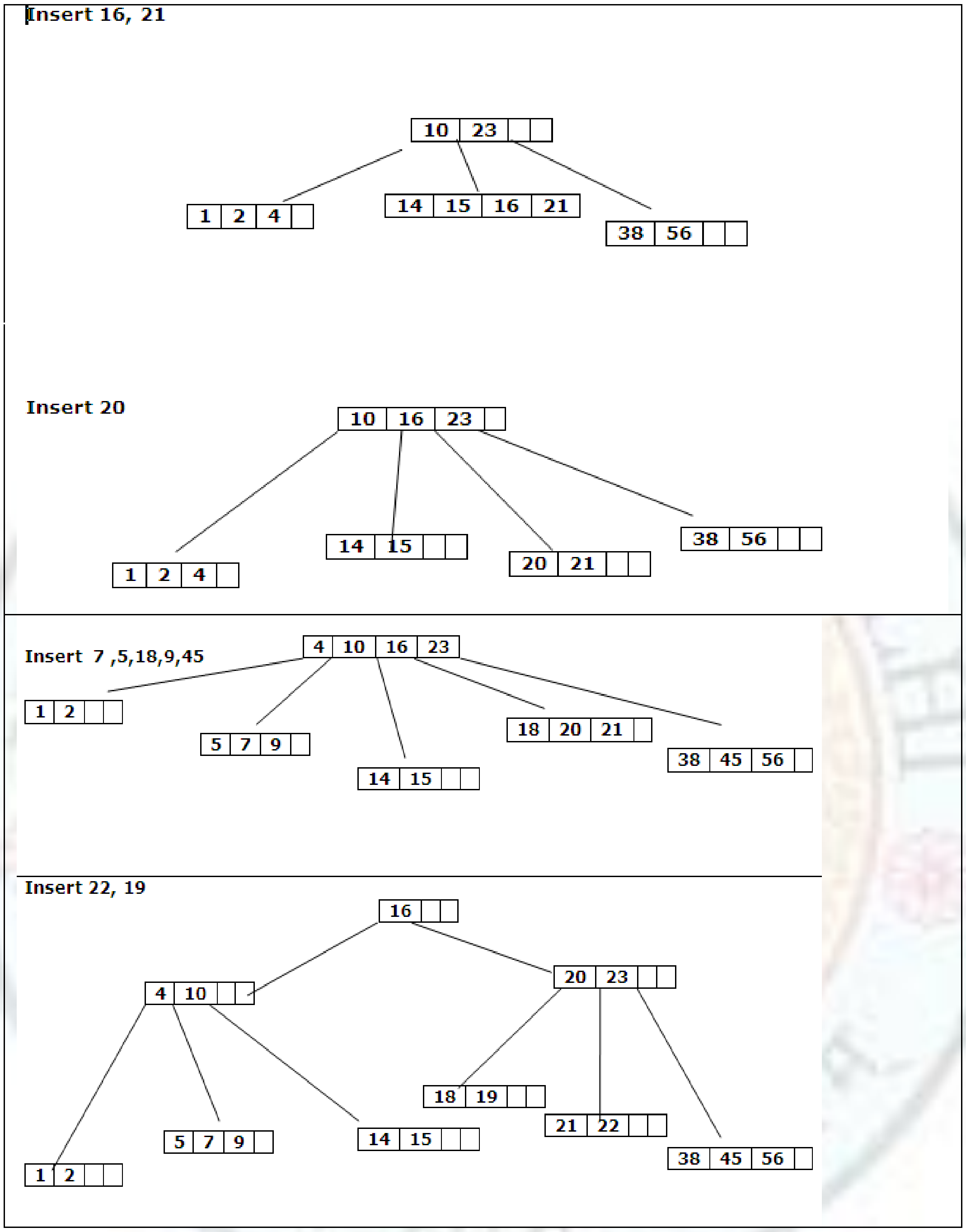


Fig 2.12(c)After Insertion stage 2

The above shown tree is fully balanced as all the leaves are on the same level and all the nodes except root node is more than half full.

2.3.2.1 Building a B-Tree from Scratch





So the procedure to insert a node can be written as :

To insert a value X in a B-Tree, there are three steps:-

1. *Using search procedure, find leaf node to which X should be added.*
2. *Add X to the node in appropriate place.*
3. *If there is M-1 or fewer values in the node after adding X, then finished.*
4. *If M values after adding X, then node is overflow, so we split the node into three parts.*

Left: first (M-1)/2 values

Right: Last (M-1)/2 values.

Middle Value: at (position 1 + (M-1)/2) goes into the parent node.

The above procedure is repeated for parent node if that also overflows.

Value Addition: Interesting Fact

Frequency of Node split in a B-tree

A split of the root node of a B-Tree creates two new nodes. All other splits add only one more node to the B-Tree. During the construction of a B-tree of p nodes, p-h splits have to be performed, where h is the height of the B-tree. Also in a B-tree of p-nodes, there are at least

$1+\left(\left\lceil \frac{m}{2} \right\rceil -1\right)(p-1)$ Keys. The rate of splits with respect to the number of keys in the B-tree can be given by

$$\frac{p-h}{1+\left(\left\lceil \frac{m}{2} \right\rceil -1\right)(p-1)}$$

After dividing the numerator and denominator by p-h and observing that $\frac{1}{p-h} \to 0$

And $\frac{p-1}{p-h} \to 1$ with the increase of p , the average probability of the split is

$\frac{1}{\left\lceil \frac{m}{2} \right\rceil -1}$. The larger the capacity of the node, the less frequently the split occurs.

Like if m=10 , probability is .25

If m=1000 ,it is .002

Source: Data Structures and Algorithms in C++ ,Adam Drozdek.

2.4 Deletion in B-Tree

Note: *While doing deletion of any key we have to be careful about the situation that the number of keys in any node should not become less than half full i.e. less than ceiling (m/2-1) as stated in step 3 of the definition.*

Two situations arise when deleting any key from a B-Tree

Situation 1: deleting a key from the leaf node.

Situation 2: Deleting a Key from the non leaf node.

Value Addition: Interesting Fact
B* Trees
A **B*-tree** is a tree data structure, a variety of B-tree used in the HFS and Reiser4 file systems, which requires non-root nodes to be at least 2/3 full instead of 1/2. To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with the node next to it. When both are full, then the two of them are split into three. It also requires the 'leftmost' key never to be used. The term is not in general use today as the implementation was never looked on positively by the computer science community-at-large. Most people use "B-tree" generically to refer to all the variations and refinements of the basic data structure.
Source: http://en.wikipedia.org/wiki/B*-tree

2.4.1 Situation 1 :Deleting a key from a leaf node

When we face the problem of deleting a key from the leaf node, actual physical deletion of the key take place and thereafter various cases appear as shown below.

Case 1 : if the leaf is at least half full after the deletion then only shifting of the left out keys is done as shown after deleting 72 in the following figure.

