Indexing

- Reason for indexing - For a large file when it contains a large number of records which will eventually acquire large number of blocks, then its access will become slow. In todays events we want a high-speed database.
- Theoretically relational database is derived from set theory, and in a set the order of elements in a set is irrelevant, so does in relations(tables).
- But in practice implementation we have to specify the order.
- A number of properties, like search, insertion and deletion will depend on the order in which elements are stored in the tables.

File organization/ organization of records in a file

1) Ordered file organization: - All the records in the file are ordered on some search key field. Here binary search is possible for e.g. dictionary.

- Because of binary search, searching is efficient
- Maintenance (insertion & deletion) is costly, as it requires re organization of entire file.
- Notes that we will get binary search only if we are using that key for searching on which indexing is done, otherwise it will behave as unsorted file

employee_id	first_name	last_name	hire_date	salary
100	Steven	King	1987-06-17	24000.00
101	Neena	Kochhar	1989-09-21	17000.00
102	Lex	De Haan	1993-01-13	17000.00
103	Alexander	Hunold	1990-01-03	9000.00
104	Bruce	Ernst	1991-05-21	6000.00
105	David	Austin	1997-06-25	4800.00
106	Valli	Pataballa	1998-02-05	4800.00

2) Unordered file organization: - All the records are inserted usually in the end of the file so not ordered according to any field, Because of this only linear search is possible.

- Because of linear search, searching is slow
- Maintenance (insertion & deletion) is easy, as it does not require re organization of entire file.

Q Suppose we have ordered file with records stored r = 30,000 on a disk with Block Size B = 1024 B. File records are of fixed size and are unspanned with record length R = 100 B. Suppose that ordering key field of file is 9 B long and a block pointer is 6 B long, Implement primary indexing?

Important Points about Indexing

- Additional auxiliary access structure is called indexes, a data technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.
- Index typically provides secondary access path, which provide alternative way to access the records without affecting the physical placement of records in the main file.
- Index file is always ordered, irrespective of weather main file is ordered or unordered. So that we can take the advantage of binary search.

Secondary Index

Search key	Address
Azad	0x2300
Priya	0x2100
Ram	0x1F00
Tom	0x2200
Vishal	0X2000

Data File

Address	RegNo	Name	Program	Age
0x1F00	14MS01	Ram	MS CS	21
0x2000	14MS10	Vishal	MS CS	22
0x2100	14MS29	Priya	MS CS	21
0x2200	14MS30	Tom	MS CS	21
0X2300	14MT59	Azad	MTech	23

- Index file always contains two columns one the attribute on which search will be done and other the block or record pointer.
- The size of index file is way smaller than that of the main file.
- As the size of the records is very smaller compare to the main file, as index file record contain only two columns key (attribute in which searching is done) and block pointer (base address of the block of main file which contains the record holding the key), while main file contains all the columns.
- Normally apart from secondary indexing (a type of indexing), the number of records in index file <= the number records in the main file.

- One index file is designed according to an attribute, means more than one index file can be designed for a main file.
- Indexing gives the advantage of faster time, but space taken by index file will be an overhead.
- Number of access required to search the correct block of main file is $\log_2$(number of blocks in index file) + 1
- Index can be created on any field of relation (primary key, non-key)

Q Data which improves the performance and accessibility of the database are called: **(NET-DEC-2015)**

(1) Indexes
(2) User Data
(3) Application Metadata
(4) Data Dictionary

Ans. 1

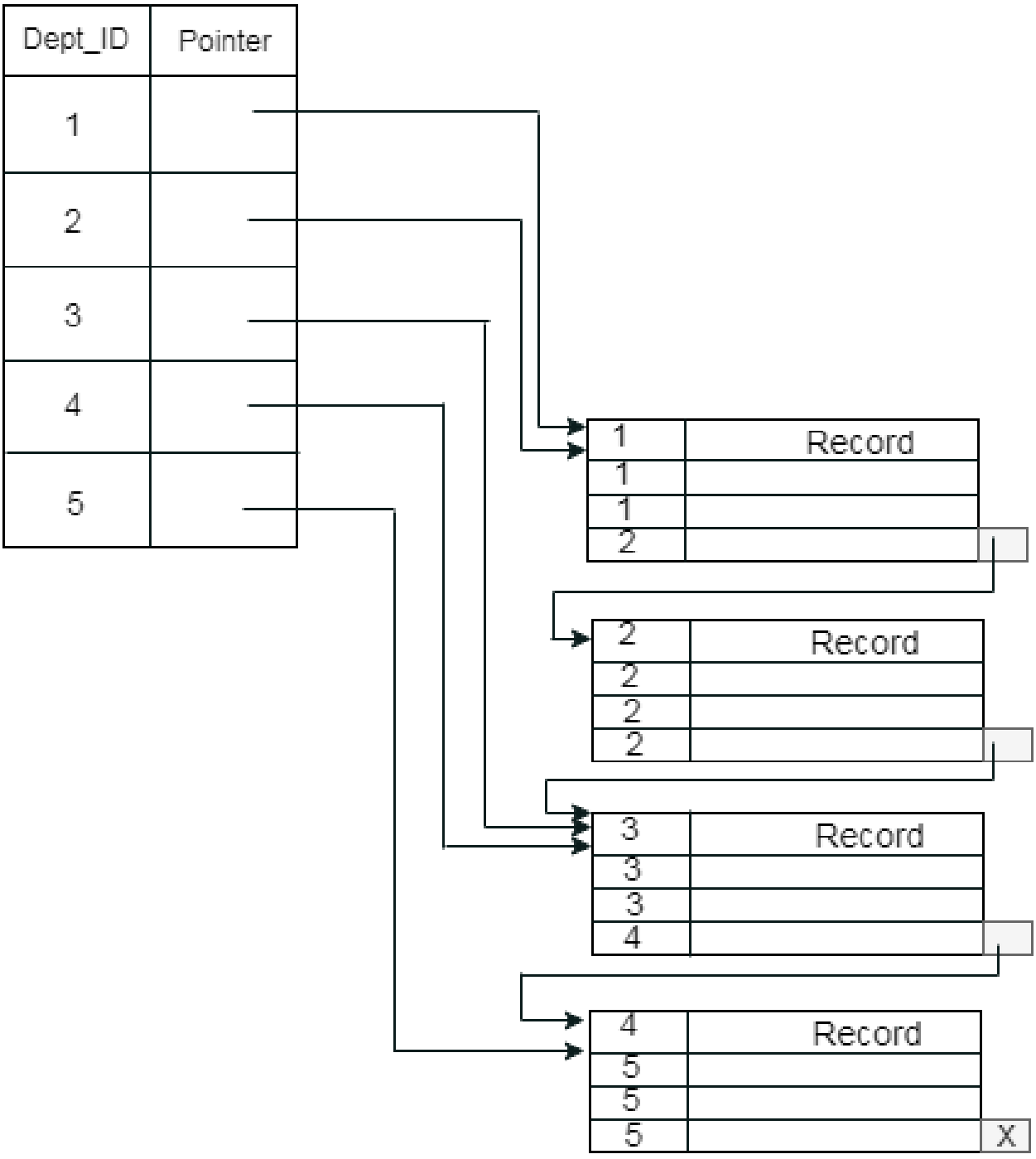
Indexing can be classified on number of criteria's one of them could be –

