File Organisation

A file organisation refers to the organisation of the data of a file into records, blocks and access structures. There are many file organisations as follows :

- sequential
- relative
- indexed sequential

The organisation most appropriate for a particular file is determined by the following :

- operational characteristics of the storage medium used
- nature of the operations to be performed on the data

The most important characteristic of a storage device that influences selection of a file organisation technique is whether the device allows **direct access** to a record without accessing all physically prior record occurrences that are stored on the device or allows only **sequential access** to record.

The other aspect i.e. nature of operation is important in determining the file organisation. A file can be accessed by a program that executes in **batch mode** or by a program that executes **interactively**. Programs developed for 'salary printing' of an organisation may be considered as an example of batch mode processing whereas a typical example of programs executing in interactive mode is query answering in railway reservation system.

File Operations

The fundamental operations that are performed by the application developer on files are the following :

- creation
- updation
    - insertion
    - deletion
    - modification
- retrieval
    - enquiry
    - report generation
- maintenance
    - restructuring
    - reorganisation

Creation

The bulk of the work in creating files involves data collection and validation. In some implementation, space is first allocated to the file, the data are loaded into the skeleton. In other implementations, the file is constructed a record at a time. The contents of a file represent a snapshot (at one instance of time) of the part of the real world that the file represents.

Updating

Changing the contents of a file to make it reflect a more current snapshot of the real world is known as updating the file. As mentioned earlier, there are three types of updation namely i**nsertion**, **deletion** and **modification**. Here insertion means adding a new record to a file and deletion means removing an existing record. Modification means changing an existing record.

Retrieval

The access of a file for purposes of extracting meaningful information is called retrieval. There are many types of retrieval of operations. They may be grossly divided into two namely enquiry type and report generation type. This classification is based on modes of processing of files. If a retrieval is processed interactively, the access is enquiry type. On the other hand, if a retrieval is processed in batch mode, the retrieval is called report generation type.

From another point of view a file retrieval request can be comprehensive or selective. Comprehensive retrieval reports information from all the records on a file whereas selective retrieval applies some qualification criteria to choose which records will supply information for output.

Maintenance

Changes that are made to files to improve the performance of the programs that access them are known as maintenance activities. There are two basic classes of maintenance operations such as restructuring and reorganisation. Restructuring a file implies that structural changes are made to the file within the context of the same file organisation. For example, field widths may be changed or a new field may be inserted. Reorganisation implies a change from one file organisation to another.

From the application point of view, the commonly used files are transaction files, master files, report files, work files, program files etc. In general, master files and program files are created, updated, retrieved from and maintained. Transaction files are generally created and used for one-time processing. Work files are created, updated and retrieved from but are not maintained. Report files generally are not updated, retrieved from or maintained.

Heap / Sequential / Unordered files

In the simplest and most basic type of organisation, records are placed in the file in the order in which they are inserted and new records are inserted at the end of the file. Such an organisation is called a heap file. Some file system such as Digital Equipment Corporation's VAX RMS call this organisation a sequential file. As the records are placed as they arrive it does not maintain any order. Hence some times it is called unordered file.

Insertion

The last disk block of the file is copied into a buffer ; the new record is added and the block is then rewritten back to disk.

Look up

Searching for a record using any search condition involves a linear search through the file block by block. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record.

Deletion

To delete a record first task is to find out the data. If the data is found, the block is copied into a buffer. Next, the record is deleted from the buffer. Finally, the block is rewritten back to the disk. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an extra deletion marker stored with each record. A record is deleted by setting the deletion marker to a certain value. Both of these deletion techniques require periodic reorganisation of the file to reclaim the unused space.

Sequential/ Ordered files

In this file records are placed sequentially onto the storage media. In addition, the records in the file are usually ordered according to the values of key attributes. The following are the advantages of ordered files over unordered files :

- reading the records in order of the ordering field values becomes extremely efficient, since no sorting is required.
- finding the next record from the current one in order of the ordering field usually requires no additional block.
- using a search condition based on the value of an ordering field results in faster access when the binary search technique is used.

