Chapter 5

File Management

LEARNING OBJECTIVES

- *Files*
- *Memory hierarchies*
- *Description of disk devices*
- *File records*
- *Sorted files*
- *Hashing techniques*
- *Extendible hashing*
- *Index update*
- *Clustering index*
- *B-Trees*
- *B+Trees*
- *Over flow in internal node*

Files

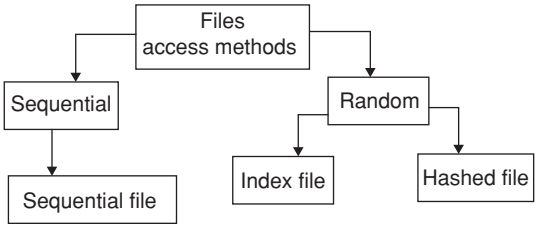
Databases are stored on magnetic disks as files of records. Computer storage media form a storage hierarchy that includes two main categories.

Primary storage This category includes storage media that can be operated on, directly by CPU, such as the computer main memory and cache memory. Primary storage provides fast access but is of limited storage capacity.

Secondary storage This category includes magnetic disks, optical disks, and tapes. These devices usually have a larger capacity, less cost, and slower access to data. Data in secondary storage cannot be processed directly by the CPU, it must be copied into primary storage.

File Structure

Taxonomy of file structure



Sequential file A sequential file is one in which records can only be accessed sequentially, one after another from beginning to end. Records are stored contiguously on the storage device.

Index files These files are used to access a record in the file. The entire index file is loaded into main memory data and indexes are stored in the same file. The term 'index file' is used as a synonym for the term 'database file'. The index file contains parameters that specify the name and location of file used to store DB.

Indexing Indexing mechanism is used to speed up access to desired data. An index file consists of records (called *index entries*) of the form.

Search-key	Pointer

Index files are typically much smaller than the original file.

Ordered indices In ordered index, index entries are stored, sorted on the search-key value.
Example: Author catalogue in library.

Memory Hierarchies

At the primary storage level, the memory hierarchy includes cache memory which is a static RAM.

The next level of primary storage is DRAM (dynamic RAM) which provides the main work area for the CPU for keeping programs and data and is called the *main memory*.

At the secondary storage level, the hierarchy includes magnetic disks, as well as mass storage in the form of CD-ROM (compact disk read-only memory) and tapes. Programs reside in DRAM and large permanent databases reside on secondary storage.

Another form of memory, *flash memory*, is non-volatile. Flash memories are high-density, high-performance memories using EEPROM (electrically erasable programmable read-only

memory) technology. The advantage of flash memory is the fast access speed, the disadvantage is that an entire block must be erased and written over at a time. Finally, magnetic tapes are used for archiving and backup storage of data.

Description of Disk Devices

Magnetic disks are used for storing large amounts of data. The capacity of a disk is the number of bytes it can store. A disk is single sided if it stores information on only one of its surfaces and double sided if both surfaces are used. To increase storage capacity, disks are assembled into a disk pack, which may include many disks and hence many surfaces. Information is stored on a disk surface in concentric circles with small width, each having a distinct diameter. Each circle is called a *track*. For disk packs, the tracks with the same diameter on the various surfaces are called a *cylinder* because of the shape they would form if connected in space.

A track usually contains a large amount of information; it is divided into smaller blocks (or) sectors. The division of track into equal-sized disk blocks (or pages) is set by the operating system during disk formatting.

Blocks are separated by fixed-size inter-block gaps, which include specially coded control information written during disk initialization. This information is used to determine which block on the track follows each inter block gap. Transfer of data between main memory and disk takes place in units of disk blocks. The hardware address of a block is the combination of a cylinder number, track number and block number is supplied to the disk I/O hardware.

The actual hardware mechanism that reads or writes a block is the disk read/write head, which is part of a system called a *disk drive*. A disk is mounted in the disk drive, which includes a motor that rotates the disk. To transfer a disk block, given its address, the disk controller must first mechanically position the read/write head on the correct track. The time required to do this is called the *seek time*. There is another delay called *rotational delay* or *latency*; the beginning of the desired block rotates into position under the read/write head. It depends on the RPM of the disk. Finally, some additional time is needed to transfer the data, which is called *block-transfer time*. Hence, the total time needed to locate and transfer an arbitrary block, given its address is the sum of the seek time, rotational delay and block transfer time. The seek time and rotational delay are usually much larger than the block transfer time.

File Records

Data is usually stored in the form of *records*. Each record consists of a collection of related data values or items where each value is of one or more bytes and corresponds to a particular field of the record. Records describe entities and their attributes.

Record type A collection of field names and their corresponding data types constitutes a record type (or) record format.

A file is a sequence of records. If every record in the file has exactly the same size (in bytes), the file is said to be made up of fixed-length records. If different records in the file have different sizes, the file is said to be made up of variable-length records.

Spanned Versus Unspanned Records

The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block.

Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit

$$bfr = \lfloor B/R \rfloor \text{ records per block}$$

The value *bfr* is called the *blocking factor* for the file. Some times R may not divide B exactly, so we have some unused space in each block equal to $B - (bfr * R)$ bytes. To utilize this unused space, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called *spanned*, because records can span more than one block. Whenever a record is larger than a block, we must use a spanned organization. If records are not allowed to cross block boundaries, the organization is called *unspanned*. This is used with fixed-length records having $B > R$, because it makes each record start at a known location in the block. For variable-length records, either a spanned or an unspanned organization can be used.

For variable-length records using spanned organization, each block may store a different number of records. In this case, the blocking factor *bfr* represents the average number of records per block for the file. We can use *bfr* to calculate the number of blocks '*b*' needed for a file of '*r*' records.

$b = \lceil (r/bfr) \rceil$ blocks

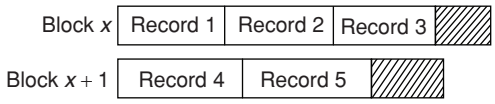


Figure 1 Unspanned records

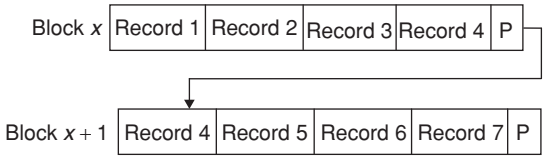


Figure 2 Spanned

There are several standard techniques for allocating the blocks of a file on disk. In contiguous allocation, the file blocks are allocated to consecutive disk blocks. In linked

allocation, each file block contains a pointer to the next file block. A combination of the two allocates clusters of consecutive disk blocks, and the clusters are linked. Clusters are sometimes called file *segments* (or) *extents*. Another possibility is to use indexed allocation, where one or more index blocks contain pointers to the actual file blocks.