- Dense Index
- Sparse Index

***DENSE INDEX**: -*

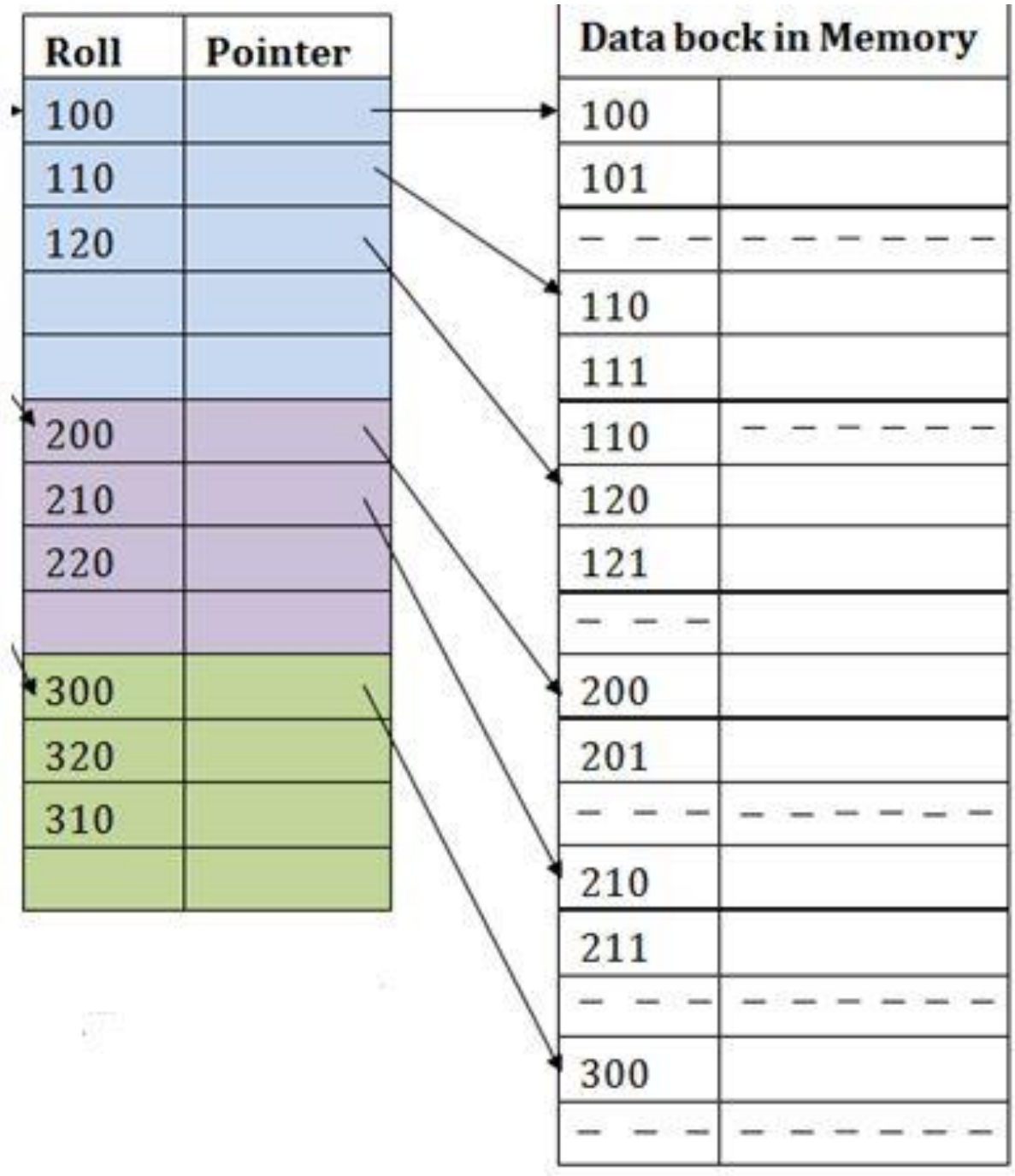
- In dense index, there is an entry in the index file for every search key value in the main file. This makes searching faster but requires more space to store index records itself.

- Note that it is not for every record, it is for every search key value. Sometime number of records in the main file >= number of search keys in the main file, for example if search key is repeated.



SPARSE INDEX:

- If an index entry is created only for some records of the main file, then it is called sparse index.
- No. of index entries in the index file < No. of records in the main file.