Direct access files

One of the direct access file organisation is based on hashing, which provides very fast access to records on certain search conditions. This organisation is called hashed file organisation. The idea behind hashing is to provide a function h called a hash function that is applied to the hash field value of a record and yields the address of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single block access to retrieve the record.

Normally relative address is calculated by hashing. Absolute addresses are device dependent. Should it be desirable to upgrade or change the device upon which the file resides, it is likely that key values would also need to be changed.

Internal Hashing

When data remain stored in memory and hashing is applied, it is called internal hashing. There are number of methods of hashing as follows :

- division-remainder hashing
- mid-square hashing
- folding

division-remainder hashing

The basic idea of this approach is to divide a key value by an appropriate number, then to use the remainder of the division as the relative address for the record. Here the choice of the appropriate divisor may not be quite so simple. There are several factors that should be considered in selecting the divisor.

- The range of values that result from the operation **key mod div** is 0 through div - 1. Thus the value of div determines the size of the relative address space. If it is known that the relative file is going to contain at least n records, then we must have div > n assuming that only one record can be stored at a given relative address.

- The divisor should be selected such that the probability of collision is minimised, a difficult objective to achieve. So we try to select a divisor that keeps the probability of collision relatively low.

Some research result suggests the following :

- divisor should be a prime number; reduces skewed results.

- select a divisor that does not contain any prime factor less than 20 ; this is probably sufficient to ensure good performance.

- regardless of how ‘good’ the divisor is, when the address space of a relative file gets full enough , the probability of collision rises dramatically.

Fullness of a relative file is measured by its load factor as shown below :

$$\text{load factor} = \frac{\text{\# records in file}}{\text{max. \# of records file can contain}}$$

As a general guideline, a load factor of 0.7 or 0.8 is about as great as can be tolerated with reasonable performance. For example, consider design of a relative file that will contain about 4000 records. The address space should accommodate at least 5000 records for 80% loading. A number that is approximately 5000 and that does not contain any prime factors less than 20 is 5003. For key value 123456789 relative address is 2762.

Mid-square Hashing

In this technique, the key is squared, then specified digits are extracted from the middle of the result to yield the relative address. If a relative address of n digits is desired, then digits are truncated at both ends of the squared key, leaving n digits from the middle. The same n digit positions must be extracted for each key. For the key values 123456789 & 987654321, the following are the respective squared values and relative addresses.

15241578750190521	8750
975461055789971041	5789

For example to accommodate 4000 records, 4 digit relative addresses are required. The total accommodation is 10000 and load factor is 0.4. Using this hash function the size of the file is 10^n where n is the number of digits extracted from the squared key values.

Folding

Comparison of Hash Functions

No hash function always performs better than all others. The mid-square method can be applied to files with fairly low loading factors to give generally good performance, but sometimes it can generate poor performance with many collisions. The folding method may be the easiest technique to compute, especially if bit patterns rather than decimal values are used, but it produces quite erratic results, unless the key length is approximately the same as the address length. If the distribution of key values is not known, then the division remainder method is the preferred hashing technique. Thus it is very commonly used.

It is to be noted that the hash algorithm acts upon the binary representation of the characters; hence hashing can be applied to non-numeric keys as well.

Other hash techniques have been developed that allow the file to grow or shrink without forcing the relocation of old records. These schemes are known as dynamic hashing, extendible hashing and virtual hashing.

Approaches to the problem of collision

Because a hashing function maps a relatively large key-value space to a relatively small address space, collision occurs.

