Unit-5

Storage & Indexing

-> A d/b consists of vast quantities of data, as this data must persist, it is stored on external storage devices such as disks, tapes.

-> And this data is fetched in to main memory as and when needed.

-> The data in d/b is mapped in to a no. of files, and a file is organized logically as a sequence of records. These records are mapped on to disk block or disk page.

-> Each file is partitioned in to fixed length storage units called data blocks. Most d/b use data block size of 4 to 8 KB.

-> A data block may contain records of same file or records of other files also based on size of data block.

-> A file may contain records of fixed length or variable length.

-> Files of fixed length records are easy to maintain.

-> Each record in a file has a unique id called record id, with this rid we can identify the disk address of the page containing the record.

→ The Buffer manager component will take the responsibility of transfer of data from main memory to disk & vice versa.

→ The space on disk is managed by disk space manager. It keeps track of pages in use, and free pages.

File organization & Indexing :

→ A file organization is a method of arranging records in a file when the file is stored on disk.

→ There are no of different file organization available such as

1. Sequential file organization
2. Heap "
3. Hash "
4. ISAM "
5. B+Tree "
6. Cluster "

A database consist of a huge amount of data. The data is grouped within a table in RDBMS, and each table have related records. A user can see that the data is stored in form of tables, but in acutal this huge amount of data is stored in physical memory in form of files.

File – A file is named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tables and optical disks.

What is File Organization?

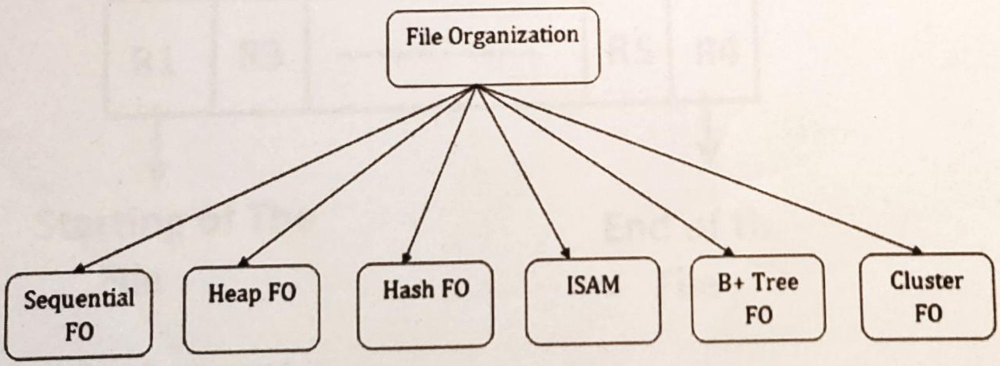
File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record. In simple terms, Storing the files in certain order is called file Organization. **File Structure** refers to the format of the label and data blocks and of any logical control record.

File Organization

- The **File** is a collection of records. Using the primary key, we can access the records. The type and frequency of access can be determined by the type of file organization which was used for a given set of records.
- File organization is a logical relationship among various records. This method defines how file records are mapped onto disk blocks.
- File organization is used to describe the way in which the records are stored in terms of blocks, and the blocks are placed on the storage medium.
- The first approach to map the database to the file is to use the several files and store only one fixed length record in any given file. An alternative approach is to structure our files so that we can contain multiple lengths for records.
- Files of fixed length records are easier to implement than the files of variable length records.

File organization contains various methods. These particular methods have pros and cons on the basis of access or selection. In the file organization, the programmer decides the best-suited file organization method according to his requirement.

Types of file organization are as follows:



- Sequential file organization
- Heap file organization
- Hash file organization
- B+ file organization
- Indexed sequential access method (ISAM)
- Cluster file organization

Types of File Organizations –

Various methods have been introduced to Organize files. These particular methods have advantages and disadvantages on the basis of access or selection . Thus it is all upon the programmer to decide the best suited file Organization method according to his requirements. Some types of File Organizations are :

Sequential File Organization

Heap File Organization

Hash File Organization

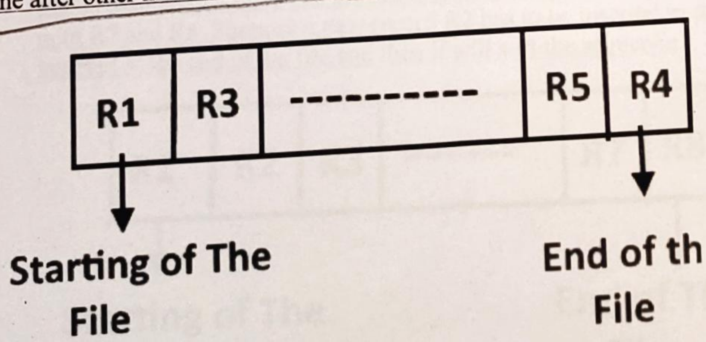
B+ Tree File Organization

Clustered File Organization

Sequential File Organization –

The easiest method for file Organization is Sequential method. In this method the the file are stored one after another in a sequential manner. There are two ways to implement this method:

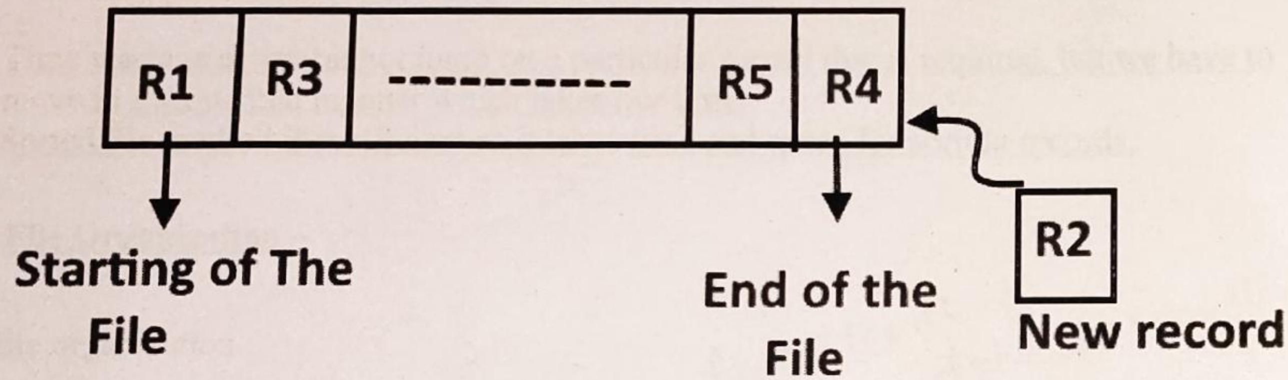
Pile File Method – This method is quite simple, in which we store the records in a sequence i.e one after other in the order in which they are inserted into the tables.



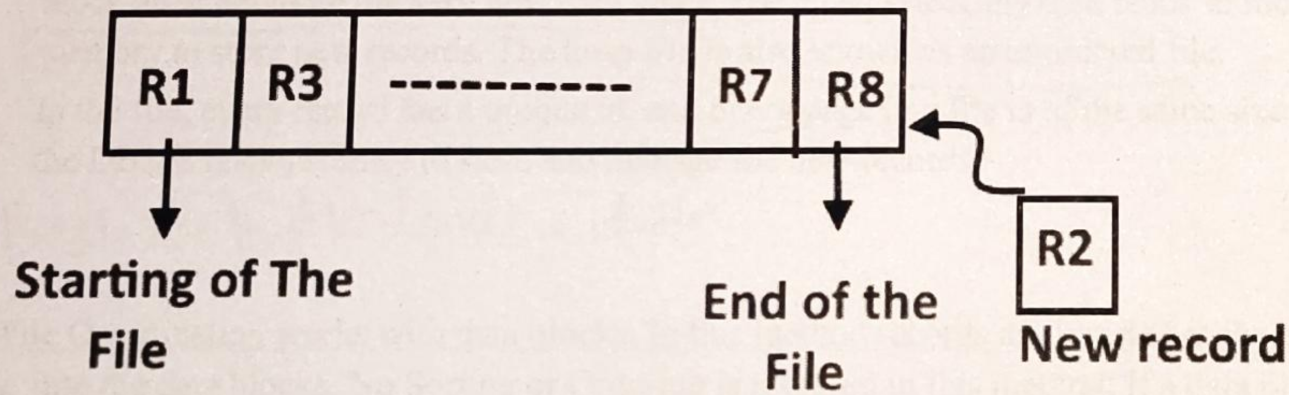
Insertion of new record –

4

Let the R1, R3 and so on upto R5 and R4 be four records in the sequence. Here, records are nothing but a row in any table. Suppose a new record R2 has to be inserted in the sequence, then it is simply placed at the end of the file.



1. **Sorted File Method** –In this method, As the name itself suggest whenever a new record has to be inserted, it is always inserted in a sorted (ascending or descending) manner. Sorting of records may be based on any primary key or any other key.



Insertion of new record –

Let us assume that there is a preexisting sorted sequence of four records R1, R3, and so on upto R7 and R8. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file and then it will sort the sequence .

Pros and Cons of Sequential File Organization –

Pros –

- Fast and efficient method for huge amount of data.
- Simple design.
- Files can be easily stored in magnetic tapes i.e cheaper storage mechanism.

Cons –

- Time wastage as we cannot jump on a particular record that is required, but we have to move in a sequential manner which takes our time.
- Sorted file method is inefficient as it takes time and space for sorting records.

Heap File Organization –

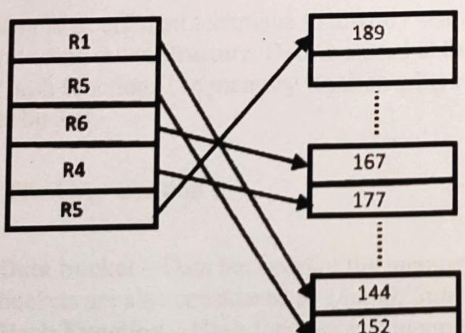
Heap file organization

linear search is done for searching.

- It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.
- When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.
- In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.

Best for bulk loading data

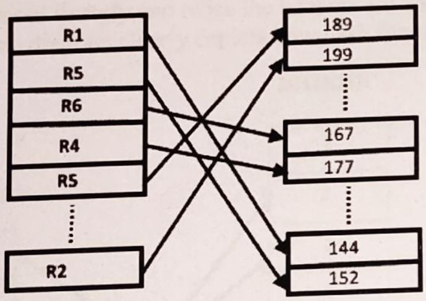
Heap File Organization works with data blocks. In this method records are inserted at the end of the file, into the data blocks. No Sorting or Ordering is required in this method. If a data block is full, the new record is stored in some other block, Here the other data block need not be the very next data block, but it can be any block in the memory. It is the responsibility of DBMS to store and manage the new records.



6

Insertion of new record –

Suppose we have four records in the heap R1, R5, R6, R4 and R3 and suppose a new record R2 has to be inserted in the heap then, since the last data block i.e data block 3 is full it will be inserted in any of the database selected by the DBMS, lets say data block 1.



If we want to search, delete or update data in heap file Organization the we will traverse the data from the beginning of the file till we get the requested record. Thus if the database is very huge, searching, deleting or updating the record will take a lot of time.

Pros and Cons of Heap File Organization –

Pros –

- Fetching and retrieving records is faster than sequential record but only in case of small databases.
- When there is a huge number of data needs to be loaded into the database at a time, then this method of file Organization is best suited.

Cons –

- Problem of unused memory blocks.
- Inefficient for larger databases.

Read next set : (DBMS File Organization-Set 2) | Hashing in DBMS

Hashing is an efficient technique to directly search the location of desired data on the disk without using index structure. Data is stored at the data blocks whose address is generated by using hash function. The memory location where these records are stored is called as data block or data bucket.