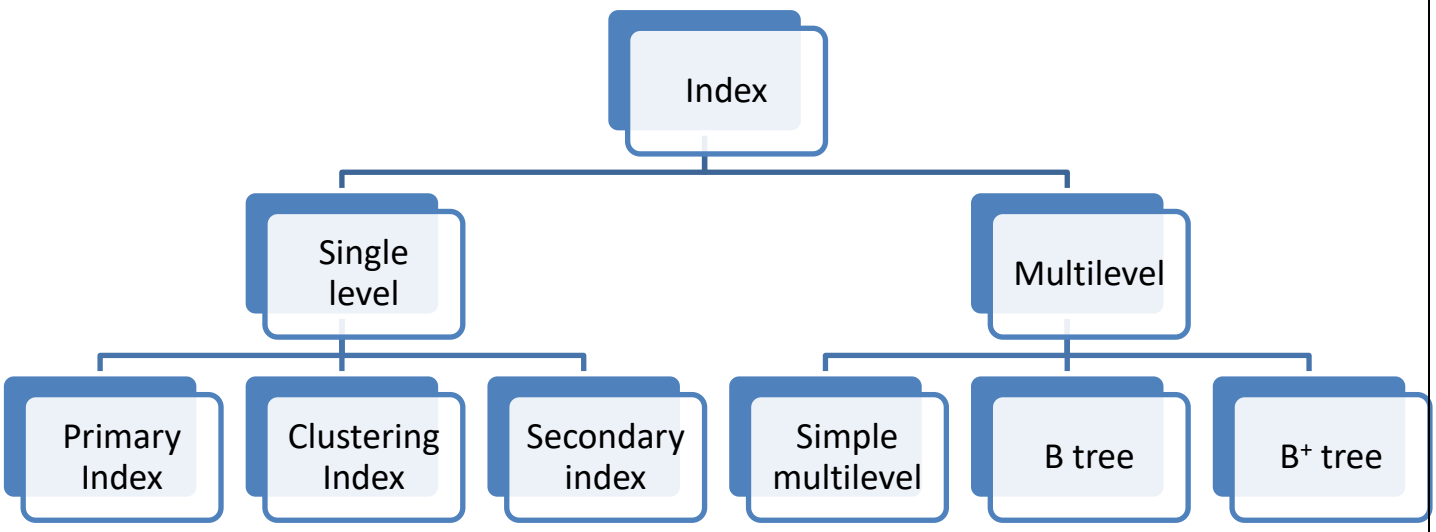


- Note: - dense and sparse are not complementary to each other, sometimes it is possible that a record is both dense and sparse.

Basic term used in Indexing

- BLOCKING FACTOR = No. of Records per block= ⌊block size/record size⌋
- No of blocks required by file = ⌈no of records / blocking factor⌉
- If file is unordered then no of block assesses required to reach correct block which contain the desired record is O(n), where n is the number of blocks.
- if file is unordered then no of block assesses required to reach correct block which contain the desired record is $O(\log_2 n)$, where n is the number of blocks.

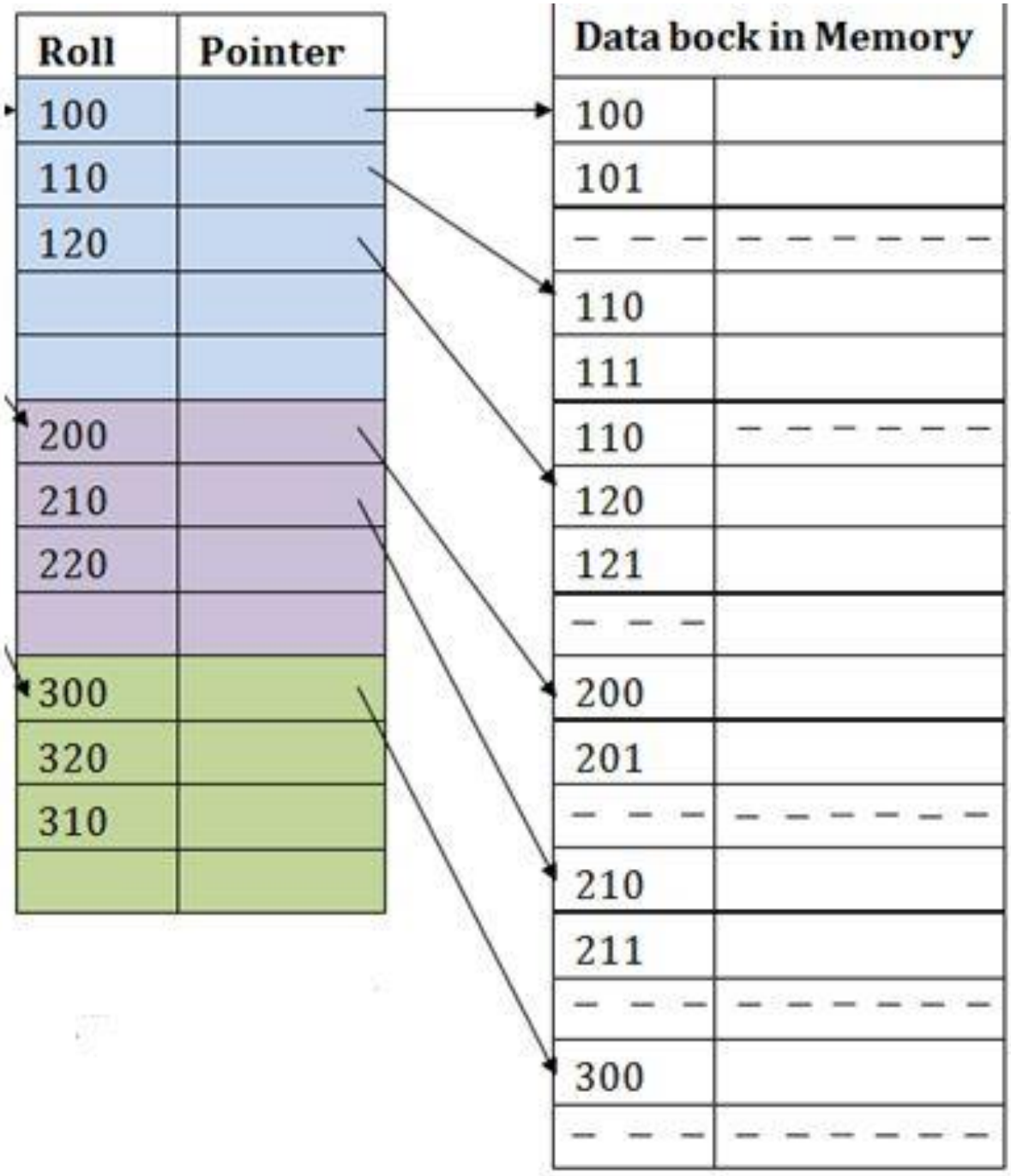
TYPES OF INDEXING



- Single level index means we create index file for the main file, and then stop the process.
- Multiple level index means, we further index the index file and keep repeating the process until we get one block.