Numerous techniques have been developed to handle collision. The most common approaches are mentioned below :

- Linear probing
- Double hashing
- Synonym chaining
- Bucket addressing

Bucket addressing

In this method, hashing is performed to blocks/ buckets of space that can accommodate multiple record occurrences rather than hashing to individual record slots. For example, consider a relative address space of 0 to M and a bucket size of B records. The address space thus can contain B(M+1) records. If the file contains N records, the load factor is –

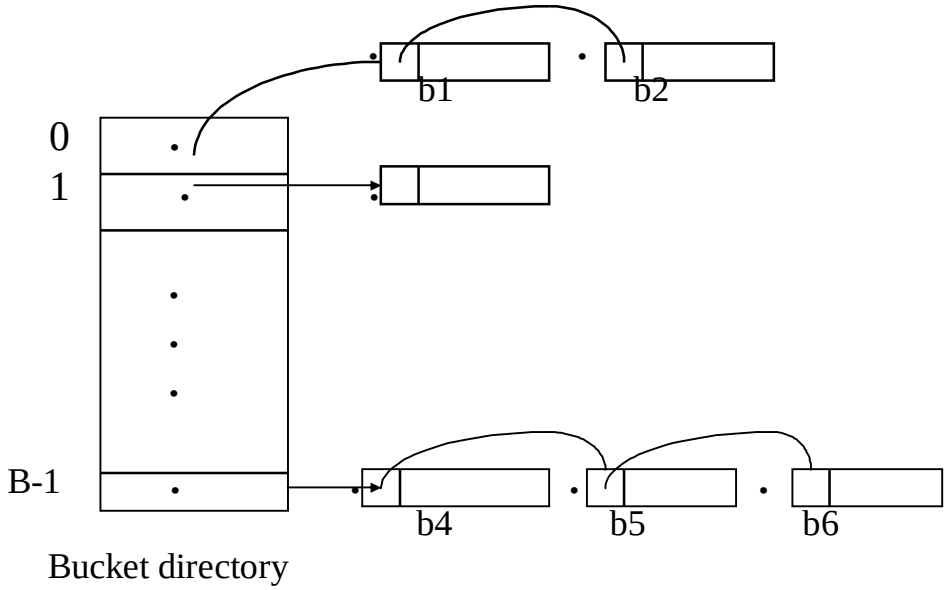
$$\frac{N}{\text{----------}}$$

B(M+1)

B records can all hash to the same relative address without causing a problematic collision. When a bucket overflows, some new location must be found for the offending record.

The size of bucket may be determined by the size of a track or of a sector on a disk or by the system's natural page size. Bucket size generally should be the same as block size for the file.

Hashed file organisation using bucket addressing



Hashed File Organization

h hashes v i.e. h(v) takes all its possible values with roughly equal probability as v ranges over likely collections of values for the key.

Look-up we compute h(v) which gives us a bucket number. We consult the first block of that bucket. Next search each non empty sub block in the block to see if it holds a record with key value v. If it is not found and the header of the block has a pointer to further

blocks in that bucket, search each of these blocks in turn until either the record with key value v is found, or the last block of the chain of blocks for the bucket has been searched.

Modification If the field to be modified is part of the key, this modification will be deletion followed by an insertion. Otherwise modification is simple 'Change'.

Insertion If same key value is found using look-up procedure there is an error ; otherwise the empty sub block in the block for bucket h(v) is the place for insertion. If no empty sub blocks exist in all the blocks of the bucket h(v), the file system is called upon to provide a new block.

Deletion After finding the record (to be deleted)by look-up procedure, deletion is made by making the sub block empty by setting its full/empty bit in the header.