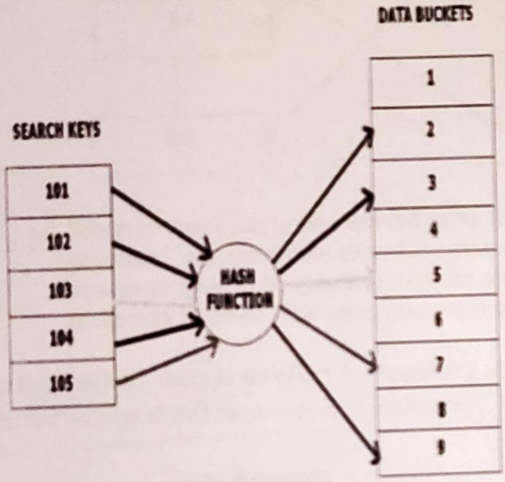
Hash File Organization :

- **Data bucket** – Data buckets are the memory locations where the records are stored. These buckets are also considered as *Unit Of Storage*.
- **Hash Function** – Hash function is a mapping function that maps all the set of search keys to actual record address. Generally, hash function uses primary key to generate the hash

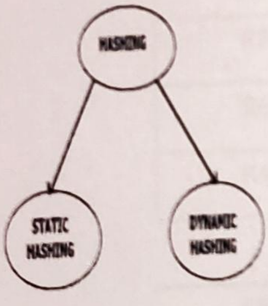
index – address of the data block. Hash function can be simple mathematical function to any complex mathematical function.

- **Hash Index-**The prefix of an entire hash value is taken as a hash index. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed ? then the depth value is increased linearly and twice the buckets are allocated.

Below given diagram clearly depicts how hash function work:



Hashing is further divided into two sub categories :



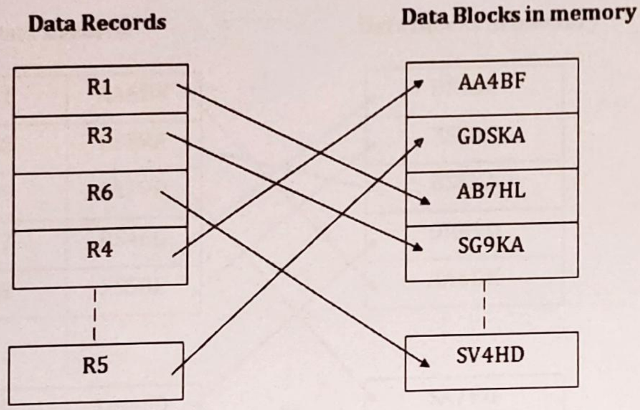
- Hash File Organization
- Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the location of disk block where the records are to be placed.

when tuples are retrieved based on exact match

together - for range of values, part of hash key field, retrival based on other attribute, not hash attribute - this hash File org. not suitable.

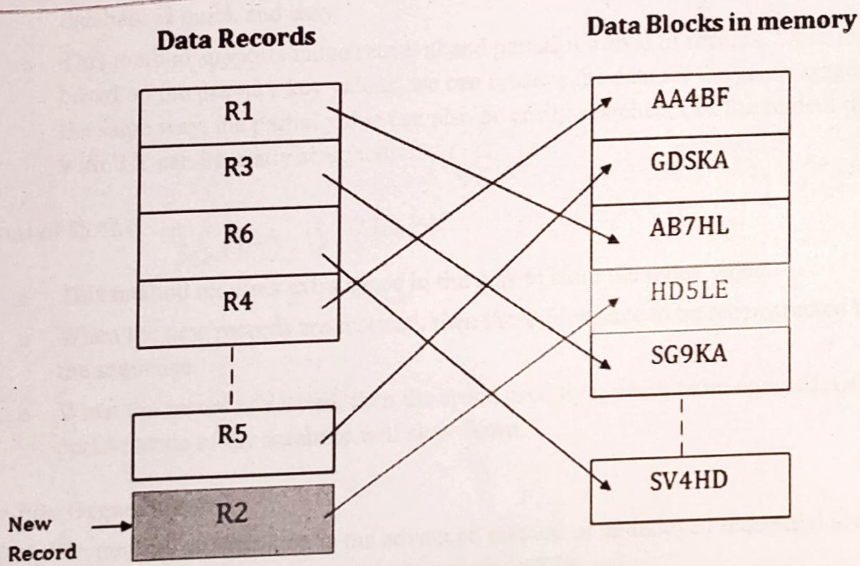
8

(13)



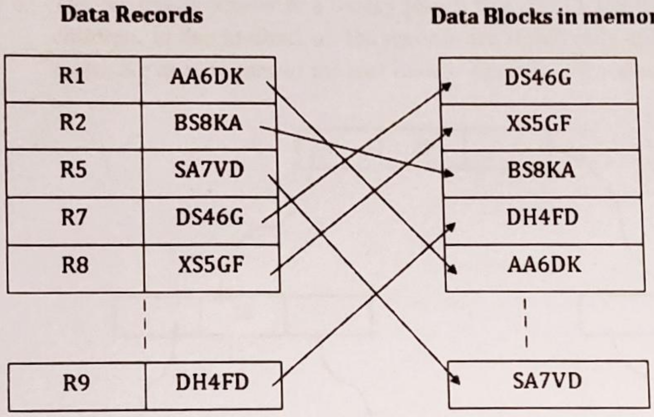
- When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address. In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted. The same process is applied in the case of delete and update.

In this method, there is no effort for searching and sorting the entire file. In this method, each record will be stored randomly in the memory not a sequential order,



Indexed sequential access method (ISAM)

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and the record is retrieved from the memory.

Pros of ISAM: Retrieval can be an exact match, range of values, Part of key also supported.

- In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

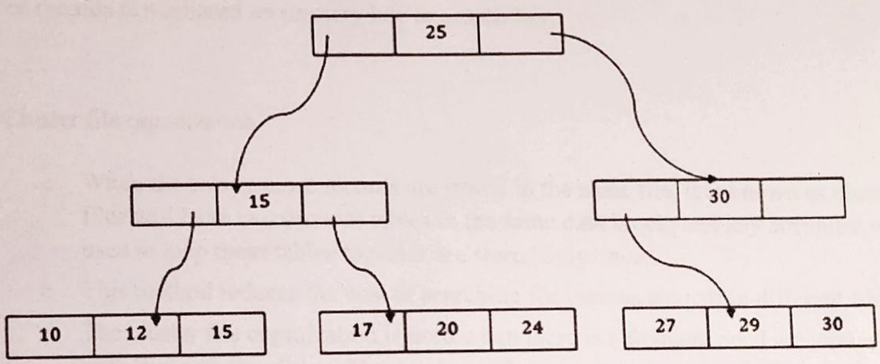
Cons of ISAM ISAM is Static

- This method requires extra space in the disk to store the index value.
- When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

B+ File Organization

- B+ tree file organization is the advanced method of an indexed sequential access method. It uses a tree-like structure to store records in File.
- It uses the same concept of key-index where the primary key is used to sort the records. For each primary key, the value of the index is generated and mapped with the record.

- The B+ tree is similar to a binary search tree (BST), but it can have more than two children. In this method, all the records are stored only at the leaf node. Intermediate nodes act as a pointer to the leaf nodes. They do not contain any records.



The above B+ tree shows that:

- There is one root node of the tree, i.e., 25.
- There is an intermediary layer with nodes. They do not store the actual record. They have only pointers to the leaf node.
- The nodes to the left of the root node contain the prior value of the root and nodes to the right contain next value of the root, i.e., 15 and 30 respectively.
- There is only one leaf node which has only values, i.e., 10, 12, 17, 20, 24, 27 and 29.
- Searching for any record is easier as all the leaf nodes are balanced.
- In this method, searching any record can be traversed through the single path and accessed easily.

Pros of B+ tree file organization   Dynamic, supports exact match, range of value etc.

- In this method, searching becomes very easy as all the records are stored only in the leaf nodes and sorted the sequential linked list.   Data is at leaf node, all other
- Traversing through the tree structure is easier and faster.   nodes are pointers present
- The size of the B+ tree has no restrictions, so the number of records can increase or decrease and the B+ tree structure can also grow or shrink.   in main memory.
- It is a balanced tree structure, and any insert/update/delete does not affect the performance of tree.

Cons of B+ tree file organization

- This method is inefficient for the static method.

11

Clustered File Organization

Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in the same disk block, that is, the ordering of records is not based on primary key or search key.

Cluster file organization

- When the two or more records are stored in the same file, it is known as clusters. These files will have two or more tables in the same data block, and key attributes which are used to map these tables together are stored only once.
- This method reduces the cost of searching for various records in different files.
- The cluster file organization is used when there is a frequent need for joining the tables with the same condition. These joins will give only a few records from both tables. In the given example, we are retrieving the record for only particular departments. This method can't be used to retrieve the record for the entire department.

Some db like oracle supports clustered, non clustered tables.
Clusters are group of 1 or more tables stored together as they share common attributes.
These related/shared col. are called cluster key.

-> A d/b consists of vast quantities of data, as this data must persist across, it is stored on external storage devices such as disks, tapes & ~~these~~ are fetched in to main memory as needed for processing.

-> The data in a d/b is mapped in to a no of files, these file will reside in the secondary memory.

-> The file is organized logically as a sequence of records. These records are mapped on to disk blocks or disk pages.

-> Each file which consists a no of records are logically partitioned in to fixed length storage units called data blocks ~~or pages~~. Most d/b use data block size of 4 to 8KB.

-> A block may contain records of same file or records of other files also based on size of data block.

-> And a file may be implemented to contain records of fixed length, records of variable length.

-> Each record in a file has a unique id called record id or rid. with this rid, we can identify the disk address of the page containing the record.

Definitions

Indexing, Indexes:

-> Indexing is a technique which helps us when we query a collection of records in multiple ways.

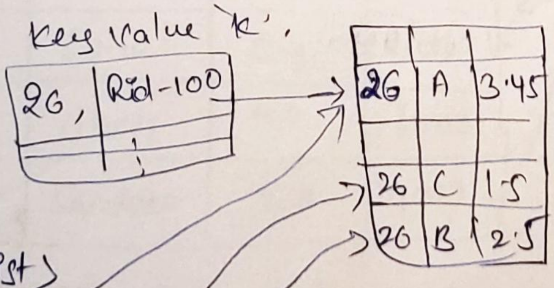
-> Indexing is a way to optimize performance of d/b by minimizing the no. of disk accesses required when a query is processed.

-> An index is a data structure consisting of 2 fields in which first field consists of search key values, 2nd field consists of a pointer to the records. The indexes are generally sorted, allowing binary search to be performed on it.

-> The entries present in the index file are called as data entry. A data entry with search key value 'k', contains enough information to locate data records with search key 'k'.

-> The different ways in which a data entry can appear in the index file are ① <K> - In this the index file contains the data records itself.

② <K, Rid>, where rid is record id of a data record with search key value 'k'.



③ <K, Rid list>

<59, rid-100, rid-61, rid-5>

Page (on disk block

There are 2 types of Indexes:

① ordered Indices - Based on sorted ordering of the values.

② Hash Indices - Based on the values being distributed across a range of buckets.

ordered Indices:

-> An ordered Index stores the values of the search keys in sorted order.

-> A file may have several indices on different search keys.

-> An index on a set of fields includes primary key then it is called a primary index.

-> The primary key attribute values are stored in a sorted order in the file or memory, and the entries in index file are also stored in sorted order.

-> An index on non primary attributes as search keys is called a clustered index. A clustered index some times may also have primary key attributes also

-> The clustered index is also ordered, and the data in memory is also stored in the sorted order of search key values.

Secondary Index: This index provides a secondary means of accessing a data file. A secondary index may be on a candidate key field or non prime attributes of a table.

→ To retrieve a record on the basis of non-prime key attributes, secondary index can be used for fast searching.

→ Secondary index must be dense with a index entry for every search key value and a pointer to every record of file.