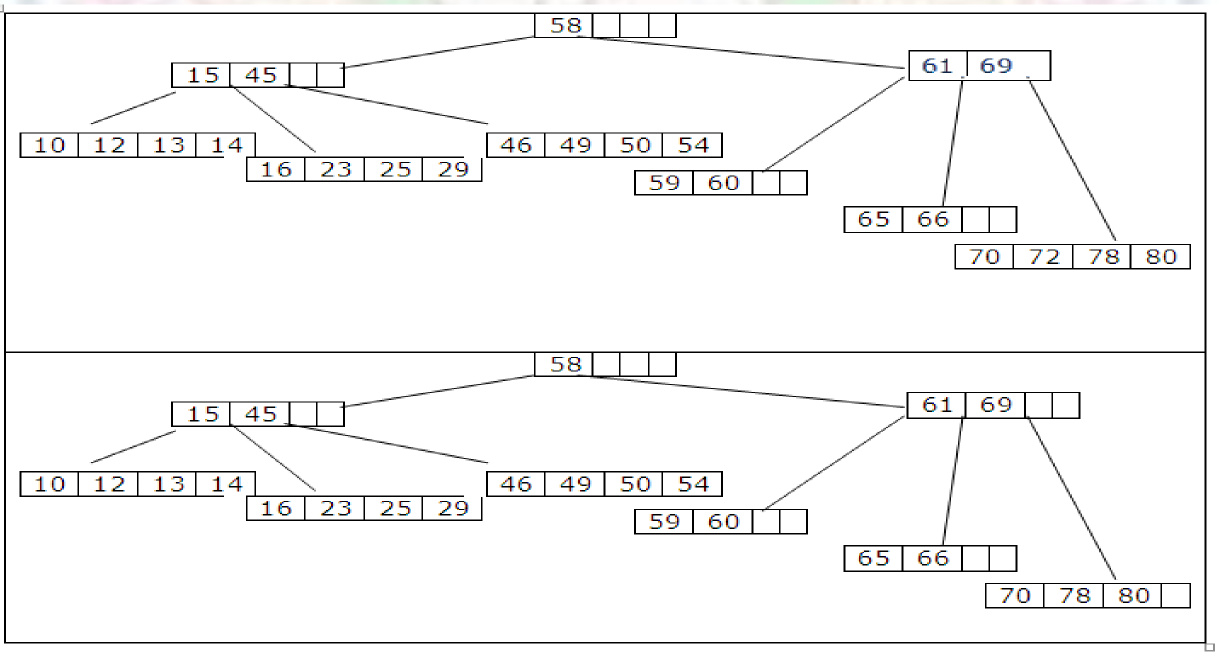


Fig 2.13 Deleting a Leaf from a Leaf

Case 2: If after deletion the leaf is less than half full i.e. the leaf underflows then we have to consider its nearby leaves for adjustments.

Adjustment 1: If there is left or right sibling leaf with keys >m/2-1, then all the keys from the under flown leaf, its sibling leaf and separator key of parent are redistributed, and one key from sibling is moved to parent as shown after deleting 66 from above B-tree.

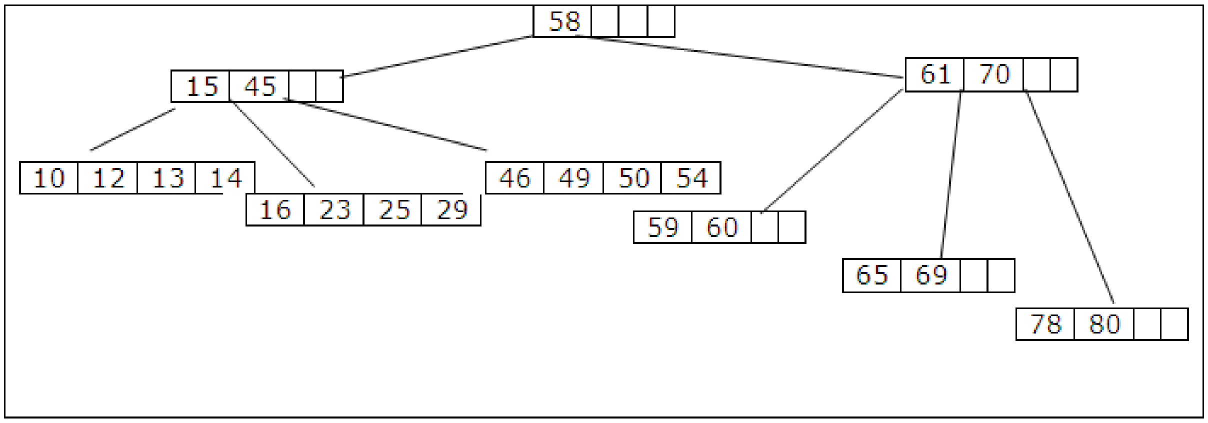


Fig 2.14 Adjustment 1

Adjustment 2: If there is left or right sibling leaf with keys =m/2-1, then all the keys from the under flown leaf, its sibling leaf and separator key of parent are put in one leaf and sibling leaf is discarded. See deletion of 69.

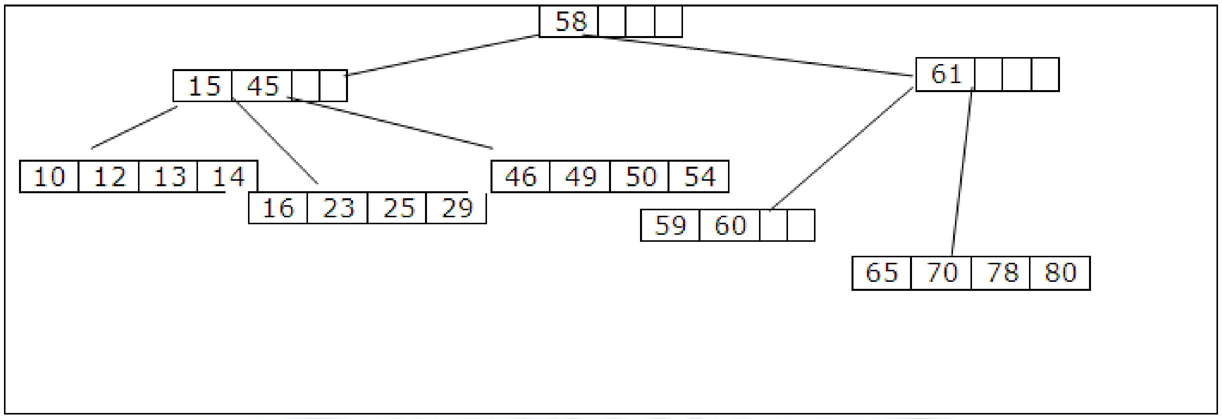


Fig 2.15 Adjustment 2

Now parent of the new node formed has one key so it has to be combined with its sibling and parents key as shown in the next adjustment.

Value Addition: Interesting Fact

B-Tree indexes in Data Bases

A B-tree index is good for a query that retrieves a range of data values. If the data to be indexed has a l
which the concepts of *less than*, *greater than*, and *equal* apply, the generic B-tree index is a useful way

Initially, the generic B-tree index supports the relational operators (<,<=,=,>=,>) on all built-in data ty
data in lexicographical sequence.