Sorted Files (Ordered Records)

We can physically order the records of a file on disk based on the values of the one of their fields called the *ordering field*. This leads to an ordered or sequential file. If the ordering field is also a key field of the file, a field guaranteed to have a unique value in each record, then the field is called the *ordering key for the file*.

Advantages

1. Reading the records in order of the ordering key values becomes extremely efficient, because no sorting is required.
2. Finding the next record from the current one in order of the ordering key usually requires no additional block access, because the next record is in the same block as the current one.
3. Using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used. This constitutes an improvement over linear searches, although it is not often used for disk files.

A binary search for disk files can be done on the blocks rather than on the records. Suppose that a file has 'b' blocks numbered 1, 2 ,…, b, the records are ordered by ascending value of their ordering key field and we are searching for a record whose ordering key field value is K. Assuming that disk addresses of the file blocks are available in the file header, the binary search usually accesses $\log_2^{(b)}$ blocks, whether the record is found (or) not, an improvement over linear searches, where, on the average, (b/2) blocks are accessed when the record is found and 'b' blocks are accessed when the record is not found.

Type of Organization	Access Method	Average Time to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Ordered scan	$b/2$
Ordered	Binary search	$\log_2 b$

Ordered files are rarely used in database applications unless an additional access path, called a *primary index*, is used; this results in an indexed sequential file. This further improves the random access time on the ordering key field.

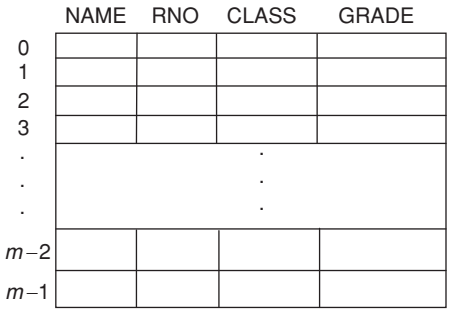
Hashing Techniques

The other type of primary file organization is based on hashing, which provides very fast access to records on certain search conditions. This organization is usually called a *hash file*.

The search condition must be an equality condition on a single field, called the *hash field* of the file. If the hash is also a key field of the file, in which case it is called the *hash key*.

The idea behind hashing is to provide a function 'h', called a hash function or randomizing function, which is applied to the hash field value of a record and yields the address of the disk block in which the record is stored. We need only a single-block access to retrieve that record.

Example:



Internal Hashing

For internal files, hashing is implemented as a hash table through the use of an array of records. Suppose that the array index range is from 0 to $M - 1$, then we have M slots whose addresses corresponds to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and $M - 1$. One common hash function is the $h(K) = K \bmod M$ function, which returns the remainder of an integer hash field value K after division by M; this value is then used for the record address.

Non-integer hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation.

A collision occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called *collision resolution*. There are different methods for collision resolution as follows:

Open addressing Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.

Chaining For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is thus maintained.

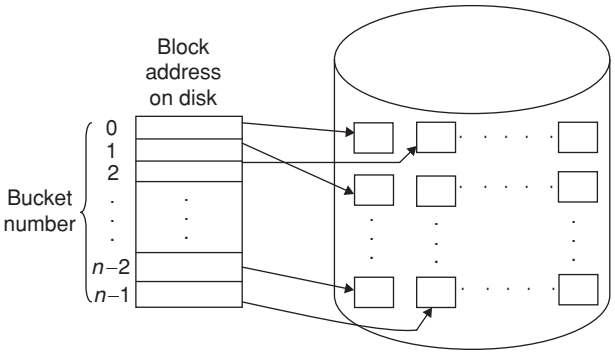
Multiple hashing The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

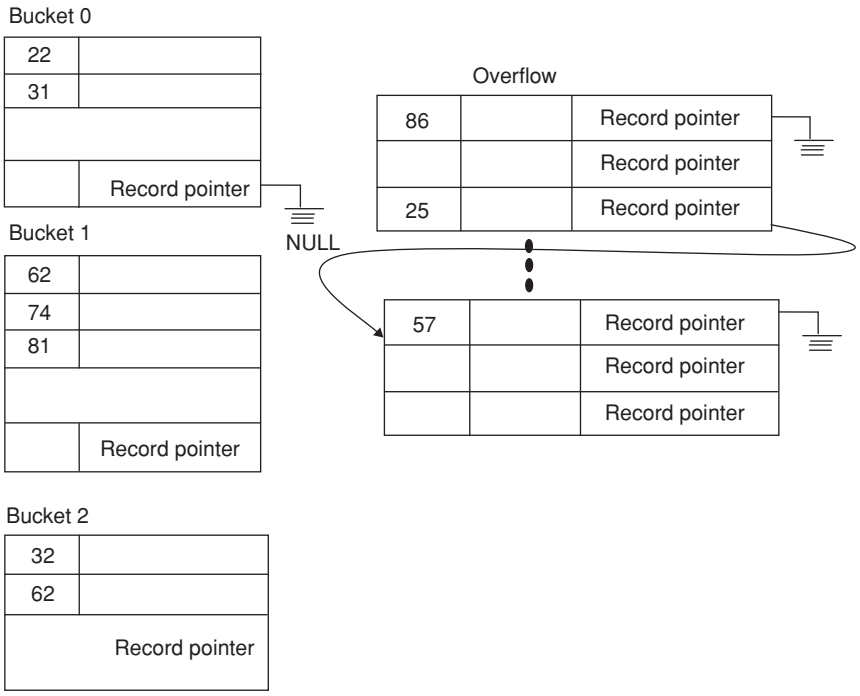
If we expect to have '*r*' records to store in the table, we should choose *M* locations for the address space such that (*r*/*M*) is between 0.7 and 0.9. It may also be useful to choose a prime number for *M*, since it has been demonstrated that this distributes the hash addresses better over the address space when the 'mod' hashing function is used. Other hash functions may require *M* to be a power of 2.