Primary index
1
2
3
4
5
6
7
8
9

Primary index

ID	Name	Age	Salary
1	Amit	20	5000
2	Amit	21	6000
3	Sandeep	35	2500
4	Sonal	24	20,000
5	Sandeep	40	8000
6	John	21	18000
7	Sonal	24	15000
8	Jhon	40	6000
9	Sandeep	20	10,000

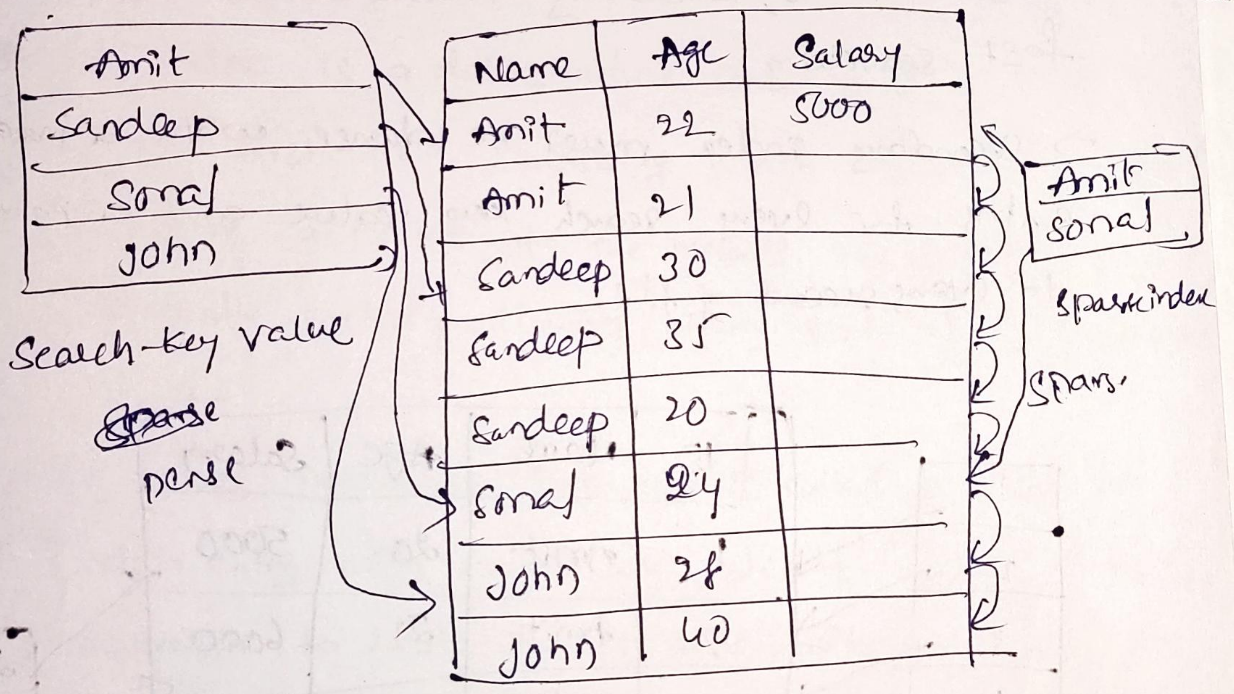
Secondary index
20
21
24
35
40

Secondary index

primary & secondary index

Non clustered Index :- It does not sort the physical data in the table.

→ In fact non clustered index is stored at one place and table is stored at another place.



Single level, Multilevel Indexing

Dense & Sparse Index - An ordered index can be dense index or sparse index.

→ follow the printout material for these topics.

Indexed Data structures:

→ There are 2 popular ways in which we can organize the data entries in an index.

① organize data entries using a hash function on search keys.

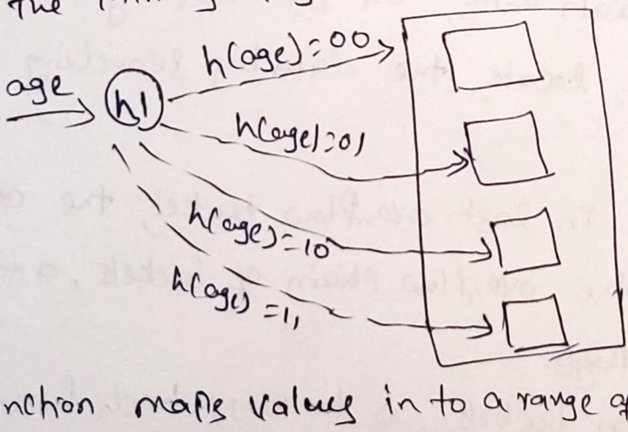
② organize data entries by building a tree like data structure.

① Hash Based Indexing:

→ In this, the records in a file are grouped in to buckets, where a bucket consists of a primary page, additional pages linked in a chain.

→ The bucket to which a record belongs can be determined by using hash function on search key.

→ So, given a bucket no, a hash based index allows us to retrieve the primary page for the bucket.



→ The hash function maps values in to a range of bucket numbers to find the page on which a desired data entry belongs.

→ Hash based techniques are

① Static Hashing

② Dynamic Hashing → Extendible Hashing
→ Linear Hashing.

Static Hashing:

→ A file consists of 0 to N-1 buckets, with 1 Primary Page per bucket initially.

→ To search for a data entry, we use hash function 'h' to identify the bucket to which it belongs & search this bucket.

→ To insert a data item, use hash function to identify the correct bucket and then put the data entry there.

→ If there is no space in the bucket, d/b allocates a new overflow page and puts the new data item in this page, and adds this page to the overflow chain of the bucket.

→ To delete a data entry, we use hashing fun. to ~~apply~~ identify correct bucket, locate the data by searching the bucket & then remove it.

→ If the data is in last overflow bucket, the overflow page is removed from the overflow chain of buckets, and added to a list of free pages.

→ If we have 'N' buckets, 0 to N-1, a hash fun. of the form h(value) = a * value + b is used, where a, b are constants selected

(x)

→ As n.of buckets in static hashing is known, as the file grows successive long overflow chains are developed.

→ The main problem with static hashing is n.of buckets is fixed.

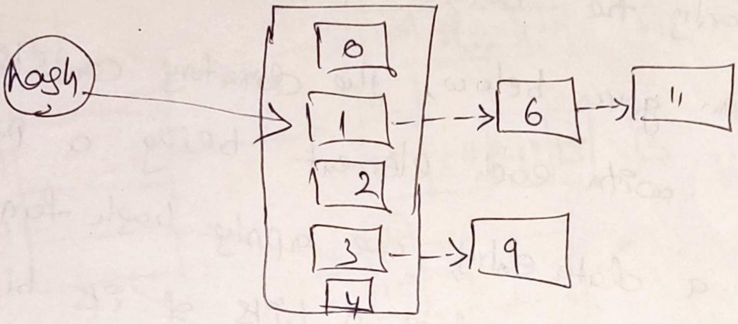
→ If a file shrinks greately, lot of space is wasted, if file grows a lot, long overflow chains will result.

→ Bucket overflow ~~occurs~~ occurs when a data item to be inserted leads to bucket which is already full.

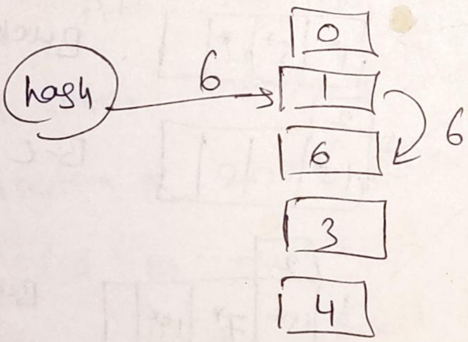
7

→ ~~Buffer~~ Bucket overflow is the condition where the bucket is full of records. In this case we can use either overflow chaining (or) Linear Probing.

Overflow chaining: when buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is also called as closed hashing.



Linear Probing: when hash fun. generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called as open hashing.



Dynamic Hashing:

→ Problem with static hashing is that it does not expand (or) shrink dynamically as the size of d/b grows (or) shrinks.

→ Dynamic hashing provides a mechanism in which data buckets are added & removed dynamically and on demand.

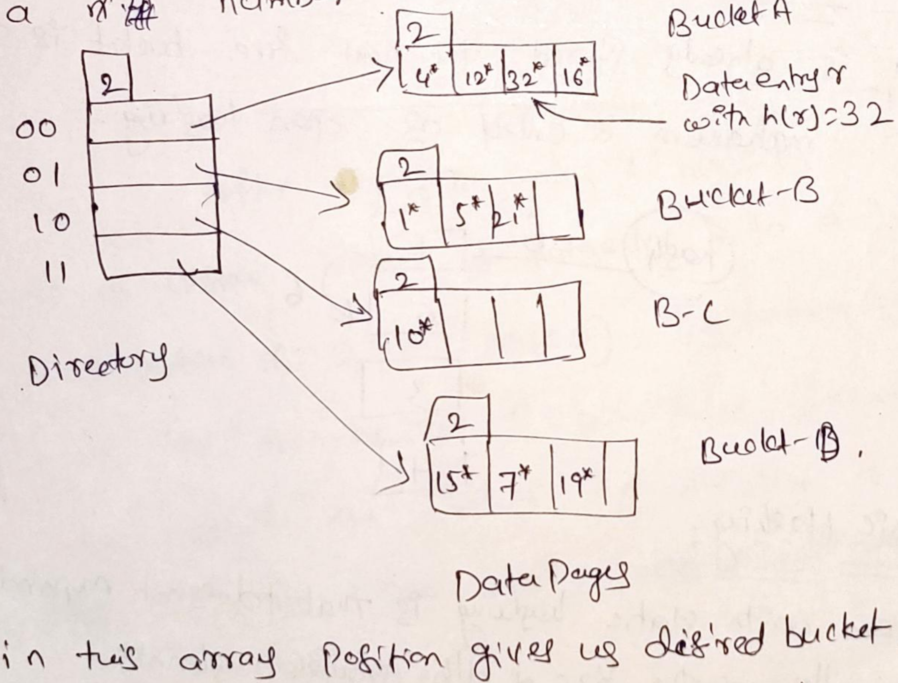
Extendible Hashing:

→ when we want to insert into a full bucket, one soln. to this is reorganize the file by doubling the n. of buckets and redistributing the entries across the new set of buckets.

→ we can use a directory of pointers to buckets, and double the size of the n. of buckets by doubling just the directory & splitting only the bucket that overflowed.

→ Consider an: given below, The directory consists of an array of size 4, with each element being a ptr. to a bucket.

→ To locate a data entry, we apply hash fun. to the search field and take the last 2 bits of its binary representation to get a number betn. 0 & 3.



→ The ptr. in this array position gives us desired bucket. we assume that each bucket can hold <u>four</u> data entries.

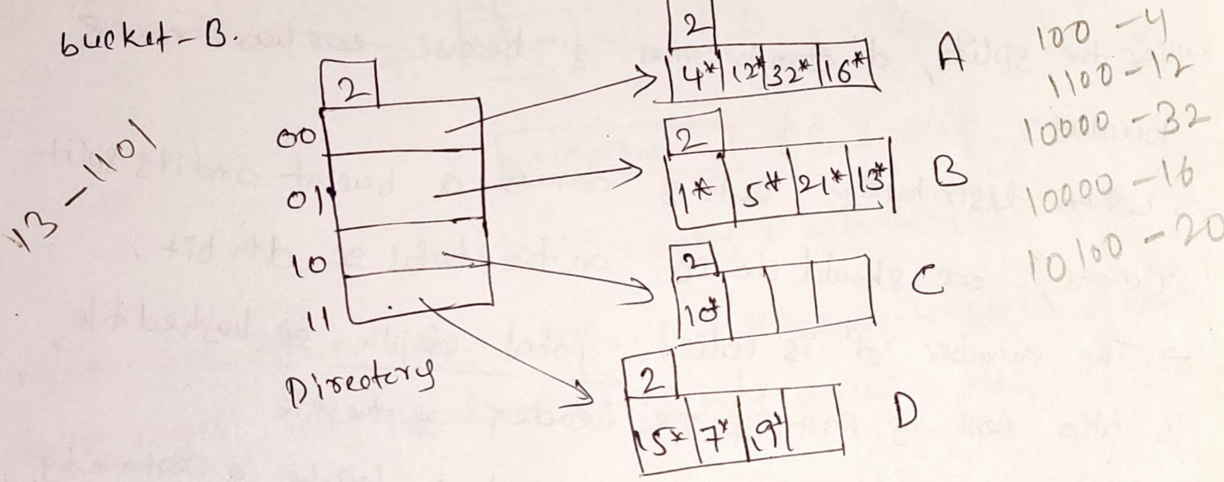
→ Therefore, to locate a data entry with hash value 5 (101), we look at directory element 01 & follow ptr. to the

1101 - 13

data page (Bucket-B)

→ To insert a data entry, we search to find the appropriate bucket.

→ so for, ex:, to insert a data entry with hash value 13 (13*) we examine the directory element 01 & go to page containing data entries 1*, 5*, 21*, as space is there the 13* is inserted in bucket-B.