The optimizer considers whether to use the B-tree index to execute a query if you define a generic B-tre

- Columns used to join two tables
- Columns that are filters for a query
- Columns in an ORDER BY or GROUP BY clause
- Results of functions that are filters for a query

Extending a Generic B-Tree Index

Initially, the generic B-tree can index data that is one of the built-in data types, and it orders the data in
sequence. However, you can extend a generic B-tree to support columns and functions on the following

- *User-defined data types* (opaque and distinct data types) that you want the B-tree index to suppo

  In this case, you need to extend the default operator class of the generic B-tree index.

- *Built-in data types* that you want to order in a different sequence from the lexicographical sequen
  tree index uses

  In this case, you need to define a different operator class from the default generic B-tree index.

Source:
http://publib.boulder.ibm.com/infocenter/idshelp/v10/index.jsp?topic=/com.ibm.perf.d

Adjustment 3 : when parent of the node which is under adjustment with one key, then the node , its sibling and root are all combined to form a new node and B-tree height also get decreased here by 1. See in the following figure where node with value 61 is combined with its sibling keys and single key root to form a new root.

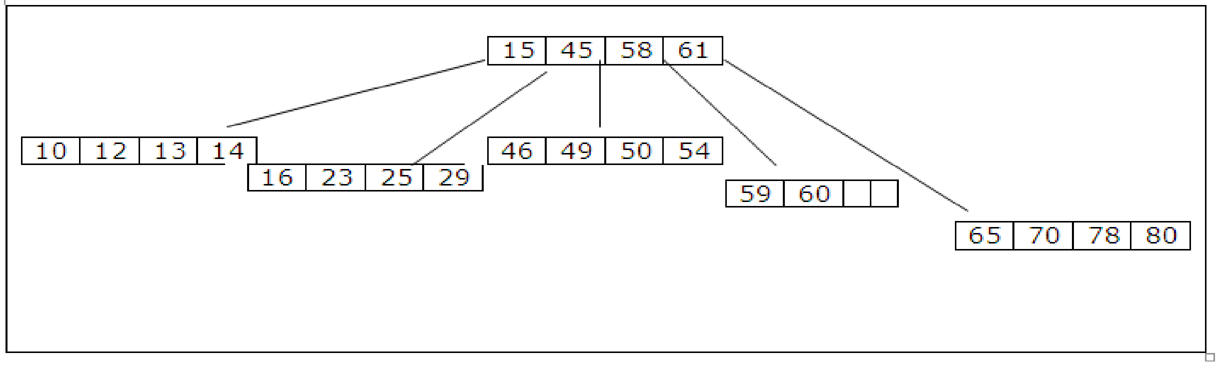


Fig 2.16 special case

2.4.2 Situation 2: Deletion from a non leaf node

We follow the strategy of deletion by copying i.e. we find the predecessor of the non leafy key (predecessor is present in one of the leaves) and replace the key to be deleted with its predecessor i.e. we copy the predecessor value to key location. Now we delete the predecessor from the leaf. Hence in this manner situation 1 is converted into situation 2.

Let us see how a key value 15 is deleted from the following B tree. The predecessor of 15 is 14 which is present in one of the leaf node. 14 is copied to the location where 15 is present and then 14 is deleted from the leaf node, hence converting situation 2 to situation 1.

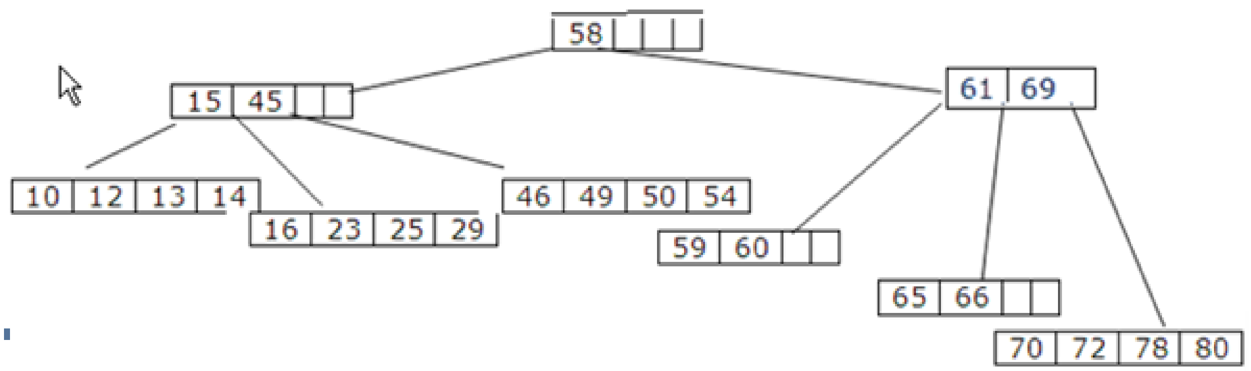


Fig 2.17 (a) Before deletion

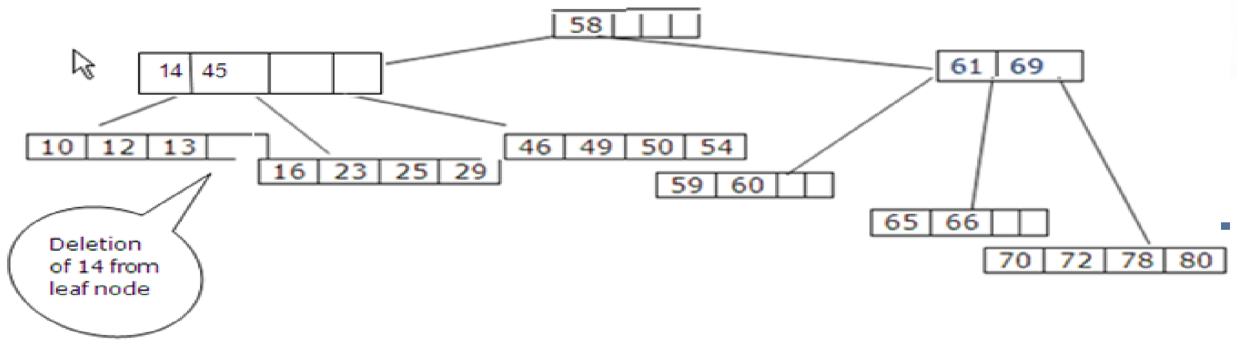


Fig 2.17(b) After deletion

Value Addition: Interesting Fact
B-Trees in File Systems
The B-tree is also used in file systems to allow quick random access to an arbitrary block in a particular file. The basic problem is turning the logical block address into a physical disk block (or perhaps to cylinder, head, and sector).   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. Converting to a physical block: the operating system just adds the logical block address to the starting physical 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 physical disk block and that entry identifies the next physical disk block of a file. The result is the disk blocks of a file are in a linked list. In order to find the physical address of block , the operating system must sequentially search the FAT. For MS/DOS, that was not a huge penalty because the disks were small and the FAT had few entries. In the FAT12 file system, there were only 4,096 entries, and the FAT would usually be resident. As disks got bigger, the FAT architecture confronts penalties. It may be necessary to perform disk reads to learn the physical address of a block the user wants to read.