PRIMARY INDEXING

- Main file is always sorted according to primary key.
- Indexing is done on Primary Key, therefore called as primary indexing
- Index file have two columns, first primary key and second anchor pointer (base address of block)
- It is an example of Sparse Indexing.
- Here first record (anchor record) of every block gets an entry in the index file
- No. of entries in the index file = No of blocks acquired by the main file.



CLUSTERED INDEXING

- Main file will be ordered on some non-key attributes
- No of entries in the index file = no of unique values of the attribute on which indexing is done.
- It is the example of Sparse as well as dense indexing

Q An index is clustered, if **(GATE-2013) (1 Marks)**
(1) it is on a set of fields that form a candidate key
(2) it is on a set of fields that include the primary key
(3) the data records of the file are organized in the same order as the data entries of the index
(4) the data records of the file are organized not in the same order as the data entries of the index
Ans: 3

Q A clustering index is defined on the fields which are of type **(GATE-2008) (1 Marks)**
1) non-key and ordering
2) non-key and non-ordering
3) key and ordering
4) key and non-ordering
ANSWER A

Q A clustering index is created when_____. **(NET-DEC-2014)**
(A) primary key is declared and ordered
(B) no key ordered
(C) foreign key ordered
(D) there is no key and no order
Ans: c

Q (NET-DEC-2018)
A clustering index is defined on the fields which are of type

Options :

91394342537. **non-key and ordering**

91394342538. **non-key and non-ordering**

91394342539. **key and ordering**

91394342540. **key and non-ordering**

SECONDARY INDEXING

- Most common scenarios, suppose that we already have a primary indexing on primary key, but there is frequent query on some other attributes, so we may decide to have one more index file with some other attribute.
- Main file is ordered according to the attribute on which indexing is done(unordered).
- Secondary indexing can be done on key or non-key attribute.
- No of entries in the index file is same as the number of entries in the index file.
- It is an example of dense indexing.

Q Suppose we have ordered file with records stored r = 30,000 on a disk with Block Size B = 1024 B. File records are of fixed size and are unspanned with record length R = 100 B. Suppose that ordering key field of file is 9 B long and a block pointer is 6 B long, Implement Secondary indexing?

Q 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 unpanned. 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 multi-level index scheme is used to store the secondary index, the number of first-level and second-level blocks in the multi-level index are respectively **(GATE-2008) (1 Marks)**

1) 8 and 0 **2)** 128 and 6 **3)** 256 and 4 **4)** 512 and 5

ANSWER C

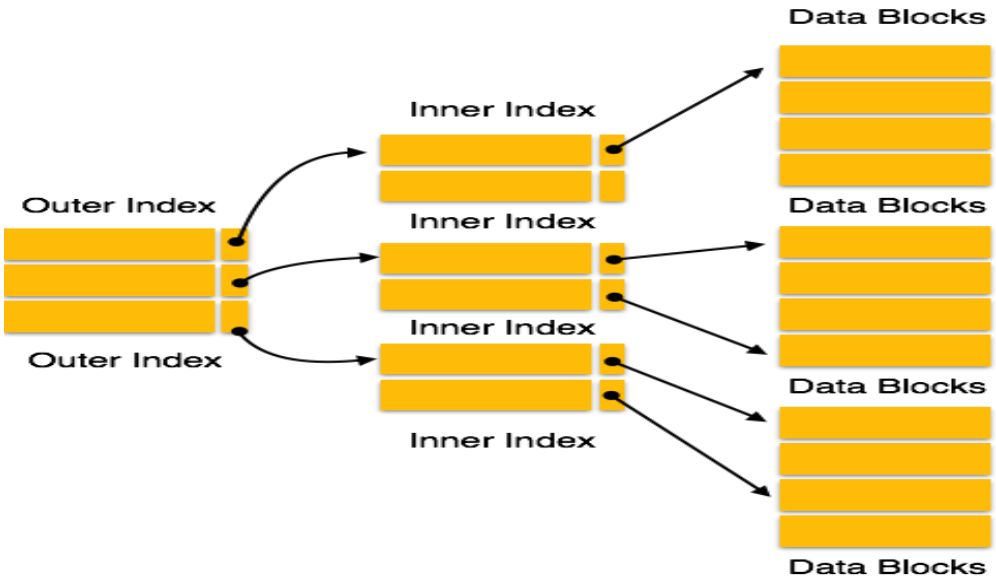
Q A file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index. Then that index is called **(GATE-2015) (1 Marks)**

(A) Dense **(B)** Sparse **(C)** Clustered **(D)** Unclustered

Answer: (C)

MULTILEVEL INDEXING

- Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block-0, which can easily be accommodated anywhere in the main memory.