Datapages

After inserting entry 'r' h(r) = 13

insertion of dataentry in to a full bucket.

→ Now, let us consider insertion of 20* (10100). By looking at directory

→ Consider the insertion of 20* (10100). By looking at directory 00, we are led to bucket - A, which is already full.

→ So, we must split the bucket by allocating a new bucket, ~~page~~ ~~that is~~ ~~being~~ (~~split~~ (~~and doubling the directory~~ ~~array~~.

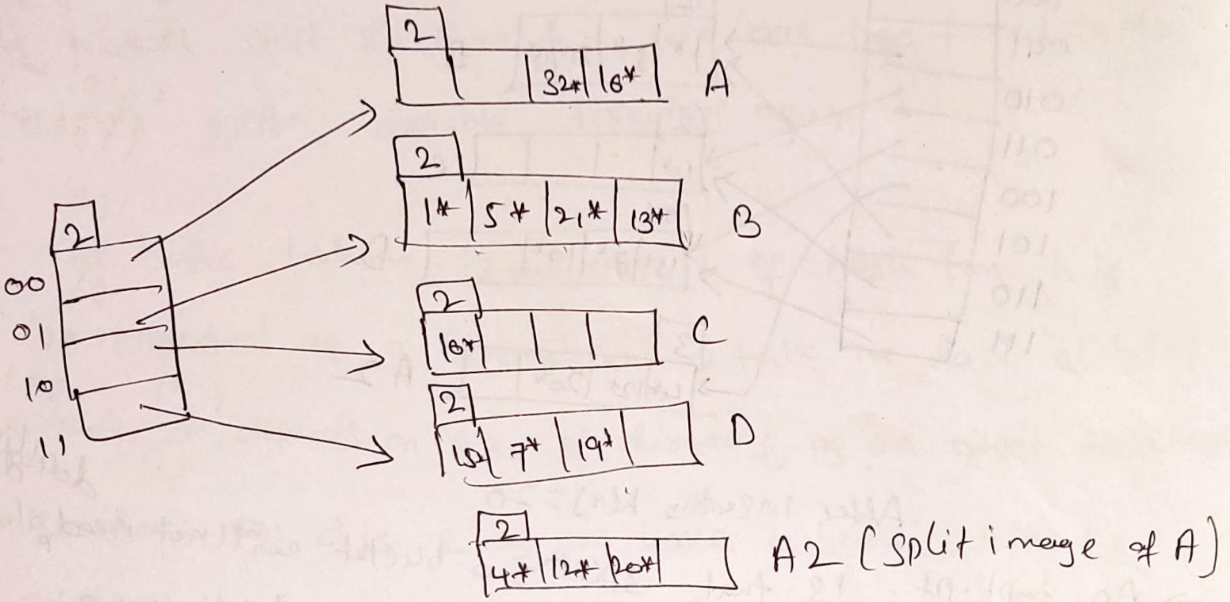
and redistributing the contents across old bucket & its 'split image'

split image:

-> To redistribute entries across old bucket and its split image, we consider last 3 bits of h(r).

-> The last 2 bits are 00, indicating a data entry that belongs to one of these 2 buckets, 3rd is used to identify the actual bucket in these 2.

→ The redistribution of entries is shown below.



while inserting entry 'r' with h(r) = 20

→ But actually, we need 3 bits to discriminate betn 2 of our data pages, but the directory has only enough slots to store all 2-bit patterns.

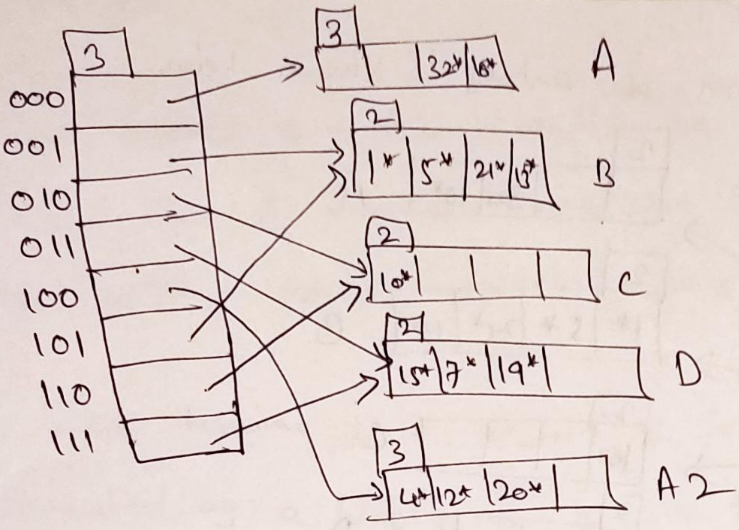
→ So, we do the doubling of directory.

→ Elements that differ only in 3rd bit from the end are said to correspond.

→ The corresponding elements of directory pt. to same bucket ~~with the~~

→ In this ex: bucket 0 is split, so new directory element 000 pts to one split, 100 pts- to other.

→ After doing all these steps the is shown in next fig.



001 - 1
101 - 5
10101 - 21
1101 - 13
1001 - 9

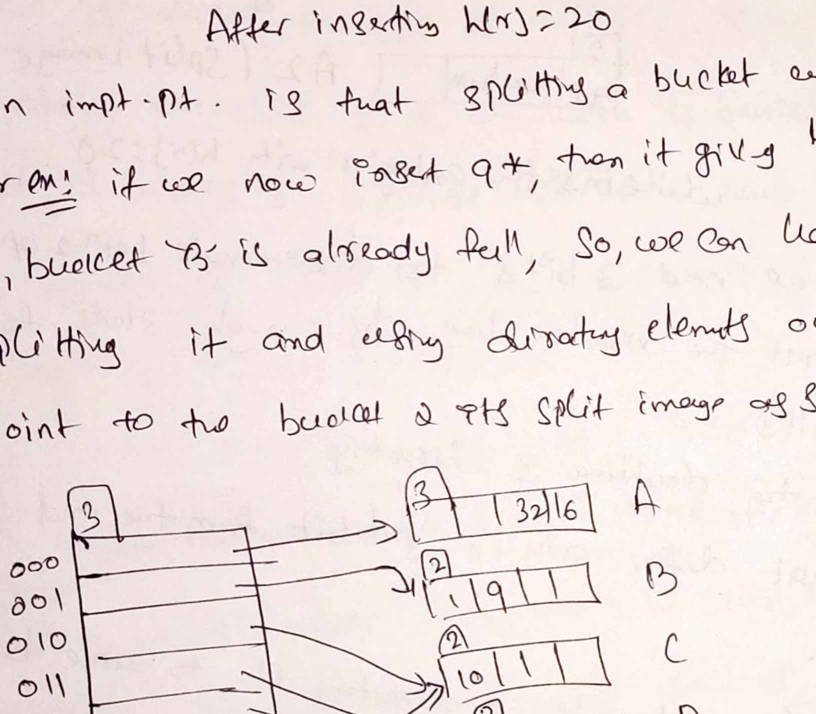
01
001 - B
001 - B)
1, 9
5,21,13

After inserting h(r) = 20

-> An impt. pt. is that splitting a bucket will not need doubling directory always.

-> For ex: if we now insert 9*, then it give by bucket B,

So, bucket 'B' is already full, So, we can handle this by splitting it and letting directory elements 001 & 101 to point to the bucket & its split image as shown below.



After inserting entry with h(r) = 9.

-> So, a bucket split does not necessarily need a directory doubling.

-> If either bucket A or A2 is full & an insert leads to a bucket split of A or A2, we are then forced to do ~~bucket split~~, double directory again.

-> The basic technique, is the result of hash fun. h is represented as a binary no., & take the last 'd' bits, where 'd' depends on size of directory, as an offset to directory.

-> In above exm: d is 2 as we have 4 buckets. After the split d becomes 3 because we have now 8 buckets.

-> The number 'd' called a global depth of hashed file, is kept as part of header of the file.

-> It is used every time we need to locate a data entry.

-> we also maintain a local depth value for each bucket.

* -> If a bucket whose local depth is equal to global depth is split, the directory must be doubled.

-> So, in case of inserting 9*, local depth is 2, global depth is 3, so directory is not doubled.

-> Initially both values are equal, which is the number of bits needed to express the total n of buckets.

-> we increment the global depth by 1 each time directory is doubled.

-> And, whenever a bucket is split, we increment its local depth by 1, and assign same new value to its split image.

→ A total of 2^{d-l} directory elements pt. to a bucket with local depth 'l'.

→ when we want to delete a data record, it is located & removed. If the delete leaves the bucket empty, it can be merged with its split image.

Linear Hashing :

→ The linear hashing does not require a directory.

→ It uses a family of hash functions h_0, h_1, h_2 -- with the property that each function's range is twice that of its predecessor.

→ That is, if h_i maps a data entry in to M buckets, then h_{i+1} maps in to 2M buckets.

→ The hash fun. can be choosen is $h_i(value) = h(value) \bmod (2^i N)$.
N is ~~no of~~ intial n. of buckets.

→ The approach can be used best in terms of rounds of splitting.

→ During a round number 'r', only hash functions h_r, h_{r+1} are in use.

→ The buckets in the file, at the begining of round are split, one by one from first to last bucket, thereby doubling the n. of buckets.

→ At any given pt. in a round, we have some buckets that are split, buckets that are not yet split, buckets created by splits in this round.

→ Now Consider the case, of searching for a data record.

→ we apply hash function h_r, and if it directs to unsplitted buckets, we will search in that bucket.

→ If the directed bucket is a splitted bucket, the record may be in any two buckets of original one, that is in so to determine which of these two buckets contain the entry, we apply the h_{r+1} function.

→ Now, Consider when an insert of a record lead to split of bucket,

→ In linear hashing, a Counter 'c' is used to indicate the current round number and is initialized to 0.

→ The bucket to split is denoted by Varible Next & is intially the first bucket.

→ we denote the n. of buckets in file at the begining of a round r by N_r. r is round number. N - intial n. of buckets

→ And $N_r = N * 2^r$ (or) 'c' is a " " $N_2 = 4 \times 2^2 = 4 \times 4 = 16$

→ Let the n. of buckets at the begining of round-0 is N_0, be N.

→ lets take a eg:

h_1	h_0
000	00
001	01
010	10
011	11

round = 0 N = 4

Next → 0 | 32* | 44* | 36* | |

| 9* | 25* | 5* | |

| 14* | 18* | 10* | 30* |

| 31* | 35* | 7* | 11* |

The actual contents of linear hashed file

16

→ when ever an insertion of a record leads to split, the bucket pointed by Next is split, and hash fun. h_{L+1} (or) h_{r+1} redistributes entries betⁿ this bucket & its split image.

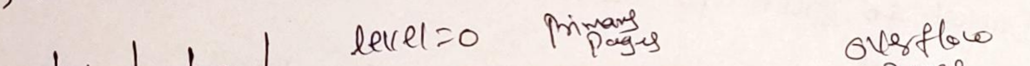
→ If current bucket no is b, then split bucket no is $b + N_r$. $r=0$

if b. No is 1 $1 + N_0 = 1 + 4 = 5$.

→ After splitting a bucket, the value of Next is incremented by 1.

→ In the ex. insertion of 43* leads to split. It 101011 leads to split of bucket pointed by Next only, not by 11 to bucket pointed by 11. we insert 43 to bucket - 4 pointed by 11 by using overflow method. And bucket - 1 is splitted to 2 buckets.

→ The above. ex.. after Completing insertion looks as below.



After inserting h(r) = 43.

→ At any time, in the middle of a round 'r' (or), all buckets above Next have been split, and the file Contains buckets that are their split images as shown above.