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 word block, then the file directory would point to that physical disk block. If the file fit in 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 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 word file could be located in two disk reads and read on the third.

Apple Computer's file system and Microsoft's NTFS uses B-Tree.

Source: http://www.absoluteastronomy.com/topics/B-tree .

Value Addition: Interesting Fact

UB-trees for Multidimensional Applications

Classical B-trees were designed for one dimensional, linearly ordered key spaces. Here, B-trees are excellent for point queries and interval queries. But many applications are multidimensional, e.g. geographic maps, (2-dim), GPS data (3-dim, because of altitude in addition to latitude and longitude), or Data Warehouse (DWH) queries like asking for

- *the sales in a geographic area for a certain product group in a certain month* (4-dim).

Range queries in such spaces correspond to multidimensional rectangles, and the multidimensional points in those rectangles must be retrieved.

By a mathematical transformation, such multidimensional data spaces can be linearized and then represented in ordinary B-trees. The overall resulting data

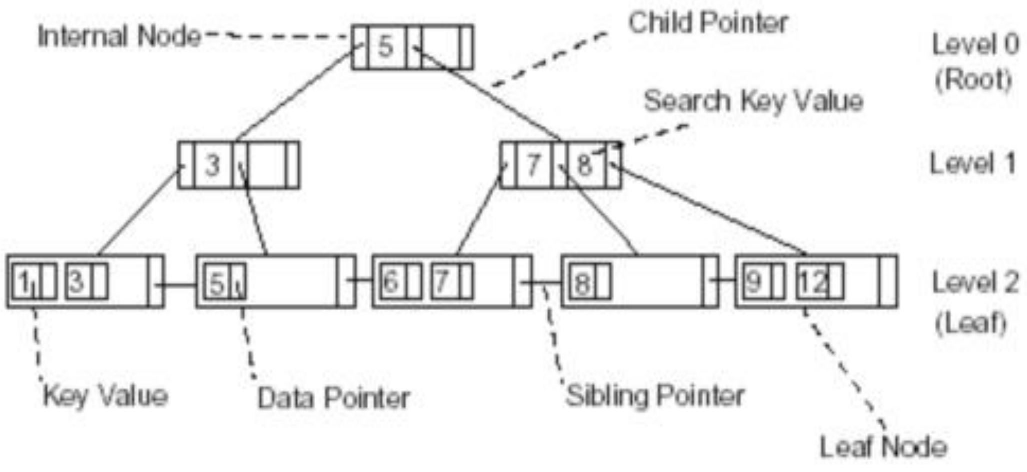
structure is called ***UB-tree*** for **Universal B-tree** (Markl et al 2000),(Website for UB-trees). The transformations used are space filling curves, and the linear order of the multidimensional keys is the order of the points on that curve.
Source: http://www.scholarpedia.org/article/B-tree_and_UB-tree .

2.5 B+ Trees: An Introduction

B+ trees are in extension to B trees. It has all the qualities of B trees with few more advantageous features.

- Similar to B trees, with few differences.
- The B + -Tree consists of two types of nodes (1) *internal nodes* and (2) *leaf node*s
- Internal nodes point to other nodes in the tree.
- Leaf nodes point to data in the database using *data pointer*s. The data is stored in the leaf nodes and all other nodes store the indexes.
- Leaf nodes are linked to each other using sibling pointer in sequential manner to form a linked list.
- Only leaf nodes needs to be traversed to scan the entire tree as data is present only in the leaf nodes without visiting the higher nodes at all reducing the block accesses to a great extent.
- Traversal is faster as compared to B trees in which the data is present in all the nodes which in turn would require more number if block accesses.
- Just like B trees, B+ trees are also balanced trees (every path from root node to leaf node has same length) and every node except the root must be at least half full. Root may contain a minimum of two entries.
- Keys may be duplicated; every key to the right of a particular key is > to that key and every key is to the left is <= key.
- Used for the purpose of indexing in relational databases.

Value Addition: Interesting Fact
Facts about internal nodes and leaf nodes in a b+ tree.



Internal Nodes

- An *internal node* in a B + -Tree consists of a set of *key values* and *pointer*s.
- The set of keys and values are ordered so that a pointer is followed by a key value.
- The last key value is followed by one pointer.
- Each pointer points to nodes containing values that are *less than or equal to* the value of the key immediately to its right
- The last pointer in an internal node is called the *infinity pointe*r. The infinity pointer points to a node containing key values that are greater than the last key value in the node.
- When an internal node is searched for a key value, the search begins at the Left most key value and moves rightwards along the keys.
- If the key value is less than the sought key then the pointer to the left of the key is known to point to a node containing keys less than the sought key.
- If the key value is greater than or equal to the sought key then the
- Pointer to the left of the key is known to point to a node containing keys between the previous key value and the current key value.

Leaf Nodes

- A *leaf node* in a B + -Tree consists of a set of *key values* and *data pointer*s.
- Each key value has one data pointer. The key values and data pointers are ordered by the key values.
- The data pointer points to a record or block in the database that contains the record identified by the key value. For instance, in the example, above, the pointer attached to key value 7 points to the record identified by the value 7.
- Searching a leaf node for a key value begins at the leftmost value and moves rightwards until a matching key is found.
- The leaf node also has a pointer to its immediate *sibling node* in the tree.
- The sibling node is the node immediately to the right of the current node.

Because of the order of keys in the B + -Tree the sibling pointer always points to a node that has key values that are greater than the key values in the current node.

Source: http://www.mec.ac.in/resources/notes/notes/ds/bplus.htm

Value Addition: Interesting Fact

Characteristics of B+ Trees.

Characteristics

For a *b*-order B+ tree with *h* levels of index:

- The maximum number of records stored is $n_{max} = b^h - b^{h-1}$
- The minimum number of keys is $$n_{kmin} = 2\left(\frac{b}{2}\right)^{h-1}$$
- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations in the worst case
- Finding a record requires $O(\log_b n)$ operations in the worst case
- Removing a (previously located) record requires $O(\log_b n)$ operations in the worst case
- Performing a range query with k elements occurring within the range requires $O(\log_b n + k)$ operations in the worst case.

Source: http://en.wikipedia.org/wiki/B%2B_tree

Value Addition: Interesting Fact

History of B+ Trees.

History

The B tree was first described in the paper *Organization and Maintenance of Large Ordered Indices. Acta Informatica 1*: 173–189 (1972) by Rudolf Bayer and Edward M. McCreight. There is no single paper introducing the B+ tree concept. Instead, the notion of maintaining all data in leaf nodes is repeatedly brought up as an interesting variant. An early survey of B trees also covering B+ trees is Douglas Comer: "The Ubiquitous B-Tree", ACM Computing Surveys 11(2): 121–137 (1979). Comer notes that the B+ tree was used in IBM's VSAM data access software and he refers to an IBM published article from 1973.