External Hashing

Hashing for disk files is called *external hashing*. To suit the characteristics of disk storage, the target address space is made of buckets, each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous blocks. The hashing function maps a key into a relative bucket number, rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address.

The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. If the capacity of bucket exceeds, we can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket. The pointers in the linked list should be record pointers, which include both a block address and a relative record position within the block.





The hash function is $h(k) = k$ mod 10 and the hashing scheme described above is called *static hashing* because a fixed number of buckets *M* is allocated. It can be a drawback for dynamic files. Suppose that we allocate *M* buckets for the address space and let '*m*' be the maximum number of records that can fit in one bucket, then at most (*m* * *M*) records will fit in the allocated space. If the number of records turns out to be substantially fewer than (*m* * *M*), we are left with a lot of unused space.

If the number of records increases to substantially more than (*m* * *M*), numerous collisions will result and retrieval will be slowed down because of the long lists of overflow

records. In either case, we may have to change the number of blocks M allocated and then use a new hashing function (based on the new value of M) to redistribute the records. These organizations can be quite time consuming for large files. Newer dynamic file organizations based on hashing allows the number of buckets to vary dynamically with only localized reorganization.

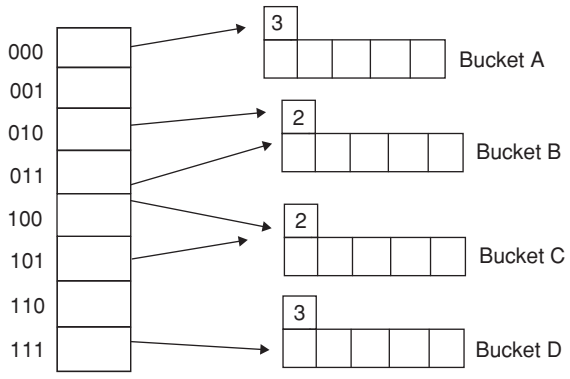
Hashing Techniques with Dynamic File Expansion

The disadvantage of static hashing is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The first scheme is extendible hashing, It stores an access structure in addition to the file hence it is similar to indexing. The main difference is that the access structure is based on the values that result after application of the hash function to the search field. In indexing, the access structure based on the values of the search field itself. The second technique, called *linear hashing*, does not require additional access structure.

These hashing schemes take advantage of the fact that the result of applying a hashing function is a non-negative integer and hence can be represented as a binary number. The access structure is built on the binary representation of the hashing function result, which is a string of bits. We call this the *hash value of a record*. Records are distributed among buckets based on the values of the leading bits in their hash values.

Extendible Hashing

In extendible hashing, a type of directory, an array of 2^d bucket addresses is maintained, where d is called the *global depth of the directory*. The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry, and the address in that determines the bucket in which the corresponding records are stored. Several directory locations with the same first d' bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket. A local depth d' is stored with each bucket specifies the number of bits on which the bucket contents are based.



The value of d can be increased and decreased by one at a time, thus doubling or halving the number of entries in the directory array. Doubling is needed if a bucket, whose local depth d' is equal to the global depth d, overflows. Halving occurs if $d > d'$ for all the buckets after some locations occur. Most record retrievals require two block accesses: one to the directory and the other to the bucket.

The main advantage of extendible hashing is the performance of the file does not degrade as the file grows, as opposed to static external hashing where collisions increases and the corresponding chaining causes additional accesses. No space is allowed in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed. The space overhead for the directory table is negligible.

Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time a reorganization is more expensive is when the directory has to be doubled (or) halved.

A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing.

Indexing

Indexes are auxiliary access structures, which are used to speed up the retrieval of records in response to certain search conditions. The index structure typically provides secondary access paths, which provide alternative ways of accessing the records without affecting the physical placement of records on disk. They enable efficient access to records based on the indexing fields that are used to construct the index.

Any field of the file can be used to create an index and multiple indexes on different fields can be constructed on the same file. To find a record or records in the file based on a certain selection criterion on an indexing field, one has to initially access the index, which points to one or more blocks in the file where the required records are located. The most prevalent types of indexes are based on ordered files (single-level indexes) and tree data structures (multilevel indexes, B^+ trees).

Dense Index files: Index record appears for every search-key value in the file.

Brighton		*A*-217	Brighton	750
Downtown		*A*-101	Downtown	600
Mianus		*A*-110	Downtown	300
Perryridge		*A*-215	Mianus	400
		*A*102	Perryridge	800

Figure 3 Dense index file.

Sparse index files These files contain index records for only some search-key values. Applicable when records are sequentially ordered on search key.

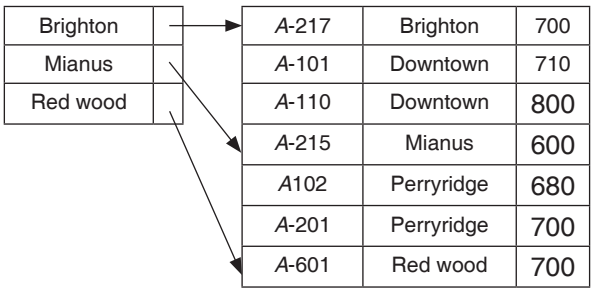


Figure 4 Sparse index file.

Compared to dense index, sparse index takes less space and less maintenance over head for insertions and deletions. It is slower than dense index for locating records.

Index Update

Record deletion If delete key was the only record in the file with its particular search-key value, the search key is deleted from the index also.

In dense index, delete the search key.

In spare index, if deleted key value exists in the index, the value is replaced by next search-key value in the file. If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Record insertion In dense index, if the search-key value doesn't appear in the index insert it.

If index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. If a new block is created, the first search-key value appearing in the new block is inserted into the index.

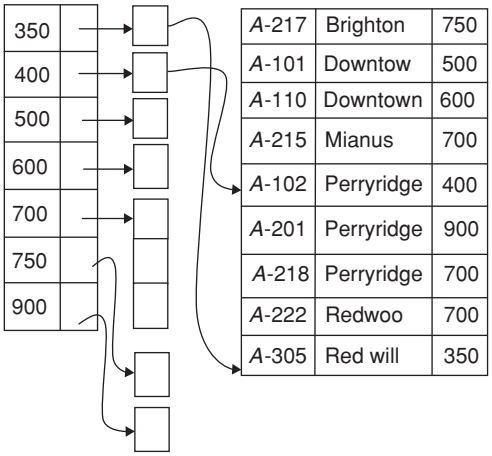


Figure 5 Secondary index.