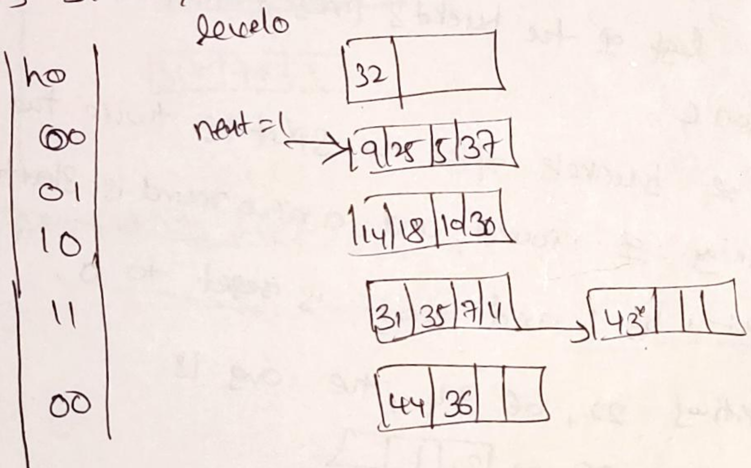
→ Buckets Next through N_C have not yet been split. If we use h_C on a data entry & obtain a number 'b' in ~~to~~ the range Next through N_C, the data entry belongs to bucket b.

→ And if we obtain a number b in the range 0 through Next, the data entry may be in bucket or in its split image.

→ Then we have to use h_{r+1} to determin to which of these 2 buckets the data entry belongs.

* Not all insertions lead to split. If we insert 37*, the 2nd bucket pointed by 01 has space, so it is inserted.

100101
01



after inserting h(r)=37

→ If the bucket pointed by Next is full, a new data entry should be inserted in this bucket. where split is needed, we don't ~~do~~ ~~the~~ use overflow, we split the bucket.

→ If we want to inset 29*, it leads to split of 01 bucket-2.
11101-29

h_0	level = 0
00	[32 | | |]
01	[9 | 25 | |]
10	Next=2 → [14 | 18 | 10 | 30]
11	[31 | 35 | 7 | 11] → [43 | | |]
00	[44 | 36 | |]
01	[5 | 37 | 29 |]

after inserting $h(r) = 29$

→ when Next is equal to $N_c - 1$ & insertion leads to split, we split the last of the buckets present in the file at the begining of round r or c.

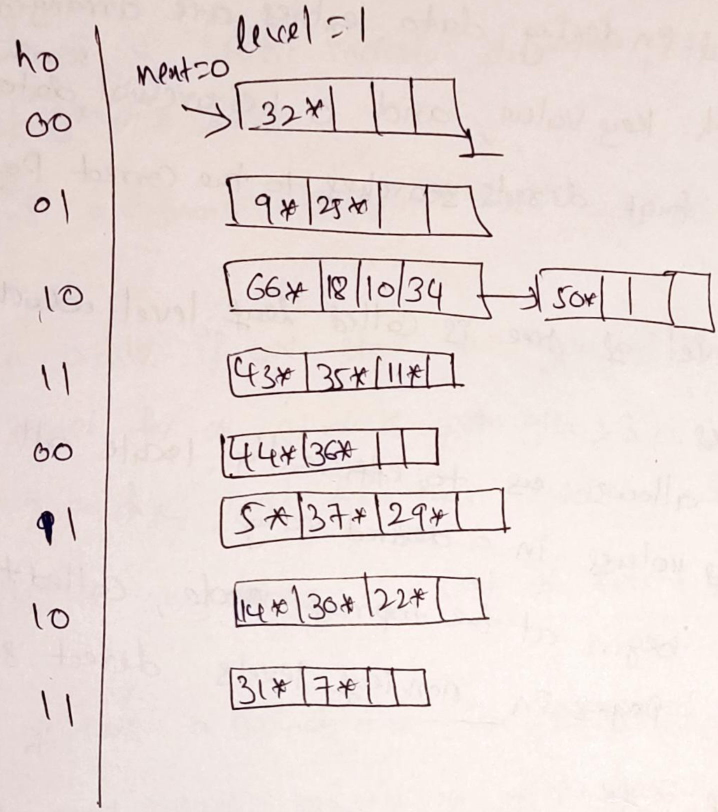
→ The n of buckets after the split is twice the number at the begining of round, and a new round is started with r or c incremented by 1 and Next is reset to 0.

→ By inserting 22, 66, 34, the org. is

00	→ [32 | | |]
01	→ [9 | 25 |]
10	→ [66 | 18 | 10 | 34]
11	→ [31 | 35 | 7 | 11] → [43 | | |] Next=3
00	→ [44 | 36 | |]
01	[5 | 37 | 29 |]
10	[14 | 30 | 22 |]

→ And inserting 50 causes a split that leads to increment of r on C by 1.

fig.



After inserting h(r)=50

Tree Based Indexing:

-> In tree based indexing data entries are arranged in sorted order by search key value, and a heirarchial search data structure is maintained that directs searches to the correct Page of data entries.

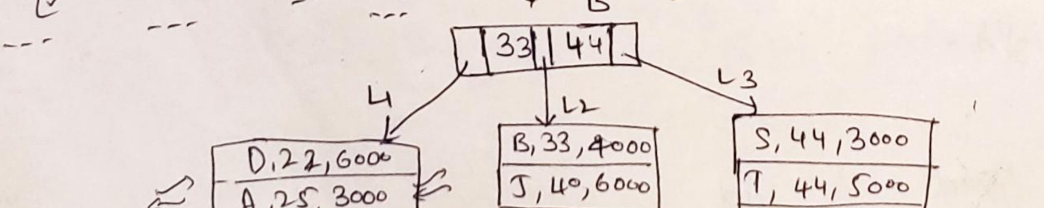
-> The lowest level of tree is called leaf level which contains the data entries.

-> This structure allows us to efficiently locate all data entries with search key values in a desired range.

-> All searches begin at the topmost node, called the root, and direct searches to the contents of pages in non-leaf levels the correct leaf Page.

-> ~~Node~~ Non-leaf Pages contain node pointers separated by search key values.

-> The node pointer to the left of a key value 'K' points to a subtree that contains only data entries < K, & to the right of K points to subtree contain only data entries > or = K.



The two index data structures are (1) ISAM (2) B+ trees.

ISAM - Indexed sequential Access Methods.

-> The ISAM is a static indexed structure, it is not suitable for files that grow & shrink.

-> B+ tree is a dynamic structure that adjusts to change in file.

-> For ex, in a file if we store the gpa of students, when we want to search for a students with gpa > 3.0, we do a binary search on file and from that mid pt. we scan the file.

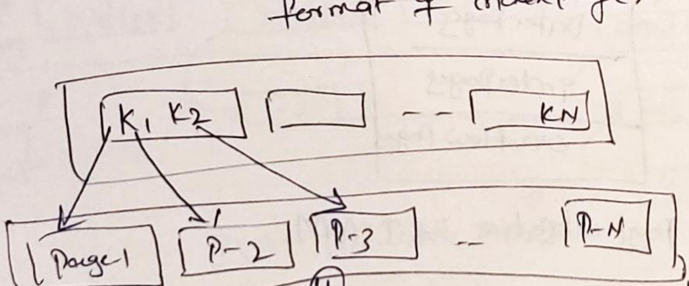
-> If the file is very large, the cost of fetching the n. of pages increases.

-> So, instead of this a second file can be created with one record per data page and each record is of the form <first key on page, ptr to page> (K, P) and this second file is also sorted by the order of key attribute.

-> We refer to the pairs of the form as index entries, and each index page contains one ptr. more than the n. of keys.

$$\boxed{P_0 | K_1 | P_1 | K_2 | P_2 | \dots K_m | P_m}$$

format of index page.



index file

Datafile

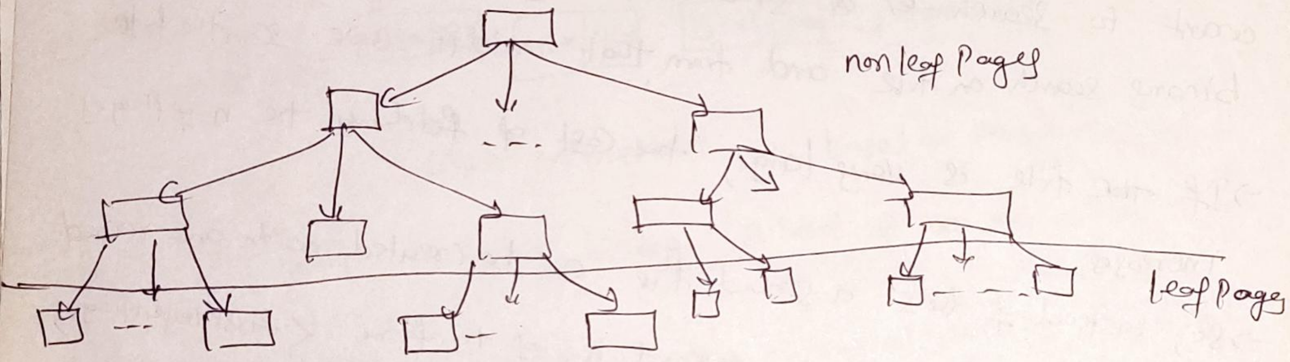
one level index structure

→ The size of index file is smaller when compare to data page.

→ So, the size of index file can be used to represent the data pages (or) file in tree indexing form, which may contain several levels of non-leaf pages instead of 1-level.

ISAM (Indexed Sequential Access Method):

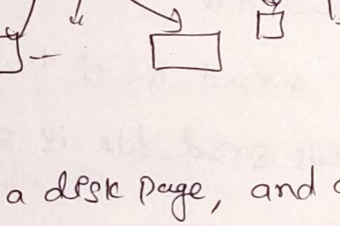
→ The data entries of the ISAM index are in leaf pages of tree and additional overflow pages chained to leaf pages as shown below.



→ Each tree node is a disk page, and all the data is stored in leaf pages.

→ When the file is created, all leaf pages are allocated sequentially & sorted on search key value. The non leaf pages are then allocated.

→ If more insertions are done, additional pages are allocated from an overflow area.

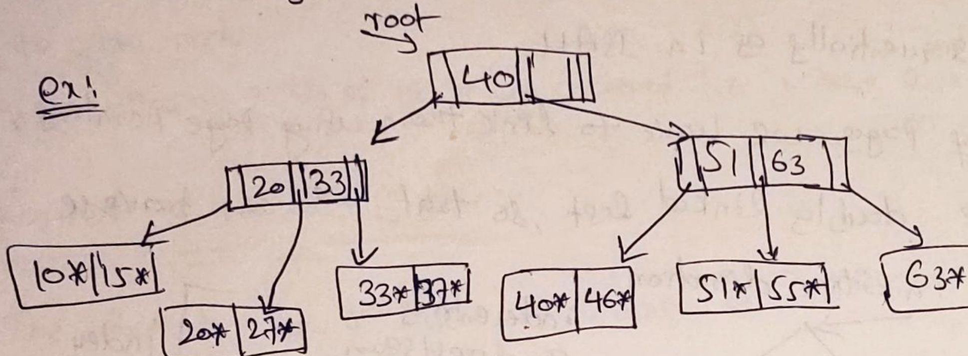


Page allocation in ISAM.

→ The insertion, deletion, search, we start at root & determine which subtree to search (or) insert or delete by comparing the value in the search field of the given record with key values in the node.

→ for a range selection, in the same way data pt. is determined, and data pages are then retrieved sequentially.

ex:



→ ISAM is best one when retrievals are based on exact key match, pattern matching, range of values & part of key specification

→ ISAM is static, is created when file is created, and if we ~~done~~ have more insertion in a file, which may lead to single leaf node, it leads to more n of lin overflow pages to a single leaf node

→ This will result in increase in time to retrieve a record because of overflow chains.