Source: http://en.wikipedia.org/wiki/B%2B_tree

Example of B+ Tree

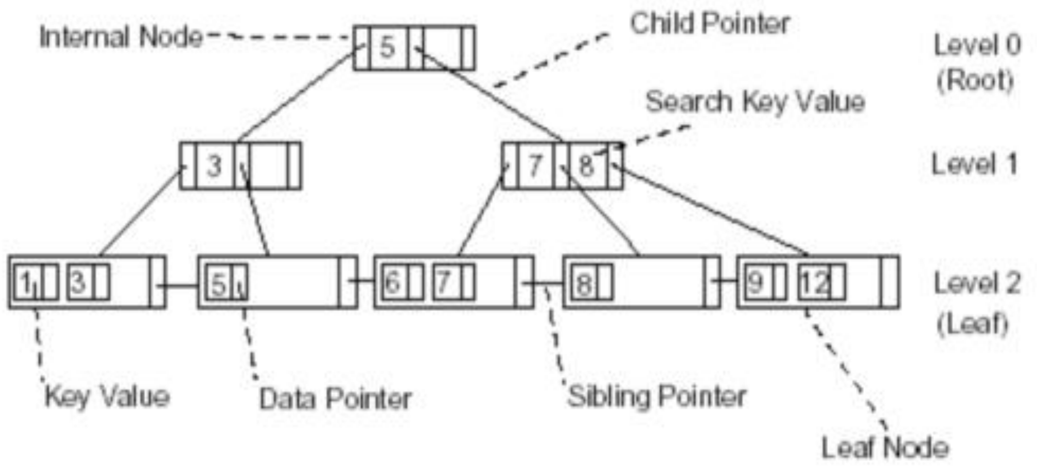


Fig 2.18

B+-tree considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially; the entire tree may be scanned without visiting the higher nodes at all.

We will see insertion and deletion in B + trees in the next section.

2.5.1 Insertion in B+ Trees

- Insert at bottom level
- If leaf node overflows, split it and copy middle element to next index page
- if index page overflows, split page and move middle element to next index page

Let us see an example if step by step insertion in a B+ tree from scratch. The B+ tree that we are considering is of order 3 that means a node in our tree can have at most three children and 2 keys.

Value Addition: Biography

Douglas Comer

Education and career

Dr Comer holds a BA in Mathematics and Physics from Houghton College earned in 1971 and a PhD in Computer Science from Pennsylvania State University earned in 1976.

He is a distinguished professor of computer science and professor of electrical and computer engineering at Purdue University in the US.

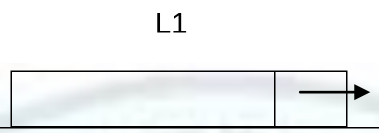
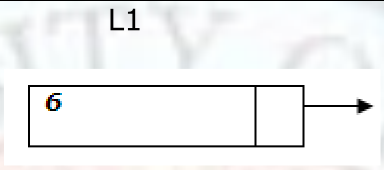
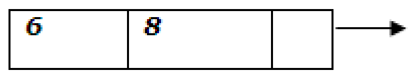
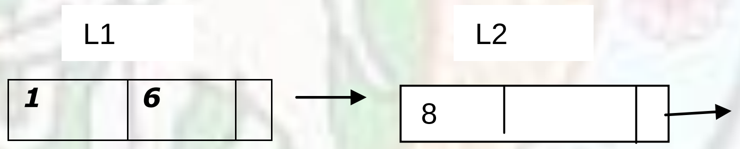
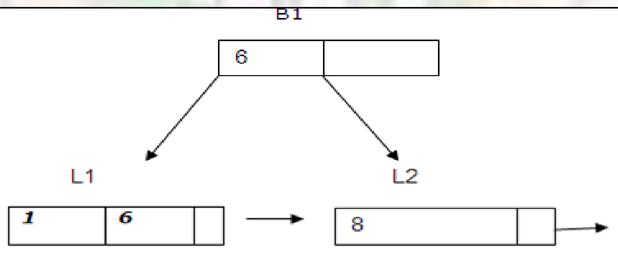
Comer came back to Purdue University in August 2009, prior to that he was acting as Vice President of Research for computer networking company Cisco Systems.

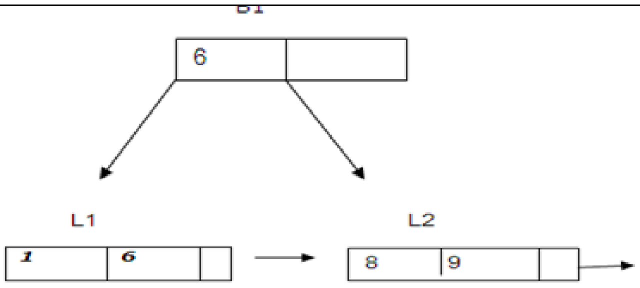
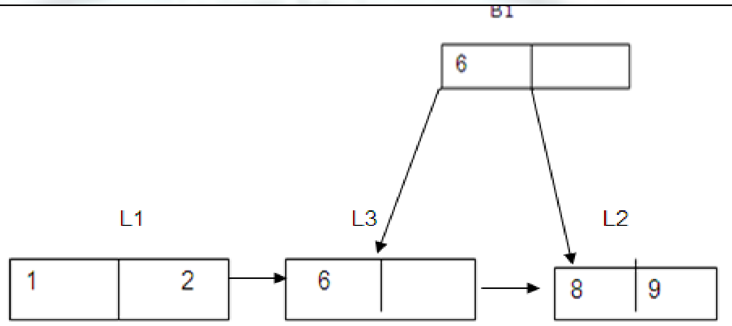
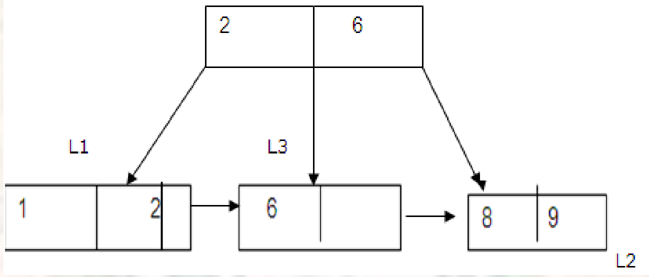
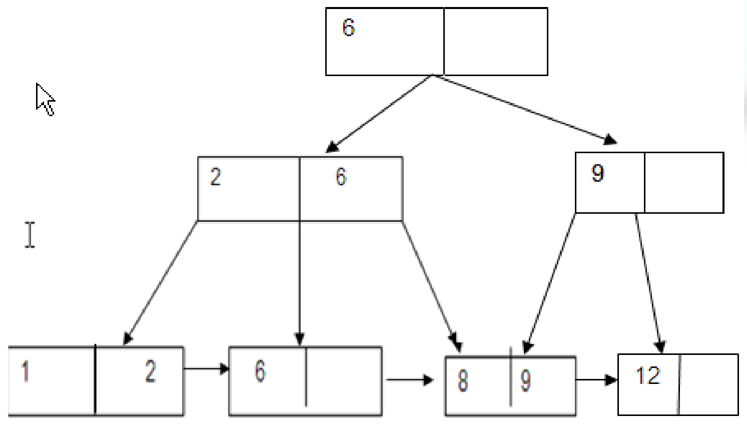
Achievements and publications

Douglas Comer headed a number of research projects associated with the creation of the Internet, and is the author of a number of books on the Internet and TCP/IP networking.