Block Access

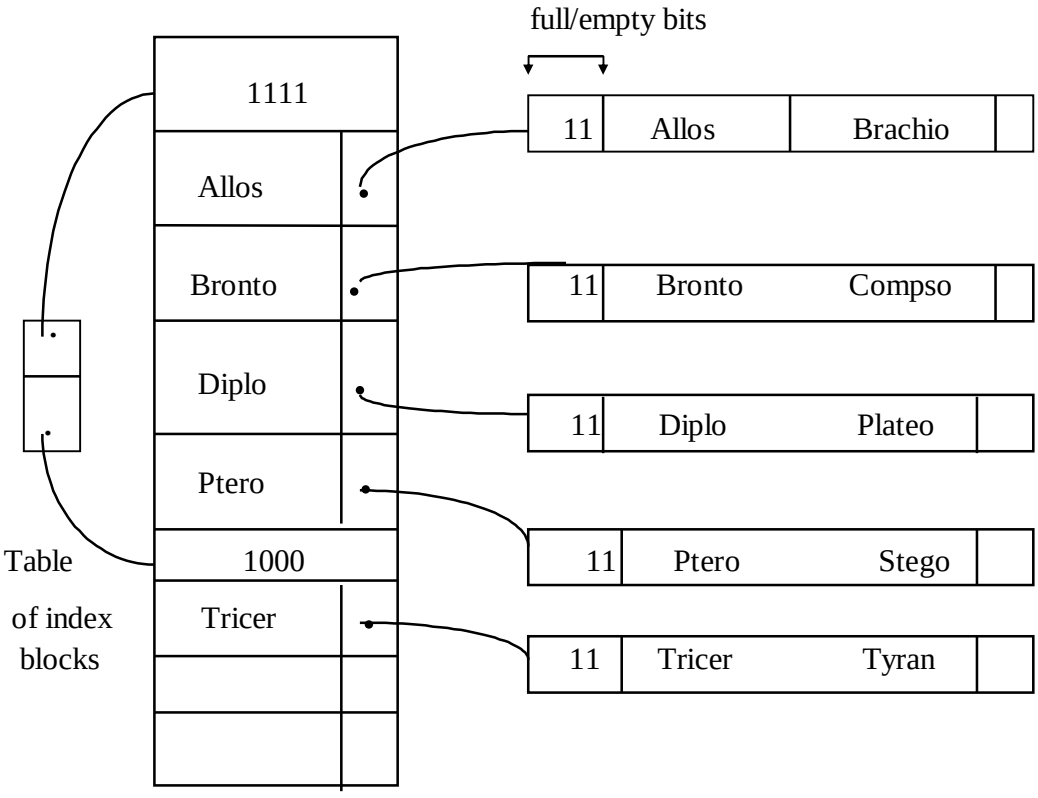
If each bucket consists of one block, the look-up takes two block access and three for other operations independent of the size of the file.

As the file grows it will be necessary on occasion to reorganize the file by changing the hash function and increasing the size of the bucket directory.

Disadvantage Range queries cannot be answered easily.

Indexed sequential file organisation

So far we discussed techniques for organising data where the fundamental concept is to use the key value in calculating the record's physical location. Here additional techniques for providing rapid access to a record in a file is discussed. As an additional aids augmented binary search tree is used. The augmentation is done by including pointers (address) to data records. This augmented binary tree is called an **index**. An index is a structured collection of key value and address pairs; the primary purpose of an index is to facilitate access to a collection of records. An index is said to be a **dense index** if it contains a key value-address pair for each record in the collection. An index that is not dense is called **sparse index**.



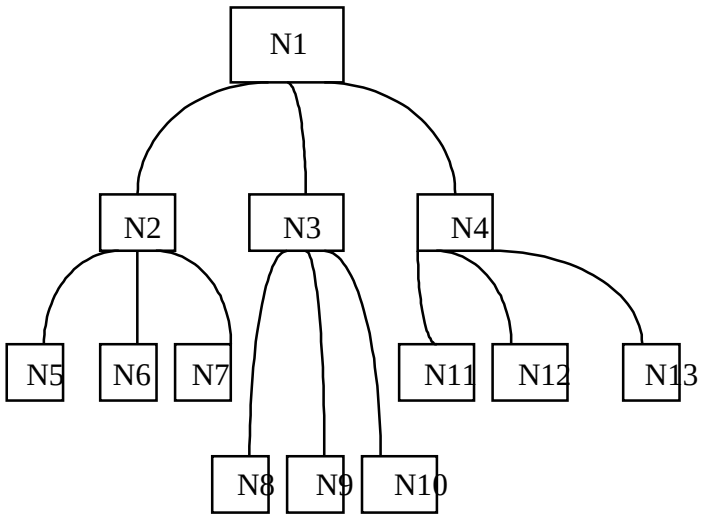
Indexed File

Look up Examine the index file to find the block whose first record has a key value v2 that covers v1. Search this block for a record with the key v1.