Reason to have B tree and B+ tree

- After studying indexing in detail now we understand that an index file is always sorted in nature and will be searched frequently, and sometimes index files can be so large that even we want to index the index file (Multilevel index), therefore we must search best data structure to meet our requirements.
- There are number of options in data structure like array, stack, link list, graph, table etc. but we want a data structure which support frequent insertion deletion but at the same time also provide speed search and give us the advantage of having a sorted data.
- If we look at the data structures option then tree seems to be the most appropriate but every kind of tree in the available option have some problems either simple tree or binary search tree or AVL tree, so we end up on designing new data structure called B-tree which are kind of specially designed for sorted stored index files in databases.
- In general, with multilevel indexing, we require dynamic structure, b and b^+ tree is generalized implementation of multilevel indexing, which are dynamic in nature, that is increasing and decreasing number of records. In the first level index file can be easily supported by other level index.
- B tree and B^+ tree also provides efficient search time, as the height of the structure is very less and they are also perfectly balanced.

B tree

- A B-tree of order m if non-empty is an m-way search tree in which.
  - The root has at least two child nodes and at most m child nodes.
  - The internal nodes except the root have at least celling(m/2) child nodes and at most m child nodes.
  - The number of keys in each internal node is one less than the number of child nodes and these keys partition the subtrees of the nodes in a manner similar to that of m-way search tree.
  - All leaf nodes are on the same level.

Root

Rules	**MAX**	**MIN**
CHILD	m	0
DATA	m-1	1

Internal except Root

Rules	***MAX***	***MIN***
CHILD	*m*	[m/2]
DATA	*m-1*	[m/2] - 1

Leaf

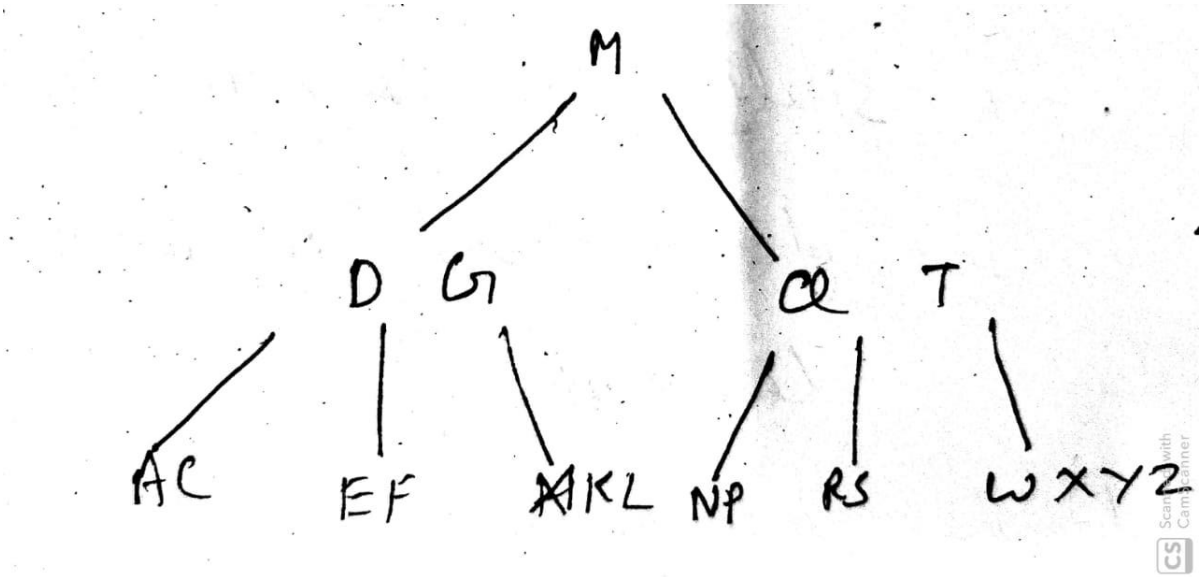
Rules	***MAX***	***MIN***
CHILD	*0*	*0*
DATA	*m-1*	[m/2] - 1

Q Consider the following elements 5, 10, 12, 13, 14, 1, 2, 3, 4 insert them into an empty b-tree of order = 3.

Q Consider the following elements 5, 10, 12, 13, 14, 1, 2, 4, 20, 18, 19, 17, 16, 15, 25, 23, 24, 22, 11, 30, 31, 28, 29 insert them into an empty b-tree of order = 3.

- A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with p – 1 search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a non-roof node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.

Q Consider the Following B-tree of order m=6, delete the following nodes H, T, R, E, A, C, S in sequence?



Deletion in B-TREE-

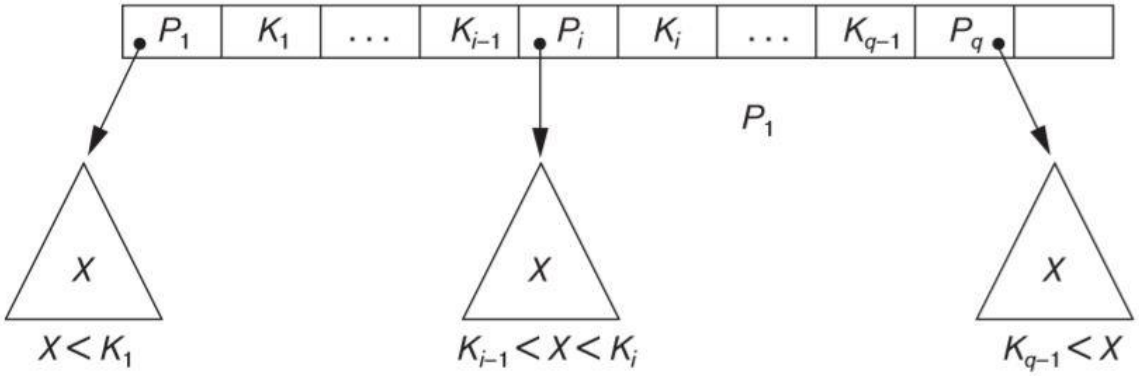
- If the deletion is from the leaf node and leaf node is satisfying the minimal condition even after the deletion, then delete the value directly.
- If deletion from leaf node renders leaf node in minimal condition, then first search the extra key in left sibling and then in the right sibling. Largest value from left sibling or smallest value from right sibling is pushed into the root node and corresponding value can be fetched from parent node to leaf node.
- If the deletion is to be from internal node, then first we check for the extra key in the left and then in the right child. If we find one, we fetch the value in the required node. And delete the key.