Dr Comer is a fellow of the ACM from 2000, as a recognition for his "work with IP-based networking supporting the modern Internet", as well as for his "contributions in research, education, and implementation in operating systems and networking"For twenty years, Professor Comer served as editor-in-chief of the research journal Software--Practice And Experience, published by John Wiley & Sons. Comer is a Fellow of the ACM and the recipient of numerous teaching awards.

Source: http://en.wikipedia.org/wiki/Douglas_Comer

The B+-Tree starts as a single leaf node. This leaf node is empty.	
Inserting Key Value 6: To insert a key search for the location where the key would be expected to occur. In our B+-Tree only *L1* is present which is empty. Hence, the key value 6 must be placed in leaf node *L1*.	
Inserting Key Value 8: search for the location where key value 8 is expected to be found. This is in leaf node *L1*. There is room in *L1* so insert the new key.	
Inserting Key Value 1: Searching for the key value 1 also results in *L1* but *L1* is now full. *L1* must be split into two nodes and keys to be redistributed.	
Create a new root node and promote the rightmost key from node *L1*.	

Insert Key Value 9: Search for the location where key 9 is expected to be located, that is, *L2*. Insert key 9 into *L2*.	
Insert Key Value 2: Search for the location where key 2 is expected to be found results in reading *L1*. But, *L1* is full and must be split.	
The rightmost key in *L1*, i.e. 2, must now be promoted up the tree.	
Insert Key Value 12 Search for the location where key 12 is expected to be found, *L2*. Try to insert 12 into *L2*. Because *L2* is full it must be split to l2 and L4. A new root of L2 and L4 is needed so rightmost value of L2 is promoted to new root named B2. Now B1 and B2 must have one root so rightmost value of B1 is promoted to form a new root which is 6.	

Value Addition: Frequently Asked Question

What are the major differences between B-Trees and B+trees?

- B+Trees are much easier and higher performing to do a full scan, as in look at every piece of data that the tree indexes, since the terminal nodes form a linked list. To do a full scan with a B-Tree you need to do a full tree traversal to find all the data.

B-Trees on the other hand can be faster when you do a seek (looking for a specific piece of data by key) especially when the tree resides in RAM or other non-block storage. Since you can elevate commonly used nodes in the tree there are less comparisons required to get to the data.

B+ Trees are especially good in block-based storage (eg: hard disk). with this in mind, you get several advantages, for example (from the top of my head):

- high fanout / low depth: that means you have to get less blocks to get to the data. with data intermingled with the pointers, each read gets less pointers, so you need more seeks to get to the data
- simple and consistent block storage: an inner node has N pointers, nothing else, a leaf node has data, nothing else. That makes it easy to parse, debug and even reconstruct.
- high key density means the top nodes are almost certainly on cache, in many cases all inner nodes get quickly cached, so only the data access has to go to disk.

One possible use of B+ tress is that it is suitable for situations where the tree grows so large that it need not fit into available memory. Thus, you'd generally expect to be doing multiple I/O's. Often it does happen that a B+ tree is used even when it in fact fits into memory, and then your cache manager might keep it there permanently. But this is a special case, not the general one, and caching policy is a separate from B+ tree maintenance as such.

Also, in a B+ tree, the leaf pages are linked together in a linked list (or doubly-linked list), which optimizes traversals (for range searches, sorting,

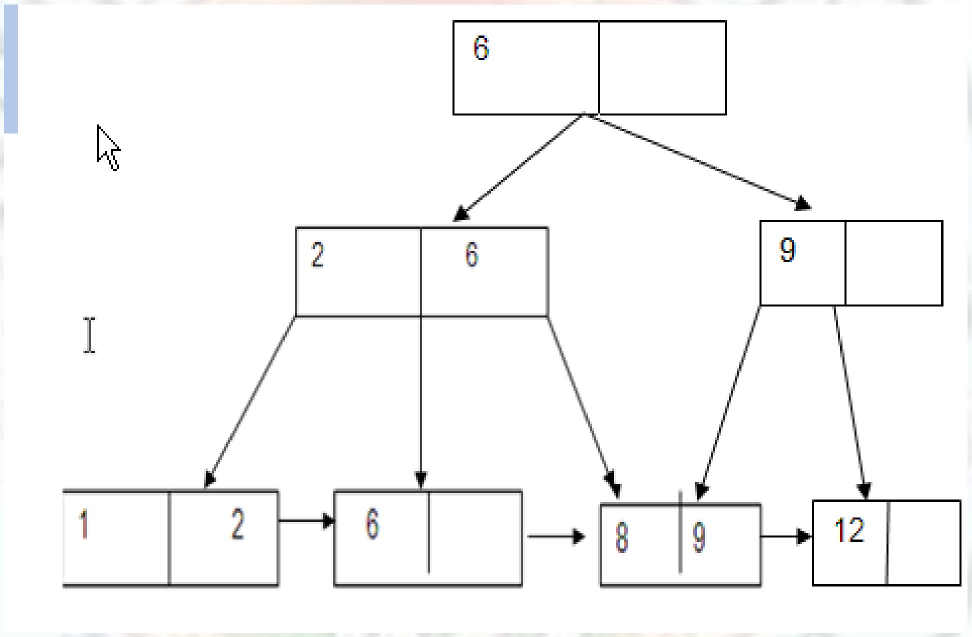
	etc.). So the number of pointers is a function of the specific algorithm that is used.
Source : http://stackoverflow.com/questions/870218/b-trees-b-trees-difference	

Value Addition: Frequently Asked Question

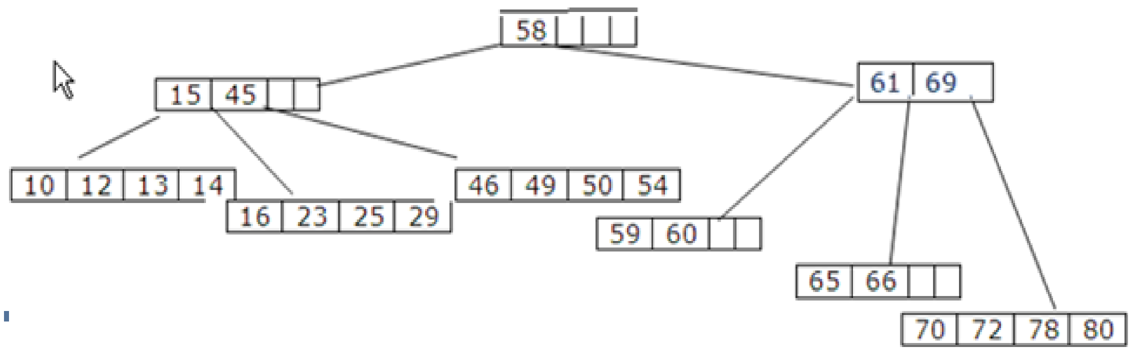
What is more often used B-Trees or B+ trees and why?