Insertion Let us consider to insert “Elas”. Searching index file it is found that “Elas” comes between “Diplo” and “Plateo”. We therefore move the “Elas” record into the sub block that hold the “Plateo” record, and the latter record becomes excess. Locate the fourth block through the index file and find that the fourth block is also full. Therefore, we have to create a new block, placed between third and fourth blocks in the order initially holding only the “Plateo” record.

Modification For the change of name of “Bronto” as “Apato”, find the “Bronto” record through the index and delete it from the second block, moving the “Compso” record left and setting the full/empty bits to 10. Next modify the record in the index file for block 2 of the file by changing its key value from “Bronto” to “Compso”.

Let us add the “Apato” record. We find “Apato” is covered by “Allos” in the index file. So through the first block of the index file insert the “Apato” record in the second sub block and the record “Brachio” becomes excess. As there is room in the next block, and we insert it ahead of “Compso” which is moved back to where it began. We modify the index record for the second block changing its key value to “Brachio”.



B-tree

A B-tree is a tree structure with an unspecified number of levels having ‘balanced feature’.

Essential properties of B-tree

a) Each block can accommodate a fixed number, exceeding two, of records. In each index block the first record, which lacks a key value, is counted as one of the records in the block. All index block have the same capacity i.e. they all have rooms for the

same number of records. All data blocks have equal capacity which may differ from the capacity of index blocks.

b) No block, except the root index block may ever be less than half-full.

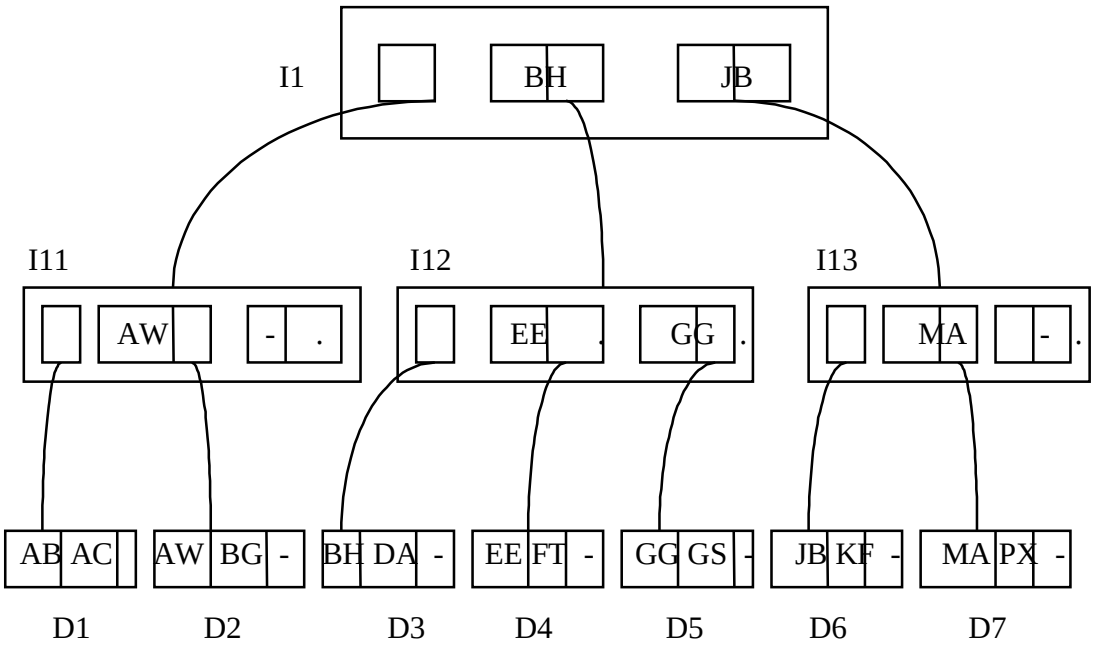
c) A B-tree is always balanced i.e. all path from root to leaf node should have same length.

Definition

A B-tree of order m is an m-way tree in which the following points are applicable :

1. Every node has ≤ m children
2. Every node, except the root and the terminal nodes has ≥⌈ m/2 ⌉ children
3. The root has at least 2 children unless it is a terminal node.
4. All terminal nodes appear on the same level and carry no information.
5. An internal node with k children contains k-1 key values.