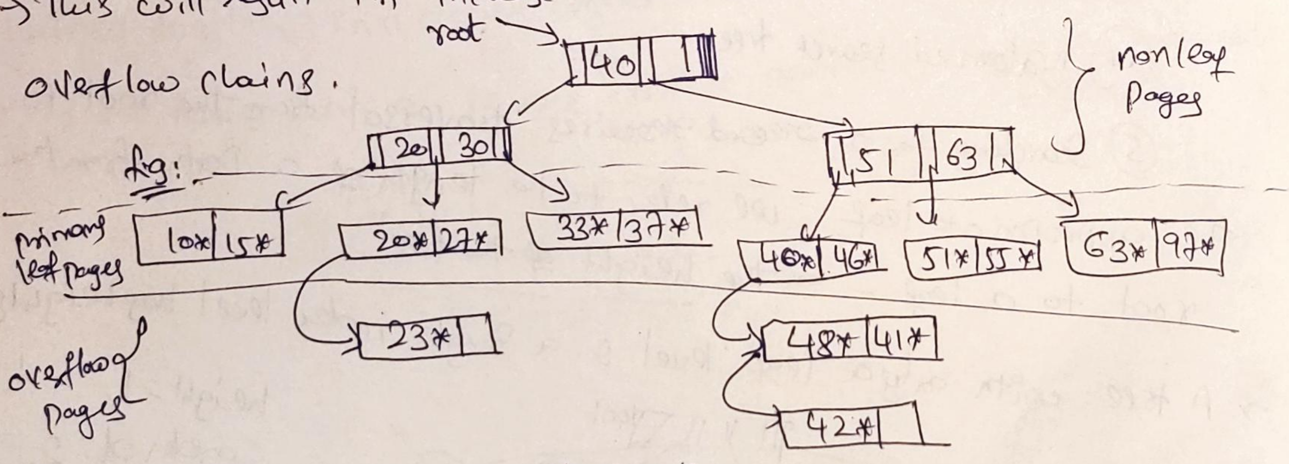


fig:

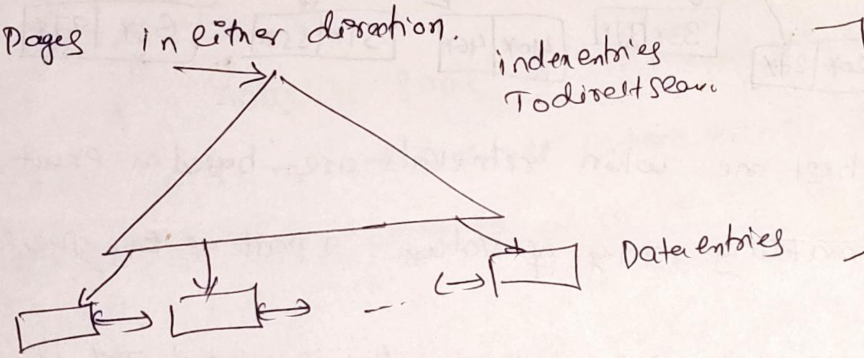
ISAM Tree after inserts.

B+Trees: A Dynamic Index structure:

→ The B+tree is a balanced tree in which the internal nodes direct the search & leaf nodes contain data entries.

→ As tree structure grow & shrinks, it is not feasible to allocate the leaf pages sequentially as in ISAM.

→ To retrieve leaf pages, we have to link them using page pointers. for this we use doubly linked list, so that we can traverse the leaf pages in either direction.

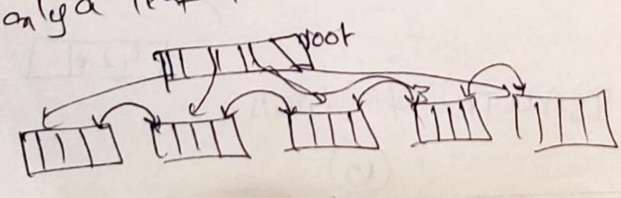


→ Main characteristics of B+tree:

1. Operations on tree kept it balanced.
2. All paths from root to leaf are of same length.
3. All leaf nodes are on same level.
4. Balanced search tree
5. Searching for a record requires traversal from the root to the appropriate leaf. we refer to the length of a path from the root to a leaf - as the height of the tree.

→ A tree with only a leaf level & a single index level has height 1

height = 1
order d = 2



→ A tree that has only root node has height 0.

→ The height of a B+ tree is rarely more than 3 or 4. In a B+ tree every node contains 'm' entries, where $d \leq m \leq 2d$.

→ d is called order of tree and it is measure of capacity of a tree node.

→ For a root node, the n. of entries required is $1 \leq m \leq 2d$.

→ Nodes consist of 'N' key values, N data pointers, N+1 node pointers pointing to other node.

→ A multiway order of m is an ordered tree where each node has at most 'm' children.

Format of a Node in B+ tree:

→ The format of a node is same as ISAM node.

→ The format of a node is same as ISAM node contain m+1 pointers to children.

→ Non leaf nodes with m indent entries contain m+1 pointers to children.

→ Pointer P_i points to a subtree in which all key values 'K' are such that $K_i \leq K < K_{i+1}$.

→ P_0 points to a tree in which all key values are less than K_1, & P_m points to a tree in which all key values are greater than or equal to K_m.

→ Leaf nodes contain data entries in K* form, and all leaf nodes are chained together in a doubly linked list.

→ As the leaves form a sequence, where range queries can be answered efficiently.

P_0	K_1	P_1	K_2	P_2	---	K_m	P_m

Format of index page.

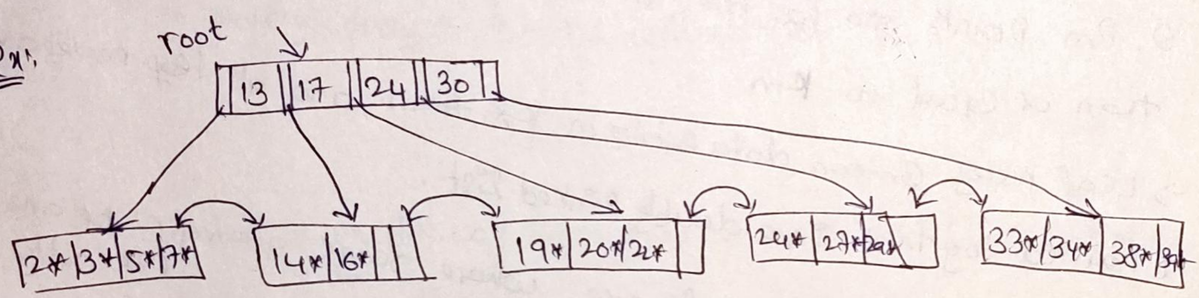
SEARCH operation on a B^+ tree:

→ The algo. for search finds the leaf node in which a given data entry belongs.

```
func find(searchkey k) returns nodeptr
    return treesearch(root, k);    //searches from root
endfunc

func tree_search(nodepointer, searchkey k) returns nodeptr
    if *nodepointer is a leaf, return nodepointer;
    else
        if K < K1 then return treesearch(P0, k);
    else
        if K >= Km then return treesearch(Pm, k)
    else
        find i such that Ki <= k < Ki+1;
            return tree_search(Pi, k);
endfunc
```

ex:



B^+ Tree Insertion

① Builds a tree from bottom-up, tree height grows as items are inserted.

② Find a leaf for a new item.

③ Start from root, if the root is a leaf node, if there is room in leaf insert the data item.

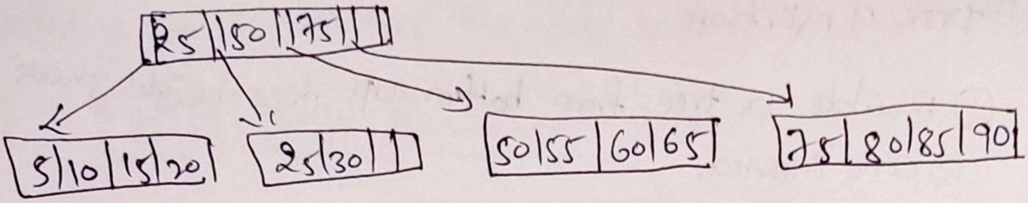
④ If leaf is full, split the node in to 2 halfs or leaves after arranging in sorted order, then copy and insert the middle element in to the (internal) parent node.

⑤ If leaf node parent is full, split the index page in to 2 halves after sorting, then find the middle element, move this middle element in to parent of the current index page.

-> when non leaf node is full, after finding middle element, elements left of middle form left node, elements right of middle will form right node.

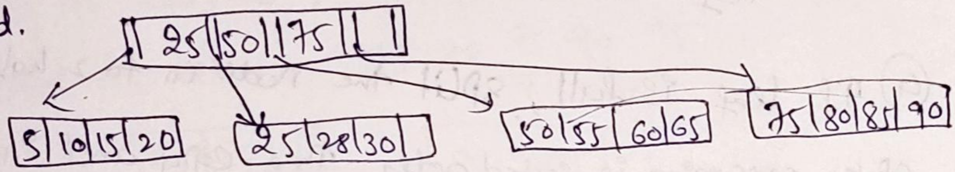
⑥ Rearrangement of pointers, data items will will be done after splitting a leaf node or non leaf node.

Example -1



Insert 28

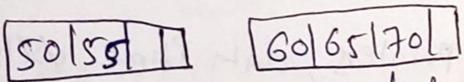
→ Inserting 28 leads to 2nd leaf Page, as space is there 28 is inserted.



Insert 70

→ Inserting 70 leads to 3rd leaf Page, ~~will~~ which is full, so it has to be splitted. The node with new item added before splitting is 50 55 60 65 70,

from the above take the middle element, which is 60 in this case, ~~above~~ place the elements before mid element in the original node, place the middle element along with remaining elements in to a new leaf node. As shown below

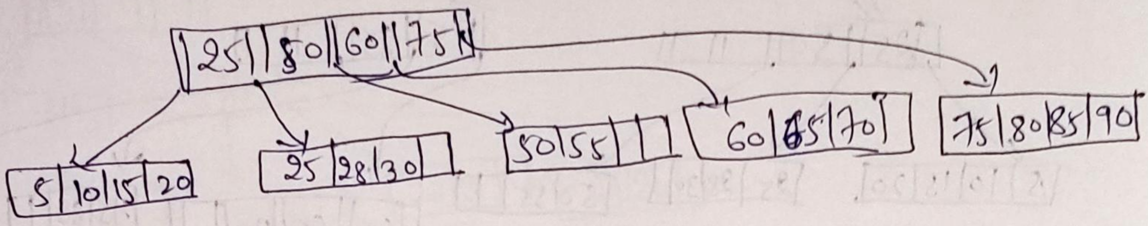


old leaf new leaf

→ Now, create a link for the new leaf by coping the middle element into the parent of the current leaf page.

→ So parent now becomes | 25 | 50 | 60 | 75 |, Create the link from this parent to new leaf.

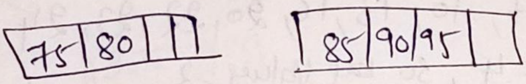
So, after inserting 70 in to tree, the tree becomes as shown below



Insert 95

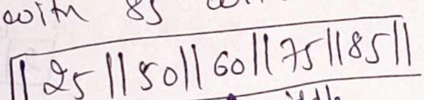
-> Inserting 95 leads to 5th leaf page, which is full, so it has to be splitted as below. ~~75~~ 80 85 90 95

↑ middle



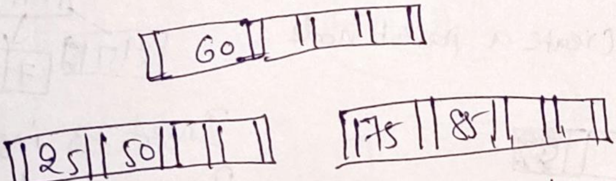
-> After this we have to make entry for middle key 85 in the parent of 5th leaf page, which is also full.

-> So, the index page has to be splitted now. Currently the index page with 85 will be



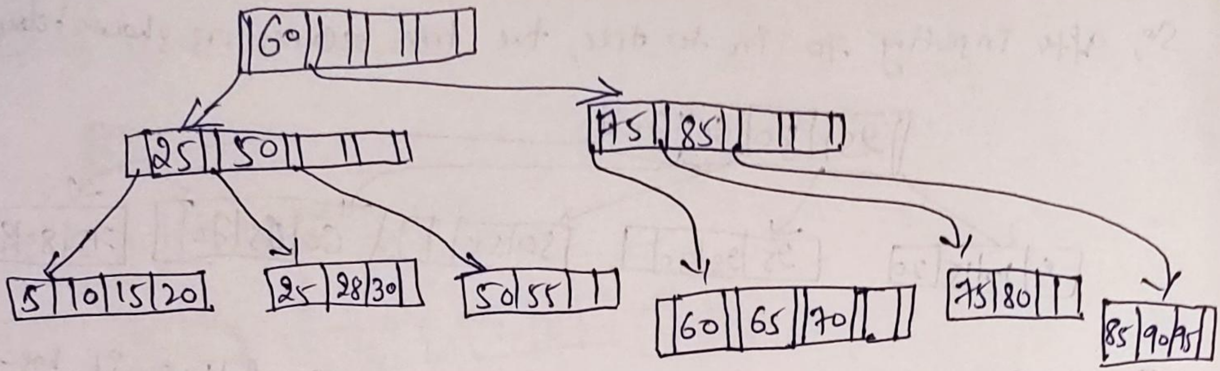
↑ middle

Take the middle element push it up to new parent.