1.B-tree indices are similar to B +-tree indices.

- Difference is that B-tree eliminates the redundant storage of search key values.
- In B +-tree some search key values appear twice.



- A corresponding B-tree allows search key values to appear only once.



- Thus we can store the index in less space.

2. **Advantages of B tree:**
    - Lack of redundant storage (but only marginally different).
    - Some searches are faster (key may be in non-leaf node).
3. **Disadvantages of B Trees:**
    - Leaf and non-leaf nodes are of different size (complicates storage)
    - Deletion may occur in a non-leaf node (more complicated)

Generally, the structural simplicity of B +-tree is preferred.

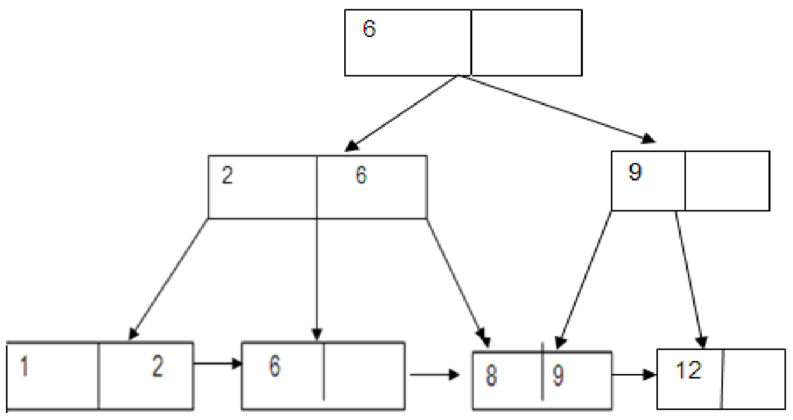
Source: **http://www.cs.sfu.ca/CC/354/zaiane/material/notes/Chapter11/node14.html**

2.5.2 Deletion in B+ Tree

Deleting entries from a B+-Tree may require some redistribution of the key values to guarantee a well-balanced tree.

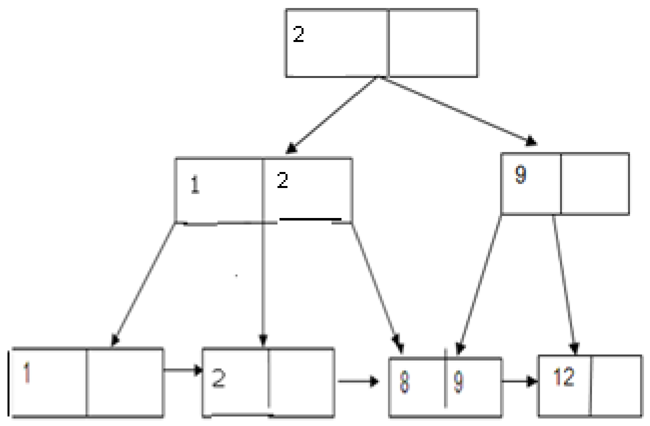
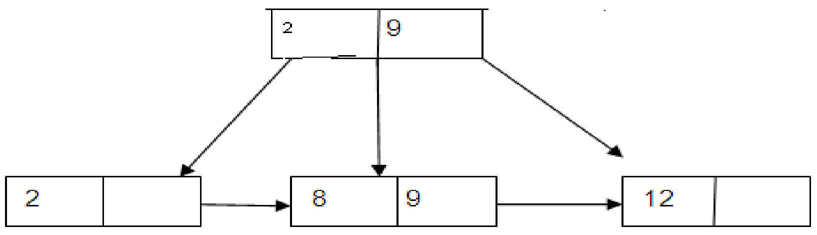
Procedure:

1. Start at root, find leaf L where entry belongs.
2. Remove the entry.
    a. If L is atleast half full then done.
    b. If L has less entries, try to redistribute, borrowing from sibling.
    c. If redistribution fails merge L and sibling.
3. If merge occurred must delete entry from parent of L
4. Merge could propagate to root decreasing height.



Consider the above tree for deletion operation.

Deletion sequence: 6, 1.

Delete Key Value 6 First, search for the location of key value 6, Delete 6.but now the leaf is less than half full. So we will redistribute the keys from its sibling leaf as it has more than half number of keys, and accordingly adjust the index node to reflect the deletion effect.	
Delete Key Value 1: *Search* for key value 1. Deleting 1 from the leaf will make it empty; also its sibling leaf is half full only, so keys can't be redistributed. Now we will remove the leaf also and adjust the index nodes accordingly.	

Value Addition: Frequently Asked Question

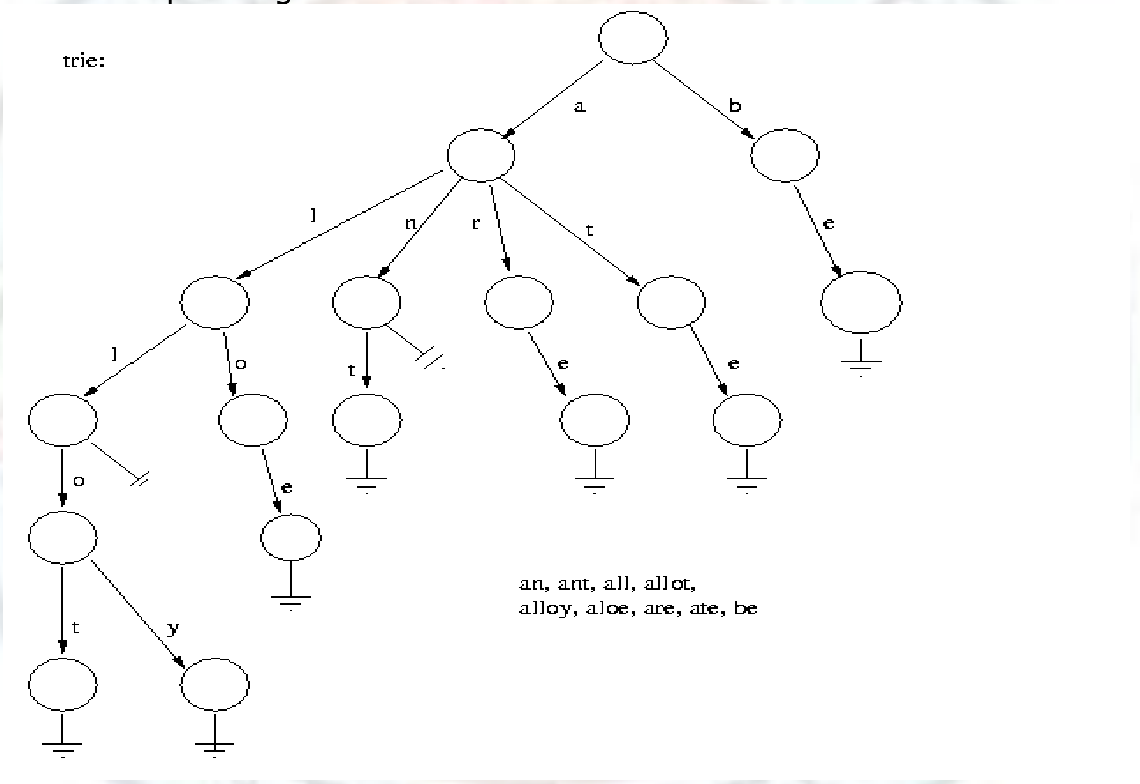
In which situations trie and B+ trees are used ?