Secondary index example:

1. Index record points to a bucket that contains pointers to all the actual records with that particular search – key value
2. secondary index have to be dense

Single-level Ordered Indexes

A file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file is called an *indexing field* or *indexing attribute*. The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are ordered so that we can do a binary search on the index.

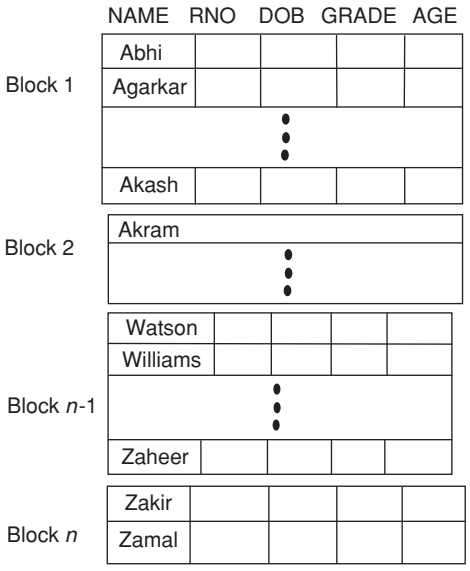
The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient. Multilevel indexing does away the need for a binary search at the expense of creating indexes to the index itself.

Types of Ordered Indexes

1. Primary index
2. Clustering index
3. Secondary index

Primary index A primary index is an ordered file whose records are of fixed length with two fields. The first field is of the same data type as the ordering key field called the *primary key of the data file*, and the second field is a pointer to a disk block (block address). There is one index entry (index record) in the index file for each block in the data file. Each index entry has the value of the primary key field for the record in a block and a pointer to that block as its two field values. The two field values of index entry i is $<k(i), p(i)>$.

Example:



To create a primary index on the ordered file shown in the above figure, we use the NAME field as primary key, because that the ordering key field on the file (assuming that each value of NAME is unique). Each entry in the index has a NAME value and a pointer. Some sample index entries are as follows:

< $k(1)$ = (Abhi), $p(1)$ = address of block 1 >
< $k(2)$ = (Akram), $p(2)$ = address of block 2 >
< $k(3)$ = (Brat), $p(3)$ = address of block 3 >

The below figure illustrates this primary index. The total number of entries in the index is the same as the number of disk blocks in the ordered data file. The first record in each block of the data file is called the *anchor record of the block (or) block anchor*.

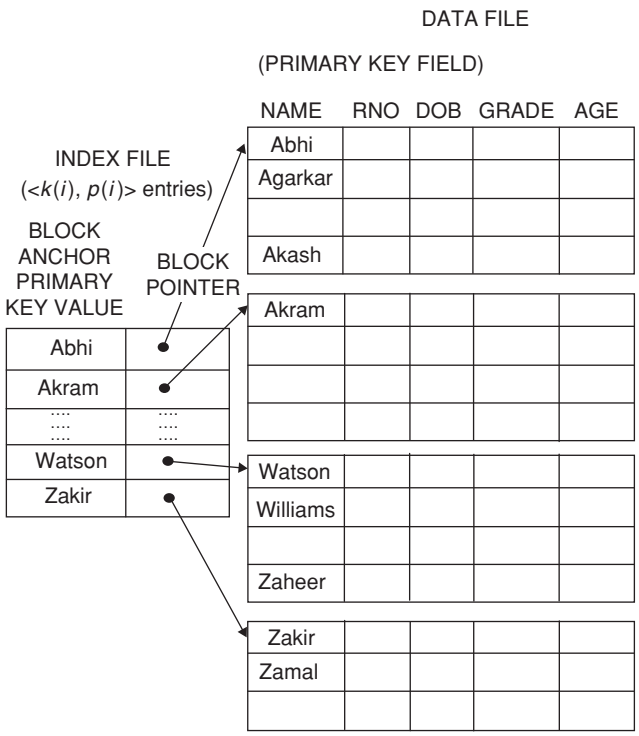


Figure 6 Primary index on the ordering key field of the file.

Indexes can also be characterized as dense or sparse. A dense index has an index entry for every search-key value (every record) in the data file. A sparse (non-dense) index has index entries only for some of the search values. A primary index is non-dense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

The index file for primary index needs fewer blocks than does the data file, for two reasons as follows:

1. There are fewer index entries than there are records in the data file.
2. Each index entry is typically smaller in size than a data record because it has only two fields. So more index entries than data records can fit in one block.

A binary search on the index file requires fewer block accesses than a binary search on the data file. The binary search for an ordered data file required $\log_2{}^b$ block accesses. But if the primary index file contains b_i blocks, then to locate a record with a search-key value requires a binary search of that index and access to the block containing that record, a total of $\log_2{}^b{}_i + 1$ accesses.

A record whose primary key value is k lies in the block whose address is $p(i)$, where $k(i) \le k \le k(i + 1)$. The ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. To retrieve a record, given the value k of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is $p(i)$.

The following example illustrates the saving in block accesses that is attainable when a primary index is used to search for a record.

Example: Suppose that we have an ordered file with r = 24,000 records stored on a disk with block size B = 512 bytes. File records are of fixed size and are unspanned, with record length R = 120 bytes.

The blocking factor for the file would be $bfr = \lfloor B/R \rfloor$

$$= \left\lfloor \frac{512}{120} \right\rfloor = \lfloor 4.26 \rfloor = 4 \text{ records per block}$$

The number of blocks needed for the file is

$$b = \left\lceil \left(\frac{r}{bfr} \right) \right\rceil = \left\lceil \frac{24,000}{42} \right\rceil = 6000 \text{ blocks}$$

A binary search on the data file would need

$\lceil \log_2{}^b = \log_2{}^{6000} \rceil = 13$ block accesses

Example: For the above data, suppose that the ordering key field of the file is V = 7 bytes long, a block pointer, P = 5 bytes long, and we have constructed a primary index for the file.