Deletion

- If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels. It has been shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69 percent full when the number of values in the tree stabilizes. This is also true of B^+ trees. If this happens, node splitting and combining will occur only rarely, so insertion and deletion become quite efficient. If the number of values grows, the tree will expand without a problem—although splitting of nodes may occur, so some insertions will take more time.

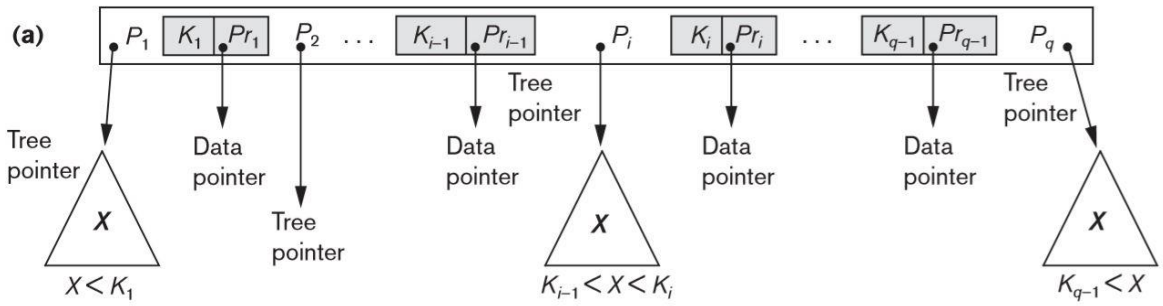
Analysis

- B- TREE- In computer science, a B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
- A **search tree of order** *p* is a tree such that each node contains *at most p* -1 search values and *p* pointers in the order $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$, where $q <= p$. Each P_i is a pointer to a child node (or a NULL pointer), and each K_i is a search value from some ordered set of values. All search values are assumed to be unique. Two constraints must hold at all times on the search tree:
  - Within each node, $K_1 < K_2 < ... < K_{q-1}$.
  - For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.



- We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the **search field** (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value.
- To guarantee that nodes are evenly distributed, so that the depth of the tree is minimized for the given set of keys and that the tree does not get skewed with some nodes being at very deep levels.
- To make the search speed uniform, so that the average time to find any random key is roughly the same
- While minimizing the number of levels in the tree is one goal, another implicit goal is to make sure that the index tree does not need too much restructuring as records are inserted into and deleted from the main file. Thus, we want the nodes to be as full as possible and do not want any nodes to be empty if there are too many deletions. Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

- The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from a node that makes it less than half full. More formally, a **B-tree of order *p***, when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:
- Each internal node in the B-tree is of the form
    - $<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_q>$ where $q <= p$. Each P_i is a **tree pointer**—a pointer to another node in the B- tree. Each Pr_i is a **data pointer**—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).
    - Within each node, $K_1 < K_2 < ... < K_{q-1}$.
    - For all search key field values X in the subtree pointed at by P_i (the ith sub- tree), we have: $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
    - Each node has at most p tree pointers.
    - Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
    - A node with q tree pointers, $q <= p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).
    - All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are NULL.



Conclusion: -

- Very less internal fragmentation, memory utilization is very good.
- Less number of nodes(blocks) are used and height is also optimized, so access will be very fast.
- Difficulty of traversing the key sequentially. Means B-TREE do not hold good for range-based queries of database.

Q A B-Tree used as an index for a large database table has four levels including the root node. If a new key is inserted in this index, then the maximum number of nodes that could be newly created in the process are: **(Gate-2005) (1 Marks)**

(A) 5 **(B)** 4 **(C)** 3 **(D)** 2

Answer: (A)

B^+ Tree

Q Consider the following elements 5, 10, 12, 13, 14, 1, 2, 3, 4 insert them into an empty b^+ tree of order = 3.

Q Consider the following elements 5, 10, 12, 13, 14, 1, 2, 4, 20, 18, 19, 17, 16, 15, 25, 23, 24, 22, 11, 30, 31, 28, 29 insert them into an empty b^+ tree of order = 5

Insertion in B^+ Tree-

- Start from root node and proceed towards leaf using the logic of binary search tree. Value is inserted in the leaf.
- If overflow condition occurs pick the median and push it into the parent node. Also copy the median or key inserted in parent node to the left or right child node.
- Repeat this procedure until tree is maintained.