Introducing Trie

A *trie* (from retrieval), is a multi-way tree structure useful for storing strings over an alphabet. It has been used to store large dictionaries of English (say) words in spelling-checking programs and in natural-language "understanding" programs. Given the data:

an, ant, all, allot, alloy, aloe, are, ate, be

the corresponding trie would be:



The idea is that all strings sharing a common stem or *prefix* hang off a common node. When the strings are words over {a..z}, a node has at most 27 children - one for each letter plus a terminator.

The elements in a string can be recovered in a scan from the root to the leaf that ends a tring. All strings in the trie can be recovered by a depth-first scan of the tree.

Trie Vs. B+ Trees

It depends on what you mean by Range.

If your range is expressed as All words beginning by, then a Trie is the right choice. On the other hand, Trie are not meant for requests like All words between XX and ZZ.

Note that the branching factor of the B+ Tree affects its performance (the number of intermediary nodes). If h is the height of the tree, then nmax = bh. Therefore h = log(nmax) / log(b).

With n = 1 000 000 000 and b = 100, we have h = 5. Therefore it means only 5 pointer dereferencing for going from the root to the leaf. It's more cache-friendly than a Trie.

Finally, B+ Tree is admittedly more difficult to implement than a Trie: it's more on a Red-Black Tree level of complexity.

Source: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Trie/

Summary

It is recommended that data should be organized to minimize the number of successive disk accesses. It is better to access large amount of data at one time than to jump from one position on the disk to another to access small pieces of information. This is the reason why M-way trees are the best when the question is of data storage on the disk. M-way trees store multiple keys in one node and they can be of the size of the whole block itself, which will help us to get maximum information in one block access. The drawback of M-way search trees is that they are unbalanced. So advancements have been done to use its good features and make it balanced also.

B-Trees are Balanced M-way trees as the path length from root to any leaf is same for all the leaves. Also it keeps the data in sorted order. The nodes at any given time can’t have less than half of the number of keys which speeds up insertion and deletion operation on the B-Trees .All these operations take logarithmic time on B Trees. We have seen the step by step insertion and Deletion in the B-Trees. It is a well suited data structure for large databases which are stored on Hard Disks.

B+ Trees are an extension to B-Trees. It has all the qualities of B-trees with some more advantageous features. Due to it’s arrangement of data in the leaf nodes only traversal is much faster here. It distinguishes between leaf nodes (which contain only data) and internal nodes (which contain indexes to data present in the leaves) as data nodes and index nodes respectively. It is used for the purpose of indexing in relational databases. We have seen in detail insertion and deletion operations on B+ Trees also.

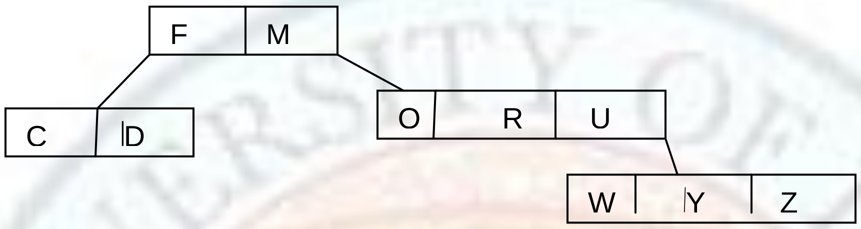
M-Way Trees And B-Trees

2.1: Construct a 3-way search tree for the list of keys in the order shown below. What are your observations?

List A: 10 , 15 ,20 ,25 , 30 , 35, 40 ,45

List B: 20, 35, 40, 10, 15, 25, 30, 45

2.2: In the following 4-way search tree trace the tree after deletion of (i)U and (ii) M



2.3: Show how a B- Tree and B+ Tree can be used to implement a priority queue. Show that any sequence of n insertions and minimum deletion operations can be performed in O (nlogn) time.

2.4: How many different 2-3 trees containing the integers 1 through 10 can you construct? How many permutations of these integers result in each tree if they are inserted into an initially empty tree in permutation order?

2.5: Start with an empty 2-3 tree and insert the keys 20, 40, 30, 10,25, 28,27,32,36,34,35,8,6,2 and 3 in this order. Draw 2-3 trees at each step.

2.6: Delete 34, 40, 3 ,36,28 from the above generated tree.

2.7: What is the maximum number of disk accesses needed to delete an element that is in a non leaf node of a b-tree of order m?

2.8: what are major differences between Binary tree, m-way tree, b-tree and B+ trees?

2.9: B+ trees are advantageous in which situations? Justify

Balanced trees :	B-tree is kept balanced by requiring that all leaf nodes are at the same depth or the height difference between any two leaves is at most 1.
Order of the tree :	Number of maximum children allowed for any node in a

	tree is order of the tee
Self-balancing Binary search Trees:	is any node based binary search tree data structure that automatically keeps its height (number of levels below the root) small in the face of arbitrary item insertions and deletions
AVL tree :	An AVL Tree is a self balancing Binary search tree, and it was the first such data structure to be invented. In an AVL tree, the heights of the two child sub trees of any node differ by at most one; therefore, it is also said to be height-balanced
Height of Binary search trees:	Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. Since a binary tree with height *h* contains at most $2^0+2^1+\cdots+2^h = 2^{h+1}-1$ nodes, it follows that the minimum height of a tree with *n* nodes is $\log_2(n)$, rounded down; that is, $\lfloor \log_2 n \rfloor$. $n \leq 2^{h+1}-1$ implies $h \geq \lceil \log_2(n+1)-1 \rceil \geq \lfloor \log_2 n \rfloor$
Analysis of an algorithm	To determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Suggested Readings

1. Data structures and algorithm in C++, Adam Drozdek,Publisher : Course Technology

2. Data Structures using C and C++,Langsam,Augestein and Tanenbaum ,Second Edition, Prentice- Hall, India

3. Abstract data types: specifications, implementations, and applications By Nell Dale, Henry M. Walke

4. Data Structure Algorithms and Applications in C++ , Sartaj Sahni, Second Edition, Mcgraw-hill

5. Data structures, Lipschutz,(s) and Pai (G.A.V.),Tata MacGraw-Hill Publishing Company limited.

Web links

1.1 http://datastructures.itgo.com/trees

1.2 http://www.brpreiss.com/books/opus5/html

1.3 http://Wikipedia.org

1.4 http://www.bluerwhite.org/btree/

1.5 http://www.mec.ac.in/resources/notes/notes/ds/bplus.htm for b+ tree insertion and deletion.