The size of each index entry is $R_i = (7 + 5) = 12$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor$

$\lfloor 512/12 \rfloor = \lfloor 42.66 \rfloor = 42$ entries per block.

The total number of index entries r_i is equal to number of blocks in the data file, which is 6000. The number of index blocks is hence

$b_i = \lceil (r_i/bfr_i) \rceil = \lceil 6000/42 \rceil = 142$ blocks

To perform a binary search on the index file would need $\lceil \log_2 b_i \rceil = \lceil \log_2 142 \rceil = 8$ block accesses. To search for a record using the index, we need one additional block access to the data file for a total of '9' block accesses.

Disadvantage: A major problem with a primary index is insertion and deletion of records. If we attempt to insert a record in its correct position in the data file, we have to not only move records to make space for the new record but also change some index entries.

Clustering Index If records of a file are physically ordered on a non-key field, which does not have a distinct value for each record, that field is called the *clustering field*. We can create a different type of index called *clustering index* to

speed up the retrieval of records that have the same value for the clustering field.

A clustering index is also an ordered file with two fields, the first field is of the same type as the clustering field of the data file, and the second field is a block pointer.

There is one entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the first block in the data file that has a record with that value for its clustering field.

The record insertion and deletion still cause problems, because the data records are physically ordered. To alleviate the problem of insertion, reserve a whole block (or a cluster of contiguous blocks) for each value of the clustering field, all records with that value are placed in the block (or block cluster). A clustering index is an example of a non-dense index, because it has an entry for every distinct value of the indexing field which is a non-key.

Secondary Index A secondary index provides a secondary means of accessing a file for which some primary access already exists. The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values. The index is an ordered file with two fields. The second field is either a block pointer or a record pointer. There can be many secondary indexes for the same file.

First consider a secondary index access structure on a key field that has a distinct value for every record such a field is some times called a *secondary key*.

The records of the data file are not physically ordered by values of the secondary key field, we cannot use block anchors. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index.

B⁻ Trees

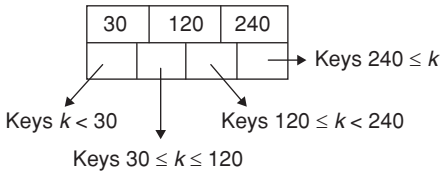
1. A commonly used index structure
2. Non-sequential, 'balanced'
3. Adapts well to insertions and deletions
4. Consists of blocks holding at most n keys and $n + 1$ pointers.
5. We consider a variation actually called a B^+ *tree*

B⁺ Trees

B^+ trees are a variant of B^- trees. In B^+ trees data stored only in leaves, leaves form a sorted linked list.

Parameter – n

Branching factor – $n + 1$

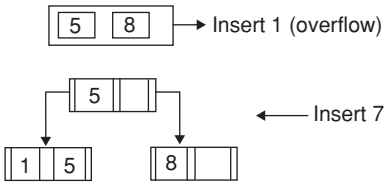


Each node (except root) has at least $n/2$ keys. B^- tree stands for balanced tree. All the paths through a B^- tree from root to different leaf nodes are of the same length (balanced path length). All leaf nodes are at the same depth level.

This ensures that number of disk accesses required for all the searches are same. The lesser the depth (level) of an index tree, the faster the search.

Insertion into B^+ tree

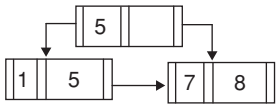
Given nodes 8 5 1 7 3 12 Initially start with root node (has no children)



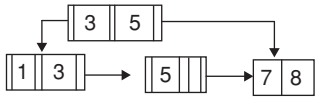
Overflow in Leaf Node

Split the leaf node First, j = ceiling $((p_{leaf} + 1)/2)$ entries are kept in the original node and the remaining moved to the new leaf.

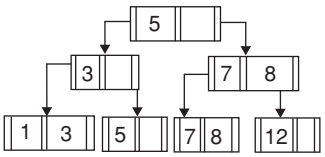
1. Create a new internal node, and jth index value is replicated in the parent internal node.
2. A pointer is added to the newly formed leaf node.



Insert 3 → overflow



Insert 12 (overflow, split propagates, new level)



Overflow in Internal Node

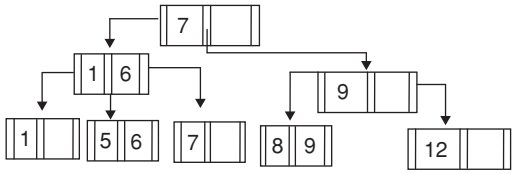
Split the internal node, the entries up to P_j where j = floor $((p + 1)/2)$ are kept in the original node and remaining moved to the new internal node

1. Create a new internal node and the jth index value is moved to the parent internal node (without replication)
2. Pointers are added to the newly formed nodes.

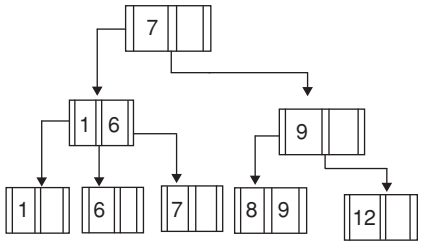
3. B^+ tree ensures some space always left in nodes for new entries. Also makes sure all nodes are at least half full.

Deletion in B^+ Trees

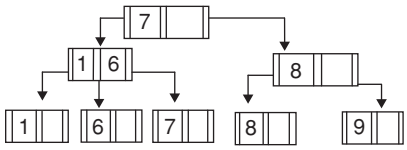
Delete 5,12,9 from the below B^+ tree:



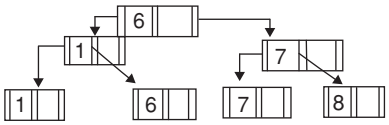
Delete 5:



Delete 12:
Under flow has occurred, so redistribute.



Delete 9: Underflow (merge with left) redistribute.