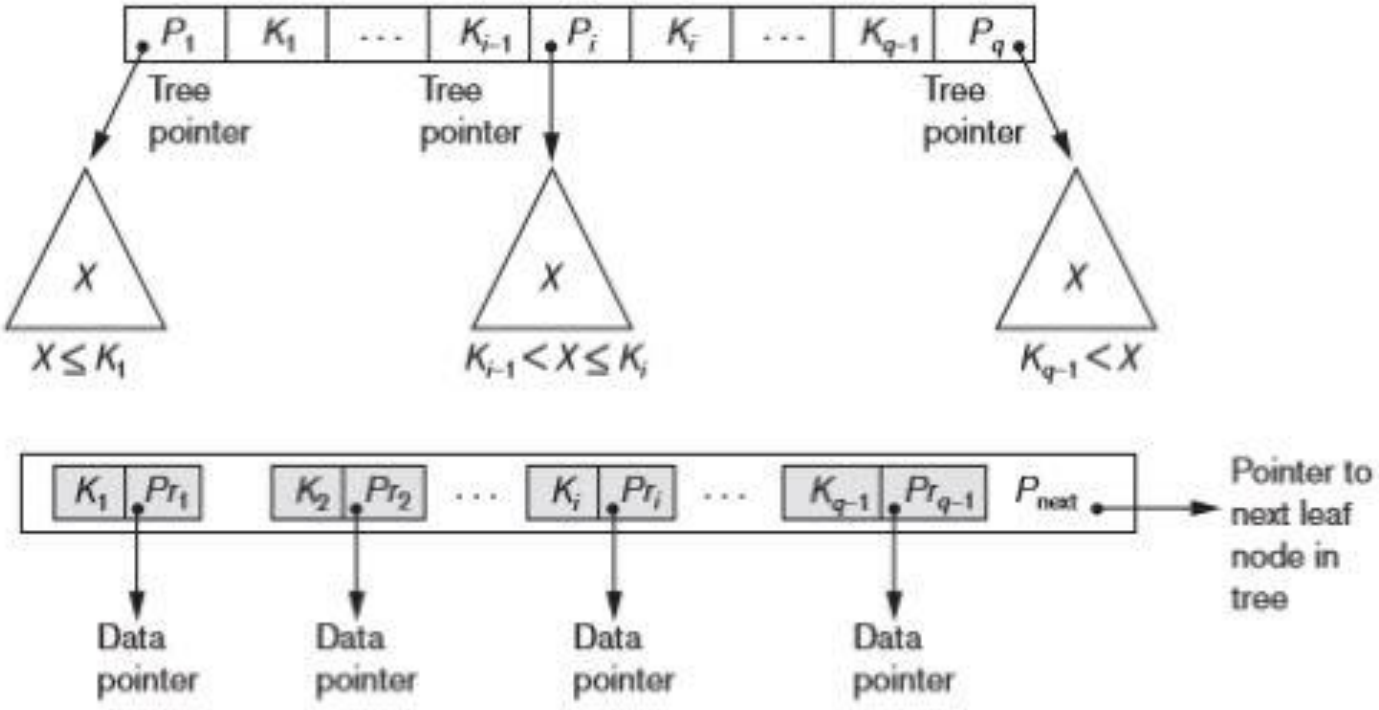
Q Consider the following elements 5, 8, 1, 7, 3, 12, 9, 6 insert them into an empty b^+ tree of order = 3. and then delete following nodes in sequence 9, 8, 12?

Deletion in B^+ Tree-

- if B^+ tree entries are deleted at the leaf nodes, then the target entry is searched and deleted.
- If it is an internal node, delete and replace with the entry from the left position.
- After deletion, underflow is tested,
- If underflow occurs, distribute the entries from the nodes left to it.
- If distribution is not possible from left, then
- Distribute from the nodes right to it.
- If distribution is not possible from left or from right, then Merge the node with left and right to it

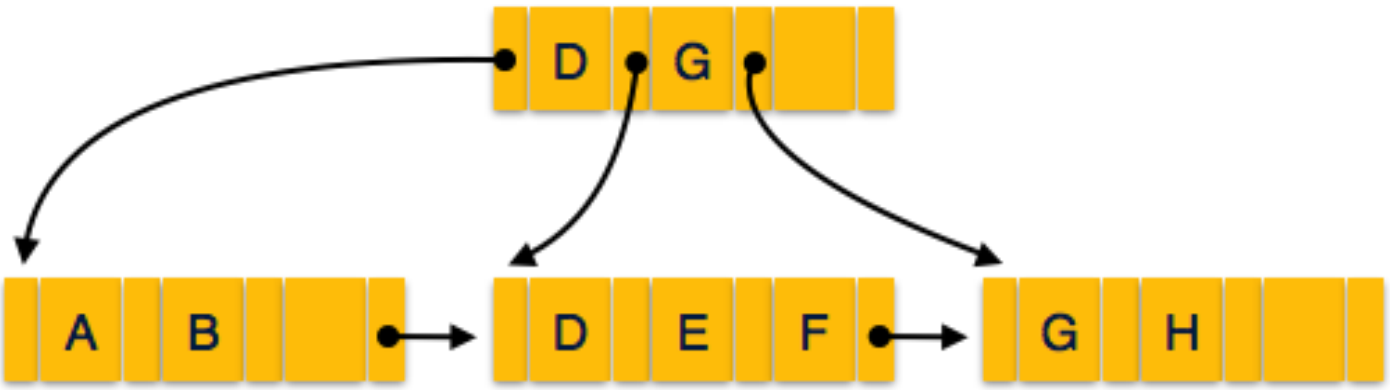
Analysis

- Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B^+-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer.
- In a B^+-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes.
- The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field.
- The leaf nodes of the B^+-tree are usually linked to provide ordered access on the search field to the records.
- Each internal node is of the form $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$
- Within each internal node, $K_1 < K_2 < ... < K_{q-1}$.
- For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X <= K_i$ for $1 < i < q$; $X <= K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
- Each internal node has at most p tree pointers.
- Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
- An internal node with q pointers, $q <= p$, has $q - 1$ search field values.
- The structure of the *leaf nodes* of a B^+-tree of order p is as follows:
- Each leaf node is of the form $<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}>$ where $q <= p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B^+-tree.
- Within each leaf node, $K_1 <= K_2 ... , K_{q-1}, q <= p$.
- Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
- Each leaf node has at least $\lceil (p/2) \rceil$ values.
- All leaf nodes are at the same level.



B^+ is the modified B-tree, where we perform two major modifications in order to support sequential search.

- Every non-leaf data will have a copy(left-right) in the leaf node.
- Every leaf node will have a pointer which will support to the node on its right.



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

The maximum number of times leaf nodes would get split up as a result of these insertions is

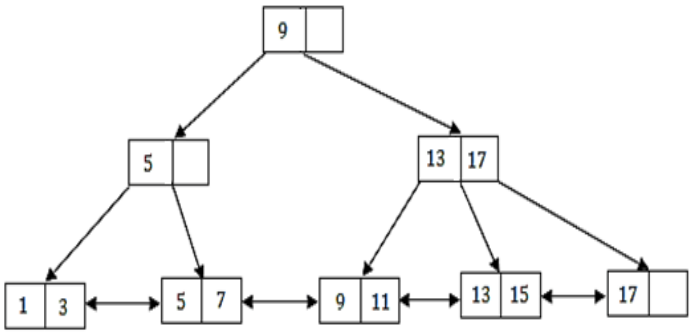
a) 2 **b)** 3 **c)** 4 **d)** 5

(GATE-2009) (2 Marks)

ANSWER d

Q (GATE-2015) (1 Marks)

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 _____.