-> And ~~arrange~~ place the elements left to middle key in the left node, place the elements right to middle key to right node. Here we won't copy the middle key in to right node.

-> once it is done link the leaf page to these nodes properly

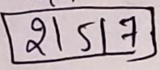


Qu:-2 Create B+ tree with key values

2, 5, 7, 10, 13, 16, 20, 22, 23, 24

N. of Pointers = 4, so key values = 3

insert 2, 5, 7



insert 10, leads to split of leaf.

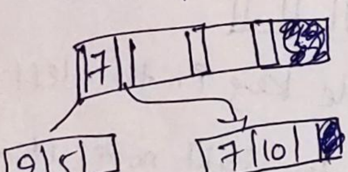
2 5 7 10

~~makes~~ do 2 halfs

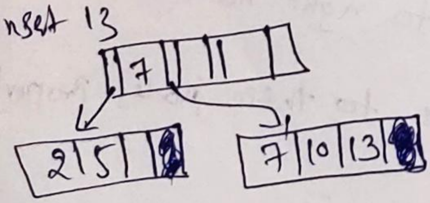
2 5   7 10

take, first element as middle key from 2nd half

It is 7, create a parent node with 7.



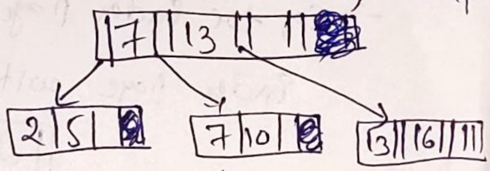
Insert 13



insert 16, leads to split of leaf.

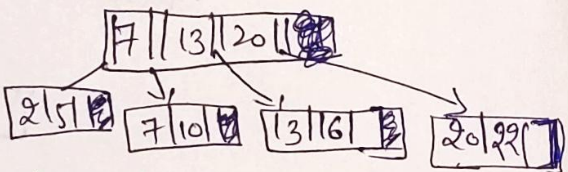
7 10 13 16

↑ middle, so copy to parent, new leaf.



Insert 20 - goes to 3rd page.

Insert 22, leads to split of 3rd page

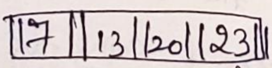


Insert 23 inserted into 4th leaf.

Insert 24, leads to split of 4th leaf.

20 22 23 24

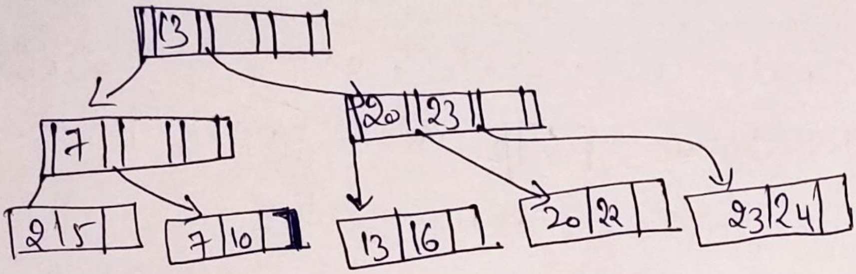
↑ middle, make entry of it in index page, it is full,



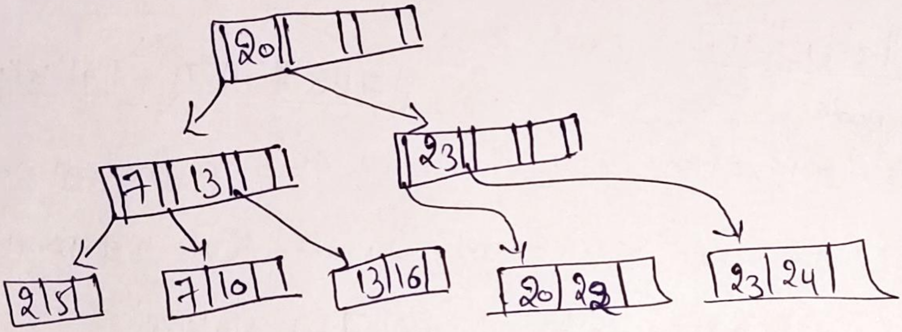
↳ full, middle key can be either 13 or 20

→ Any one can be taken as middle key

Taking 13 as key element

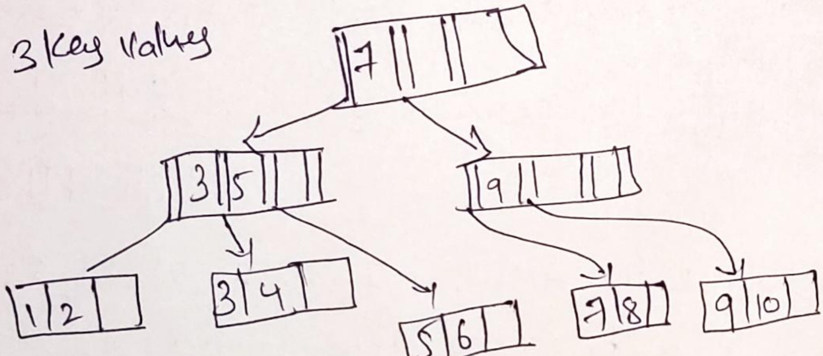


Taking 20 as middle key, the tree will be as below



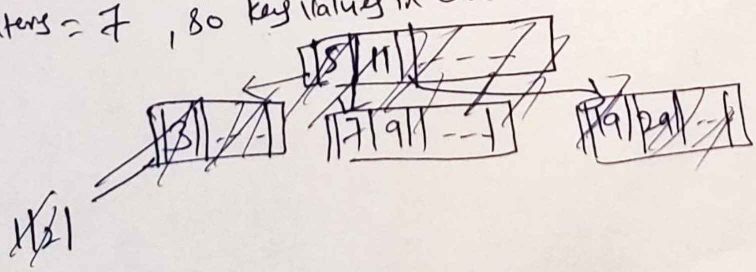
Ex-3 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

3 key values



ex:4 2, 3, 5, 7, 11, 17, 19, 23, 29, 31

No of Pointers = 7 , so key values in each node



→ when the data key values exceed the size of a leaf node, overflow happens, in that case to insert the data item

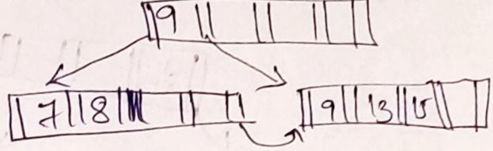
split the node in to 2 nodes ; (n – n. of Pointers in each node)

→ 1st node contains ~~n/2 values~~ $\lceil (n-1)/2 \rceil$ values.

→ 2nd node contains remaining values

→ copy the smallest search key value or data item of the 2nd node to the parent node.

ex: | 7 | 9 | 13 | 15 | insert 8 ⇒



Leaf node.

→ when an insert operation leads to overflow of nonleaf node

then split the nonleaf node to 2 nodes (n – n of pointers)

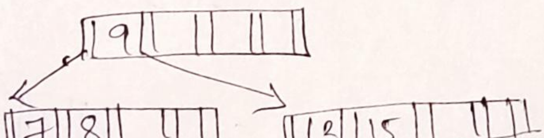
→ 1st node contains $\lceil n/2 \rceil - 1$ values

→ Move the smallest of the remaining elements along with a pointer to the parent

→ 2nd node contains the remaining values

Nonleaf node

| 7 | 9 | 13 | 15 | insert 8 ⇒



Ex: 5 Construct B+ tree for 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42; with n = 4 (n. of Pointers)

ISAM – Indexed Sequential Access Method

ISAM is a static index structure – effective when the file is not frequently updated. Not suitable for files that grow and shrink. When an ISAM file is created, index nodes are fixed, and their pointers do not change during inserts and deletes that occur later (only content of leaf nodes change afterwards). As a consequence of this, if inserts to some leaf node exceed the node's capacity, new records are stored in overflow chains. If there are many more inserts than deletions from a table, these overflow chains can gradually become very large, and this affects the time required for retrieval of a record.

B+ tree – a dynamic structure that adjusts to changes in the file gracefully. It is the the most widely used structure because it adjusts well to changes and supports both equality and range queries.

It is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries. The leaf nodes are organized into a doubly linked list allowing us to easily traverse the leaf pages in either direction.

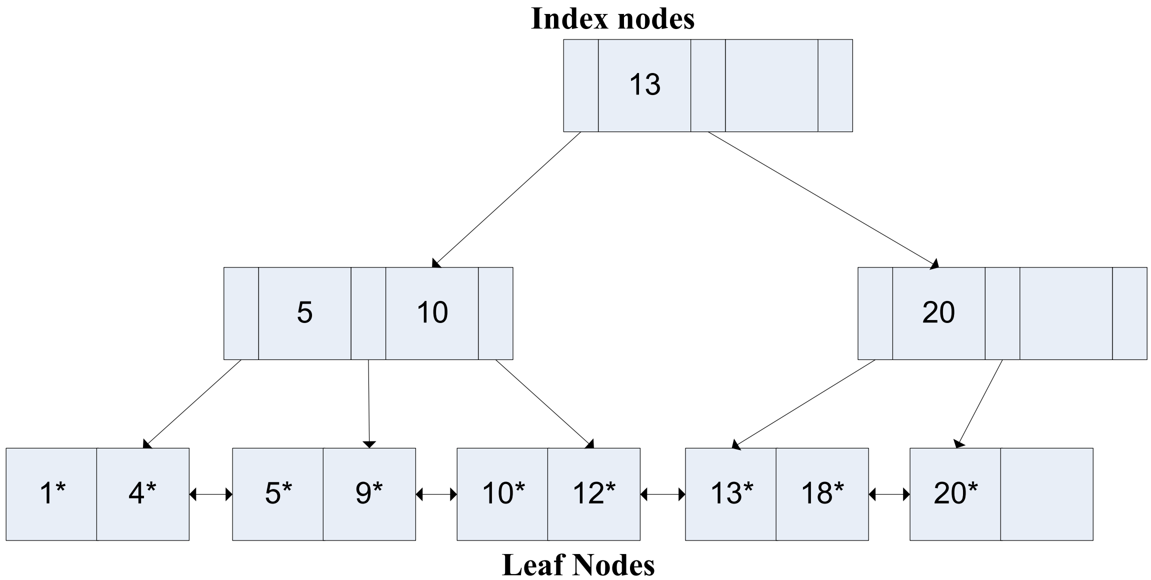
Main characteristics of a B+ tree:

- Operations (insert, delete) on the tree keep it balanced. $\text{Log}_f N$ cost where f=fanout, N = # of leaf pages.
- Minimum occupancy of 50% is guaranteed for each node except the root node if the deletion algorithm we will present is used. (in practice, deletes just delete the data entry because files usually grow, not shrink). Each node contains m entries where $d \le m \le 2d$ entries. d is referred to as the **order** of the tree.
- Search for a record is just a traversal from the root to the appropriate leaf. This is the height of the tree – because it is balanced is consistent. Because of the high fan-out, the height of a B+ tree is rarely more than 3 or 4.