Advantages

1. B^- *Trees and* B^+ *trees*: B^- tree is a data structure used for external memory.
2. B^- trees are better than binary search trees if data is stored in external memory.
3. Each node in a tree should correspond to a block of data.
4. Each node can store many data items and has many successors.
5. The B^- tree has fewer levels but search for an item takes more comparisons at each level.
6. If a B^- tree has order '*d*', then each node (except root) has at least d/2 children, then the depth of the tree is at most $\log_{d/2}$ (size) + 1.
7. In the worst case, we need $(d - 1)$ comparisons in each node (using linear search)
8. Fewer disk accesses are required compared to binary Tree.
9. The usual data structure for an index is the B^+ tree.
10. Every modern DBMS contains some variant of B^- trees in addition with other index structures depending on the application.
11. B^- trees and B^+ trees are one and the same. They differ from B^- trees in having all data in the leaf blocks.
12. Compared to binary trees, B^- trees will have higher branching factor.
13. Binary trees can degenerate to a linear list, B^- trees are balanced, so this is not possible.
14. In B^+ tree, the values in inner nodes are repeated in the leaf nodes.
15. The height of the tree might decrease, because the data pointer is needed only in the leaf nodes, we can also get a sorted sequence.
16. In B^- trees, all leaves have the same distance from root hence B^- trees are balanced. This ensures that the chain of links followed to access a leaf node is never too long.
17. The time complexity of search operation in B^- tree (tree height) is $O(\log n)$, where 'n' is the number of entries.
18. Advantage of B^+ tree automatically reorganizes itself with small and local changes while doing insertions and deletions, reorganization of entire file is not required to maintain performance.
19. Disadvantage of B^+ tree, extra Insertion and deletion overhead, space overhead.
20. B^+ trees can be used as dynamic multilevel Indexes.

Exercises

Practice Problems I

Directions for questions 1 to 20: Select the correct alternative from the given choices.

1. Consider the following specifications of a disk. Block size of a disk is 512 bytes, inter-block gap size is 128 bytes Number of blocks per track is 20 and number of tracks per surface is 400.

(i) What is the capacity of disk including Inter block gap?

(A) 124000 (B) 1260000
(C) 5120000 (D) 512000

(ii) What is the capacity of disk excluding Inter block gap?

(A) 25400 (B) 25600
(C) 25800 (D) 25900

2. Consider the following specifications of a disk. Block size of disk is 512 bytes, 2000 tracks per surface, 50 sectors per track and 5 double sided platters.

(i) What is the capacity of track in bytes?
(A) 4096000 (B) 4086000
(C) 4076000 (D) 4066000

(ii) What is the capacity of surface in bytes?
(A) 25600000 (B) 512000
(C) 5120000 (D) 51200000

(iii) What is the capacity of disk in bytes?
(A) 512*10^4 (B) 512*10^5
(C) 512*10^6 (D) 512*10^7

(iv) How many cylinders does it have?
(A) 512 (B) 1000
(C) 2000 (D) 2048

(v) Identify the In valid disk block size from below:
(A) 2048 (B) 51200
(C) 4098 (D) 4096

3. What is the order of internal node of B^+ tree suppose that a child pointer takes 6 bytes, the search field value takes 14 bytes and the block size is 512 bytes?
(A) 23 (B) 24
(C) 25 (D) 26

4. The order of a leaf node in a B^+ tree is the maximum number of (value, data, record pointer) pairs it can hold. Given that block size is 1 k bytes (1024 bytes), data record pointer is 7 bytes long, the value field is '9' bytes long and block pointer is 6 bytes.
(A) 63 (B) 64
(C) 65 (D) 66

5. The following key values are inserted into a B^+ tree in which order of the internal nodes is 3, and that of the leaf nodes is 2, in the sequence given below. The order of internal nodes is the maximum number of tree pointers in each node, and the order of leaf nodes is the maximum number of data items that can be stored in it. The B^+ tree is initially empty. 10, 3, 6, 8, 4, 2, 1. What is the maximum number of times leaf nodes would get split up as a result of these insertions?
(A) 3 (B) 4
(C) 5 (D) 6

6. For the same key values given in the above question, suppose the key values are inserted into a B^- tree in which order of the internal nodes is 3 and that of leaf nodes is 2. The order of internal nodes is the maximum number of tree pointers in each node and the order of leaf nodes is the maximum number of data items that can be stored in it. The B^- tree is initially empty. What is the maximum number of times leaf nodes would get split up as a result of these insertions?
(A) 1 (B) 2
(C) 3 (D) 4

7. Suppose that we have an ordered file with 45,000 records stored on a disk with block size 2048 bytes. File records are of fixed size and are unspanned with record length 120 bytes.

(i) What is the blocking factor?
(A) 16 (B) 17
(C) 18 (D) 19

(ii) What is the number of blocks needed for the file?
(A) 2642 (B) 2644
(C) 2646 (D) 2648

(iii) How many block accesses are required to search for a particular data file using binary search?
(A) 10 (B) 11
(C) 12 (D) 13

8. Suppose that the ordering key field of the file is 12 bytes long, a block pointer is 8 bytes long, and we have constructed a primary index for the file. Consider the file specifications given in the above questions.

(i) What is the size of each index entry?
(A) 16 (B) 18
(C) 20 (D) 22

(ii) What is the blocking factor for the index?
(A) 101 (B) 102
(C) 103 (D) 104

(iii) What is the total number of index entries?
(A) 2642 (B) 2644
(C) 2646 (D) 2648

(iv) What is the number of index blocks?
(A) 22 (B) 24
(C) 26 (D) 28

(v) How many block accesses are required, if binary search is used?
(A) 3 (B) 4
(C) 5 (D) 6

9. For the file specifications given in Q. No. 7, if we construct secondary index on a non-ordering key field of the file that is 12 bytes long, a block-pointer of size 8 bytes, each index entry is 20 bytes long and the blocking factor is 102 entries per block.

(i) What is the total number of index blocks?
(A) 422 (B) 424
(C) 442 (D) 444

(ii) How many block accesses are required to access the secondary index using binary search?
(A) 6 (B) 7
(C) 8 (D) 9

10. For the file specifications given in Q. No. 8, if we construct a multilevel index, number of 1st-level blocks are 442, blocking factor is 102, each index entry is 20 bytes long.