The following is a B-tree.



It shows a B-tree in which D1, D2, ….,D7 are data blocks sorted on the primary key. The symbol '-' to represent an absent record and '.' to represent a nil pointer . Blocks I11, I12, I13 are index blocks that point to data blocks. These index blocks are pointed to by another index block I1 which is the root block.

In a B-tree it is convenient to make the index show the lowest key value in each block. For the first block to which an index points to, there is no benefit in showing the lowest key value in the first block, because any record that has key value less than the first in the second block should be inserted in the first block, regardless of the current lowest key value in the first block.

Look-up

To look up a given key value, we always begin by linearly searching the root block until we find the first key value greater than the sought key value or until we reach the end of the block. We then follow the pointer that is contained in the index record immediately before the record where the search stopped. If the search stopped at the end of the block, we follow the last pointer in the block. If the ptr. takes us to a further index block, we search it in the same way to select a ptr. to its descendent. When we arrive at a data block we linearly search it for the sought key value.

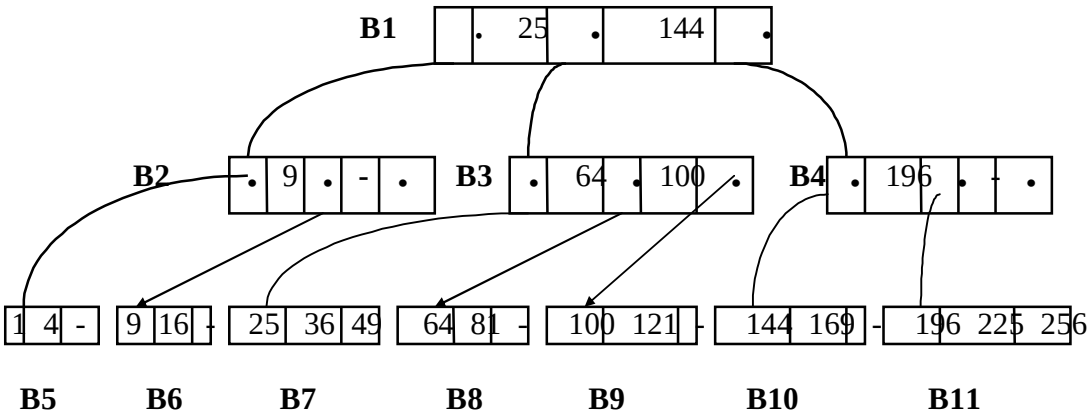
Insertion

To insert a new record into B-tree, first locate the appropriate data block, following B-tree look-up procedure. If there is spare space in the data block, we insert a new record in sequence, moving subsequent records if necessary. If there is no room for the insertion, acquire a new block. Some of the records from the old block are moved to the new block and the old as well as new block contain almost equal number of records. Index records must be updated accordingly.