Ans: 4

Q 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? **(GATE-2010) (1 Marks)**

Answer 2

Q B^+ Trees are considered **BALANCED** because **(GATE-2016) (1 Marks)**

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

Ans: a

Q Which of the following is a key factor for preferring B^+-trees to binary search trees for indexing database relations? **(Gate-2005) (1 Marks)**

a) Database relations have a large number of records

b) Database relations are sorted on the primary key

c) B^+-trees require less memory than binary search trees

d) Data transfer form disks is in blocks

Ans: d

Q B+ trees are preferred to binary trees in databases because **(Gate-2000) (1 Marks)**
(A) Disk capacities are greater than memory capacities
(B) Disk access is much slower than memory access
(C) Disk data transfer rates are much less than memory data transfer rates
(D) Disks are more reliable than memory
Answer: (B)

Q Which of the following is correct? **(Gate-1999) (1 Marks**)

a) B-trees are for storing data on disk and B+ trees are for main memory.

b) Range queries are faster on B+ trees.

c) B-trees are for primary indexes and B++ trees are for secondary indexes.

d) The height of a B+ tree is independent of the number of records.

Ans: b

Q Which one of the following statements is NOT correct about the B^+ tree data structure used for creating an index of a relational database table? **(GATE-2019) (1 Marks)**
(1) Each leaf node has a pointer to the next leaf node
(2) Non-leaf nodes have pointers to data records

(3) B+ Tree is a height-balanced tree

(4) Key values in each node are kept in sorted order

Ans: b

Q In a B+ tree, if the search-key value is 8 bytes long, the block size is 512 bytes and the block pointer is 2 bytes, then the maximum order of the B+ tree is. **(GATE-2017) (2 Marks)**
Ans: 52

Q Consider 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 (Gate-2015) (2 Marks)
Answer: 50

Q 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 the block size is 1K bytes, data record pointer is 7 bytes

long, the value field is 9 bytes long and a block pointer is 6 bytes long, what is the order of the leaf node? **(GATE-2007) (1 Marks)**

a) 63 **b)** 64 **c)** 67 **d)** 68

ANSWER a

Q In a database file structure, the search key field is 9 bytes long, the block size is 512 bytes, a record pointer is 7 bytes and a block pointer is 6 bytes. The largest possible order of a non-leaf node in a B+ tree implementing this file structure is **(Gate-2006) (2 Marks)**

(A) 23 **(B)** 24 **(C)** 34 **(D)** 44

Answer: (C)

Q The order of an internal node in a B+ tree index is the maximum number of children it can have. Suppose that a child pointer takes 6 bytes, the search field value takes 14 bytes, and the block size is 512 bytes. What is the order of the internal node? **(Gate-2004) (2 Marks)**

(A) 24 **(B)** 25 **(C)** 26 **(D)** 27

Answer: (C)

Q A B+ -tree index is to be built on the Name attribute of the relation STUDENT. Assume that all student names are of length 8 bytes, disk block are size 512 bytes, and index pointers are of size 4 bytes. Given this scenario, what would be the best choice of the degree (i.e. the number of pointers per node) of the B+ -tree? **(Gate-2002) (2 Marks)**

(A) 16 **(B)** 42 **(C)** 43 **(D)** 44

Answer: (C)

Q Consider a table T in a relational database with a key field K. A B-tree of order p is used as an access structure on K, where p denotes the maximum number of tree pointers in a B-tree index node. Assume that K is 10 bytes long; disk block size is 512 bytes; each data pointer P_D is 8 bytes long and each block pointer P_B is 5 bytes long. In order for each B-tree node to fit in a single disk block, the maximum value of p is **(Gate-2004) (2 Marks)**

(A) 20 **(B)** 22 **(C)** 23 **(D)** 32

Answer: (C)

It is **23.**

$$(p-1)(\text{key\_ptr\_size} + \text{record\_ptr\_size}) + p.(\text{block\_ptr\_size}) \leq 512$$
$$\implies (p-1)(10+8) + p \times 5 \leq 512$$
$$\implies 23p \leq 530$$
$$\implies p \leq 23.04$$

So, maximum value of p possible will be 23.

Q Consider a join (relation algebra) between relations r(R) and s(S) using the nested loop method. There are 3 buffers each of size equal to disk block size, out of which one buffer is reserved for intermediate results. Assuming size(r(R)) < size(s(S)), the join will have fewer number of disk block accesses if **(GATE-2014) (2 Marks)**

(1) relation r(R) is in the outer loop

(2) relation s(S) is in the outer loop

(3) join selection factor between r(R) and s(S) is more than 0.5

(4) join selection factor between r(R) and s(S) is less than 0.5

Ans: a

Q Match the following: **(NET-DEC-2013)**

List – I	List – II
a. Secondary Index	**i.** Functional Dependency
b. Non-procedural Query Language	**ii.** B-Tree
c. Closure of set of Attributes	**iii.** Relational Algebraic Operation
d. Natural JOIN	**iv.** Domain Calculus

Codes:

	a	b	c	d
a)	i	ii	iv	iii
b)	ii	i	iv	iii
c)	i	iii	iv	ii
d)	ii	iv	i	iii

Ans: D

Q B + tree are preferred to binary tree in database because **(NET-DEC-2011)**

(A) Disk capacities are greater than memory capacities

(B) Disk access much slower than memory access

(C) Disk data transfer rates are much less than memory data transfer rate

(D) Disk are more reliable than memory

Ans: b

Q 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? **(GATE-2008) (1 Marks)**

Answer 5

Q In the indexed scheme of blocks to a file, the maximum possible size of the file depends on: **(NET-JUNE-2015)**

(1) The number of blocks used for index and the size of index

(2) Size of Blocks and size of Address

(3) Size of index

(4) Size of block

Ans. 1