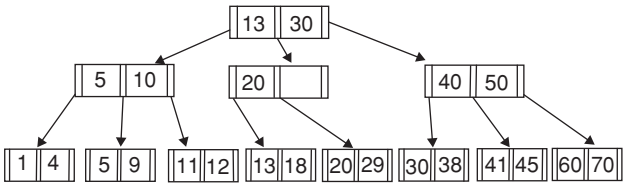
(i) What is the number of 2nd-level blocks?
(A) 4 (B) 5
(C) 6 (D) 7

(ii) What is the number of 3rd-level blocks?
(A) 0 (B) 1
(C) 2 (D) 3

11. Construct a B^+ tree for (1,4,7,10,17,21,31) with $n = 4$, which nodes will appear two times in a final B^+ tree?
(A) 17,7,20 (B) 17,7,20,25
(C) 17,20,25 (D) 7,17.25

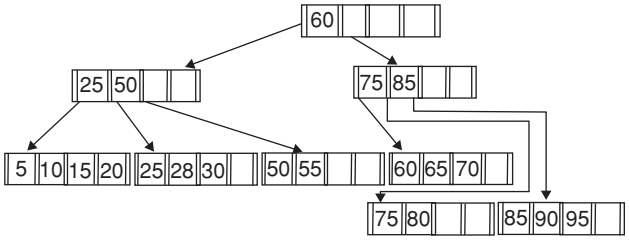
12. Suppose the hash function is $h(x) = x \bmod 8$ and each bucket can hold at most two records. The extendable hash structure after inserting 1, 4, 5 ,7, 8, 2, 20, what is the local depth of '4'?
(A) 0 (B) 1
(C) 2 (D) 3

13. Consider the given B^+ tree, insert 19 into the tree, what would be the new element in level 2?



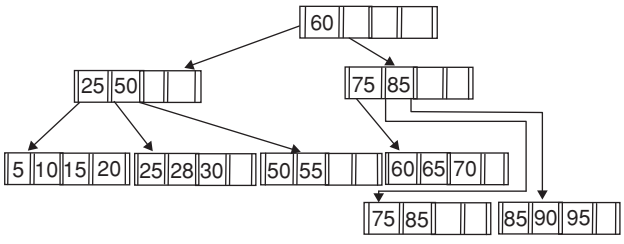
(A) 13 (B) 18
(C) 20 (D) 29

14. Consider the given B^+ tree, delete 70 and 25 from the tree, what are the elements present in level 2? ($\therefore$ root is at level 1)



(A) 25, 50, 75 (B) 25, 50,75, 85
(C) 28, 50, 75, 85 (D) 28, 50, 65, 75

15. Delete 60 from the above given tree (Q. No. 14). After deletion, what is the total number of nodes present in the tree?



(A) 5 (B) 6
(C) 7 (D) 8

16. What will be the number of index records/block?
(A) 68 (B) 65
(C) 69 (D) None

17. What will be the number of index blocks?
(A) 442 (B) 440
(C) 400 (D) None

18. Consider the following:
Block size = 1025 bytes
Record length in data file = 100 bytes
Total number of records = 30000
Search key = 9 bytes
Pointer = 6 bytes
What is the number of index blocks?
(A) 44 (B) 45
(C) 46 (D) None

19. Which of the following is maximum search time t_{max} in B^- trees?

(A) $t_{max} = a \log_2 \left(\frac{N}{2}\right)\left[\frac{a+d}{\log_2 m} + \frac{bm}{\log_2 m} + c\right]$

(B) $t_{max} = a \log_2 \left(\frac{N}{2}\right)\left[\frac{a+d}{\log_2 m} + \frac{bm}{\log_2 m} + c\right]$

(C) $t_{max} = a \log_2 N\left[\frac{a+d}{\log_2 m} + \frac{bm}{\log_2 m} + c\right]$

(D) $t_{max} = a \log_2 (N)\left[\frac{a}{\log_2 m} + \frac{bm}{\log_2 m} + c\right]$

20. Consider a B^+ tree. A child pointer takes 3 bytes, the search field value takes 7 bytes, and the block size is 256 bytes. What is the order of the internal node?
(A) 63 (B) 64
(C) 65 (D) 66

Practice Problems 2

Directions for questions 1 to 20: Select the correct alternative from the given choices.

1. Which of the following is true?
(A) Every conflict serializable is view serializable
(B) Every view serializable is conflict serializable
(C) Both A and B
(D) A schedule can be either only conflict serializable or only view-serializable.

2. Which one is the 2-phase locking rule?
(A) Two transactions cannot have conflicting locks
(B) No unlock operation can precede a lock operation in the same transaction.
(C) No data is/are affected until all locks are obtained and until the transaction is in its locked point.
(D) All of the above

3. If Transaction T_i has obtained an exclusive mode lock on item Q, then
(A) T_i can read Q
(B) T_i can write Q
(C) T_i can read and write
(D) Neither read nor write

4. Phantom phenomenon is
(A) A transaction retrieves a collection of objects but sees same result.
(B) A transaction retrieves a collection of objects but sees different results.
(C) Transaction T_1 waits for T_2 and T_2 waits for T_1
(D) This problem arises when the transaction has not locked all the objects.

5. We can avoid the starvation of transactions by granting locks by following manner:

When a transaction T_i requests a lock on a data item Q in a particular mode M, the concurrency control manager grants the lock provided that
(A) There is no other transaction holding a lock on Q in a mode that conflicts with M.
(B) There is no other transaction that is waiting for a lock on Q,
(C) (A) and (B)
(D) None

6. Which one is correct?
(A) Upgrading can take place only in shrinking phase
(B) Upgrading can take place only in growing phase.
(C) Downgrading can take place only in growing phase
(D) (A) and (C) both

7. A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:
(A) When a transaction T_i issues a read (Q) operation, the system issues a lock $s(Q)$ instruction followed by the read instruction.
(B) When T_i issues a write Q operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade Q instruction followed by the write Q instruction, otherwise the system issues a lock -$X(Q)$ instruction, followed by the write Q instruction.
(C) All locks obtained by a transaction are unlocked after that transaction commits or aborts.
(D) All of the above

8. Which one is correct?
(A) A lock manager can be implemented as a process that receives messages from transactions and sends messages in reply.
(B) It uses linked list of records.
(C) It uses hash table called lock table.
(D) All of the above

Common data questions 9 and 10: Transaction T_1 has 5 instructions. Transaction T_2 has 3 instructions.