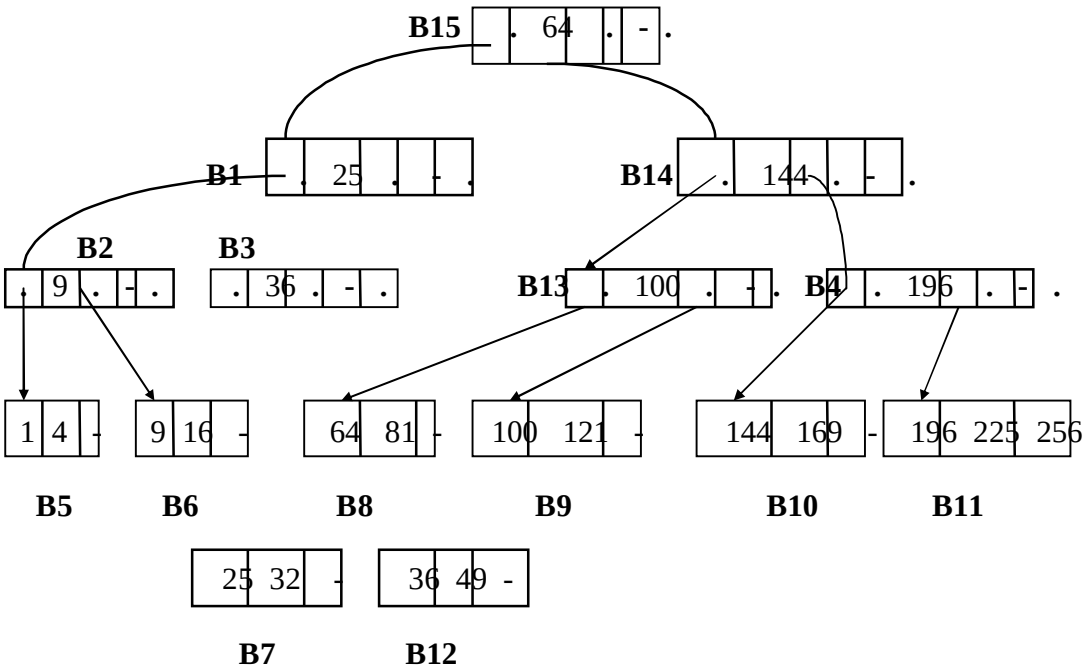
Deletion

By look-up find the record to be deleted. After deleting it shift the blocks' subsequent records down to close the gap. If the block is now less than half full, look for an adjacent child of the same parent that contains more than the minimum number of records. If such a block is found, we shift records out of it into the block from which deletion took place so as to even out the numbers of records in the two blocks as far as possible. The index must be updated accordingly.

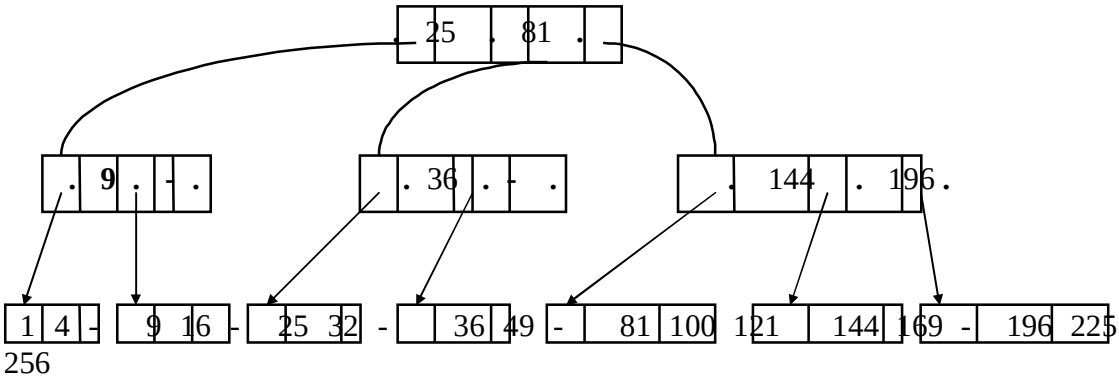
Example



Initial B- tree



B-tree after insertion of 32



B-tree after deletion 64

Time Analysis of B-tree

Suppose we have a file with n records organised into a B-tree with parameters d and e (d for index block, e for data block). Any index block has 2e-1 >= 3 number of records. Any data block has 2d-1 >= 3 number of records.

The tree will have no more than n / e leaves, no more than n / de parents of leaves, n / d^2 e parents of parents of leaves and so on.

If there is i nodes on paths from the root to leaves, then $n \geq d^{i-1}e$, or else there would be fewer than one node at the level of the root, which is impossible.

Hence $\log_d n/e \geq i - 1$

$i - 1 \leq \log_d(n/e)$

$i + 1 \leq 2 + \log_d (n/e)$

From look-up operation i read operations on blocks is sufficient. For insertion, deletion and modification one more block operation is required. Here height of the tree i - 1.

Example
n = 1,000,000, d = 50, e = 5

The expected number of read/write of blocks in an operation is $2 + \log (200000) \leq 6$.
This is greater than that of hashed access which is 3 (approx).