9. The number of non-serial transactions will be
(A) 15 (B) 8
(C) 2 (D) 56

10. The number of serial transaction schedules will be
(A) 15 (B) 8
(C) 2 (D) 56

11. In a heap file system, which of the following function finds 'average number of blocks to be read'?

(A) $\frac{i}{n} = \frac{1}{2}(1+n) = \frac{n}{2}$

(B) $\sum_{i=1}^{n} \frac{i}{n} = \frac{1}{2}(1+n) = \frac{n}{2}$

(C) $\sum_{i=0}^{n-1} \frac{i}{n} = \frac{1}{2}(1+n) = \frac{n}{2}$

(D) All of the above

12. What is the disadvantage in one directory per user?
(A) Different applications can be divided into separate groups.
(B) Different applications cannot be divided into separate groups
(C) All files are in a single group
(D) All of the above

13. What are the possible violations if an application program uses isolation-level 'Read uncommitted'?
(A) Dirty read problem
(B) Non-repeatable read problem
(C) Phantom phenomenon
(D) All of the above

14. The two-phase locking protocol
(A) ensures serializability
(B) issues locks in two phases
(C) unlocks in two phases
(D) All of the above

15. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the
(A) block point (B) critical section
(C) growing point (D) lock point

16. Which of the following is not a problem of file management system?
(A) Data redundancy
(B) Lack of data independence
(C) Program dependence
(D) All of the above

17. Which of the following is/are true about master list of an index file?
(i) Is sorted in ascending order
(ii) A number is assigned to each record.
(A) Only (i) (B) Only (ii)
(C) Both (i) and (ii) (D) None of the above

18. To have a file, holding a list is necessary to
(i) Identify the records in the list
(ii) Identify the name, and type of the fields of each record.
(iii) Decide which fields will be used as sort of index keys.
(A) Only (i) and (ii)
(B) Only (i) and (iii)
(C) Only (ii) and (iii)
(D) All of the above

19. Two files may be joined into a third file, if the following is true:
(A) if they have row in common
(B) if they have a field in common
(C) Both (A) and (B)
(D) None

20. The minimum number of record movements required to merge four files *w*(with 10 records), *x*(with 20 records), *y*(with 15 records) and *z*(with 5 records) is:
(A) 50 (B) 40
(C) 30 (D) 35

Previous Years' Questions

1. A clustering index is defined on the fields which are of type **[2008]**
(A) non-key and ordering
(B) non-key and non-ordering
(C) key and ordering
(D) key and non-ordering

2. A B^- tree of order 4 is built from scratch by 10 successive insertions. What is the maximum number of node splitting operations that may take place? **[2008]**
(A) 3 (B) 4
(C) 5 (D) 6

3. Consider a file of 16384 records. Each record is 32 bytes long and its key field is of size 6 bytes. The file is ordered on a non-key field, and the file organization is unspanned. The file is stored in a file system with block size 1024 bytes, and the size of a block pointer is 10 bytes. If the secondary index is built on the key field of the file, and a multilevel index scheme is used to store the secondary index, the number of first-level and second-level blocks in the multilevel index are respectively **[2008]**
(A) 8 and 0 (B) 128 and 6
(C) 256 and 4 (D) 512 and 5

4. The following key values are inserted into a B^+ tree in which order of the internal node s is 3, and that of the leaf nodes is 2, in the sequence given below. The order of internal nodes is the maximum number of tree pointers in each node, and the order of leaf nodes is the maximum number of data items that can be stored in it. The B^+ tree is initially empty.

10, 3, 6, 8, 4, 2, 1

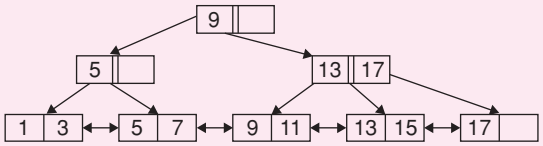
The maximum number of times leaf nodes would get split up as a result of these insertions is **[2009]**
(A) 2 (B) 3
(C) 4 (D) 5

5. Consider a B^+ tree in which the maximum number of keys in a node is 5. What is the minimum number of keys in any non-root node? **[2010]**
(A) 1 (B) 2
(C) 3 (D) 4

6. An index is clustered, if **[2013]**
(A) it is on a set of fields that form a candidate key.
(B) it is on a set of fields that include the primary key.
(C) the data records of the file are organized in the same order as the data entries of the index.
(D) the data records of the file are organized not in the same order as the data entries of the index.

7. A file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index. Then that index is called **[2015]**
(A) Dense (B) Sparse
(C) Clustered (D) Unclustered

8. With reference to the B+ tree index of order 1 shown below, the minimum number of nodes (including the Root node) that must be fetched in order to satisfy the following query: "Get all records with a search key greater than or equal to 7 and less than 15" is _____ **[2015]**



9. Consider a B^+ tree in which the search key is 12 bytes long, block size is 1024 bytes, record pointer is 10 bytes long and block pointer is 8 bytes long. The maximum number of keys that can be accommodated in each non-leaf node of the tree is _____. **[2015]**

10. B+ Trees are considered **BALANCED** because **[2016]**

(A) The lengths of the paths from the root to all leaf nodes are all equal.
(B) The lengths of the paths from the root to all leaf nodes differ from each other by at most 1.
(C) The number of children of any two non - leaf sibling nodes differ by at most 1.
(D) The number of records in any two leaf nodes differ by at most 1.

11. In a B+ tree, if the search-key value is 8 bytes long, the block size is 512 bytes and the block pointer size is 2 bytes, then the maximum order of the B+ tree is _____. **[2017]**

Answer Keys

Exercises

Practice Problems 1

1. (i) C (ii) A **2.** (i) B (ii) D (iii) C (iv) C (v) C **3.** C **4.** A **5.** C **6.** B
7. (i) B (ii) D (iii) C **8.** (i) C (ii) B (iii) D (iv) C (v) C **9.** (i) C (ii) D
10. (i) B (ii) B **11.** B **12.** D **13.** B **14.** C **15.** B **16.** A **17.** A **18.** B
19. A **20.** C

Practice Problems 2

1. A **2.** D **3.** C **4.** B **5.** C **6.** B **7.** D **8.** D **9.** D **10.** C
11. B **12.** B **13.** D **14.** D **15.** D **16.** D **17.** B **18.** D **19.** B **20.** B

Previous Years' Questions

1. A **2.** C **3.** C **4.** C **5.** B **6.** C **7.** C **8.** 5 **9.** 50 **10.** A
10. 52