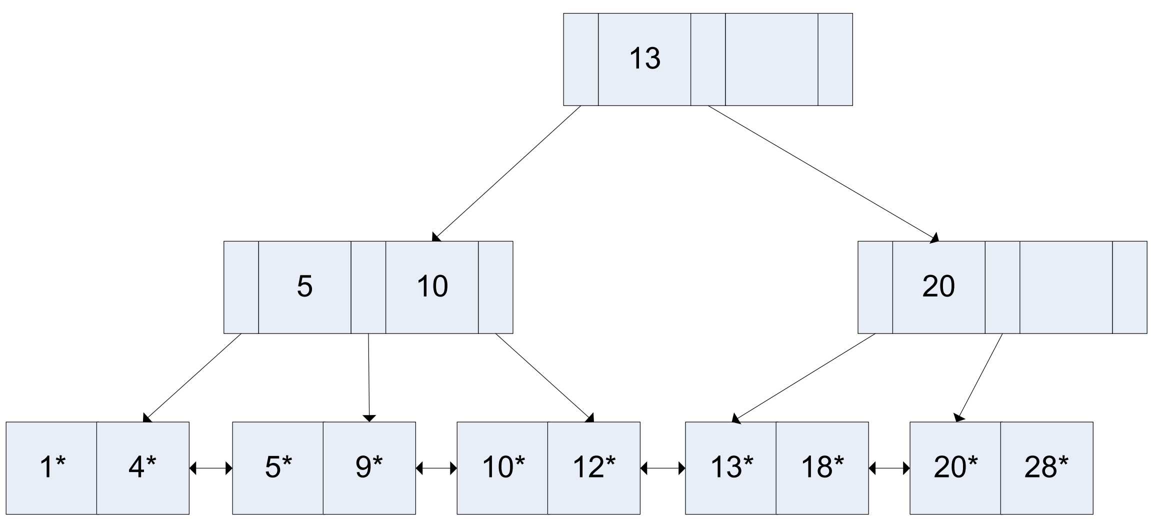
The `insert` algorithm for B+ Trees:

Leaf page full?	Index page full?	Action
No	No	Place the record in sorted position in the appropriate leaf page
Yes	No	1. Split the leaf page 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
Yes	Yes	1. Split the leaf page. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

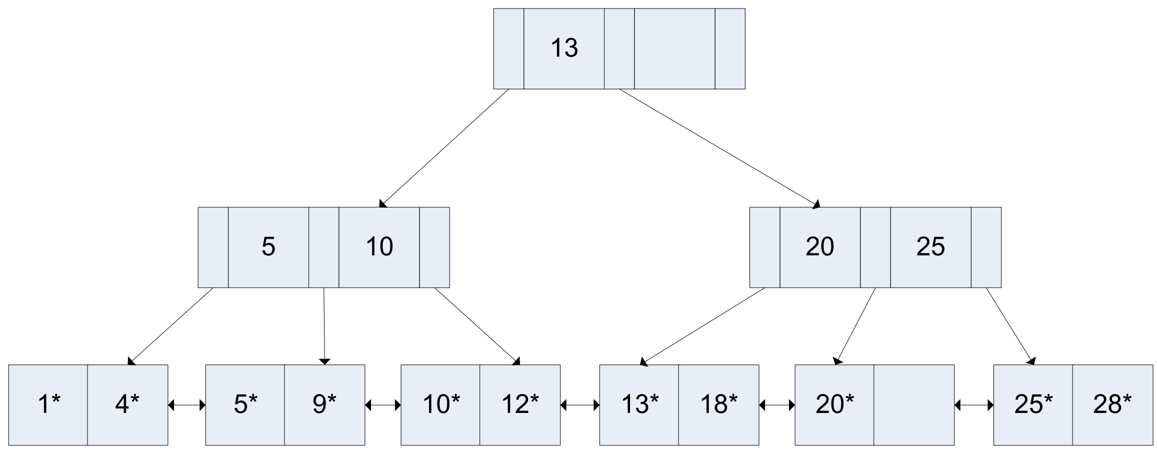
Examples of insertion with B+ tree with order = 1. Starting with a tree looking like this:



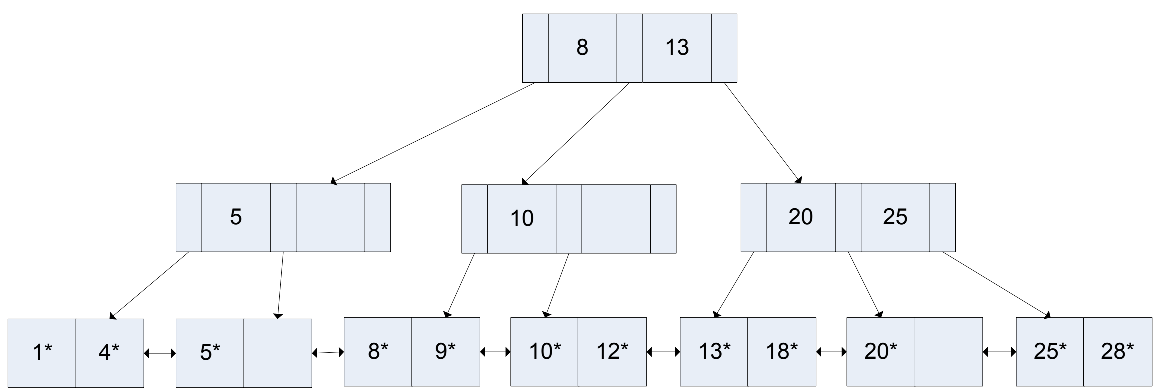
Our first insertion has an index of 28. We look at the leaf node to see if there is room. Finding an empty slot, we place the index in node in sorted order.



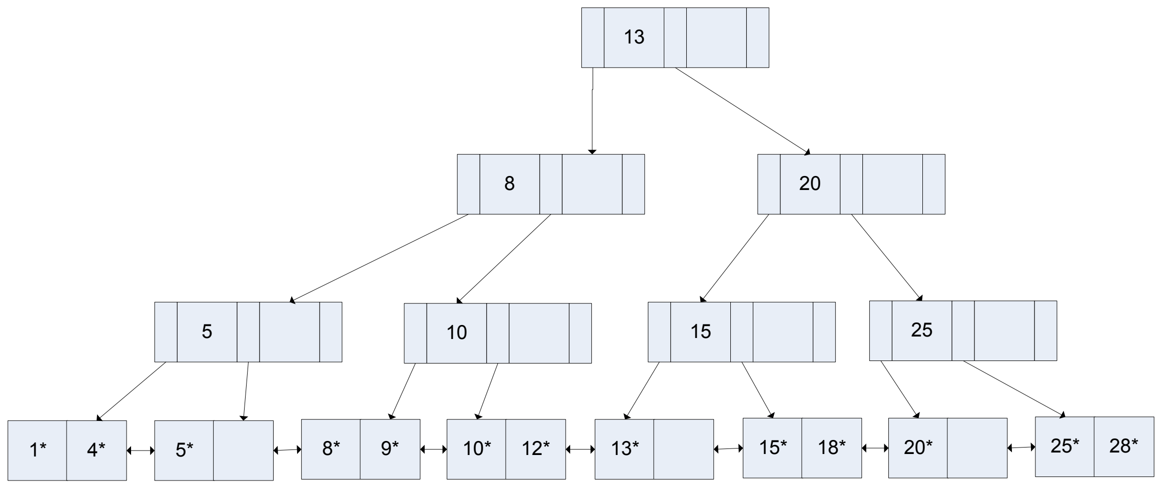
Our next insertion is at 25. We look at the leaf node it would go in and find there is no room. We split the node, and roll the middle value to the index mode above it.



Our next case occurs when we want to add 8. The leaf node is full, so we split it and attempt to roll the index to the index node. It is full, so we must split it as well.



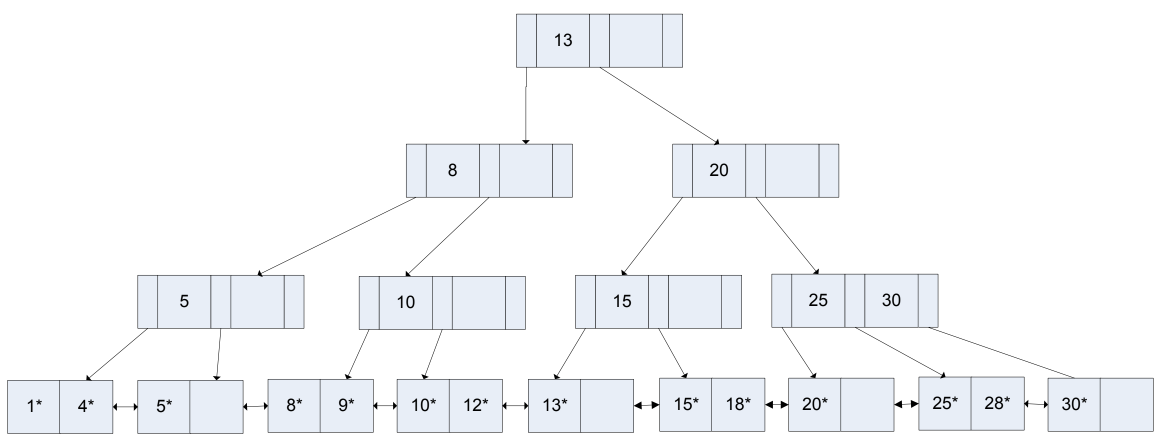
Our last case occurs when we want to add 15. This is going to result in the root node being split. The leaf node is full, as are the two index nodes above it. This gives us:



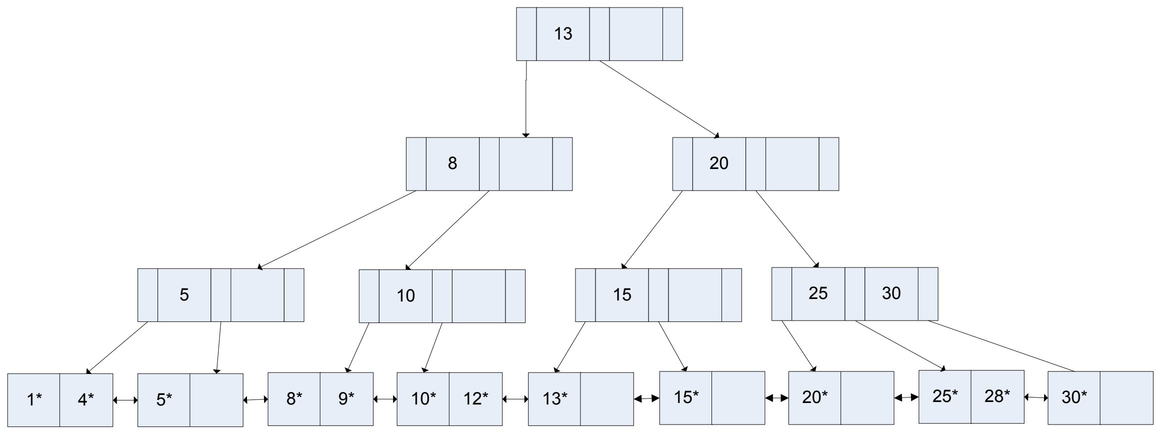
The delete algorithm:

No	No	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
Yes	No	Combine the leaf page and its sibling. Change the index page to reflect the change.
Yes	Yes	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. Continue combining index pages until you reach a page with thecorrect fill factor or you reach the root page.

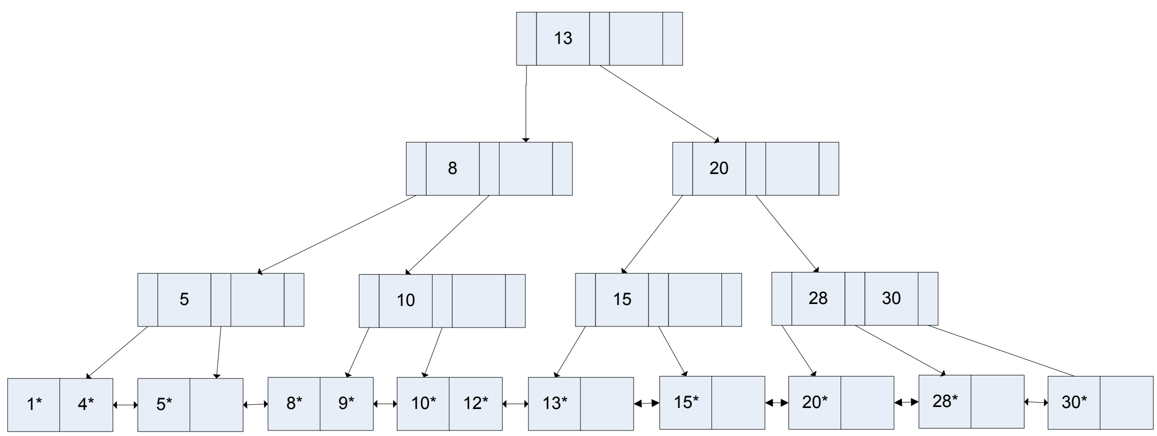
Let’s take our tree from the insert example with a minor modification (we have added 30 to give us an index node with 2 indexes in it:



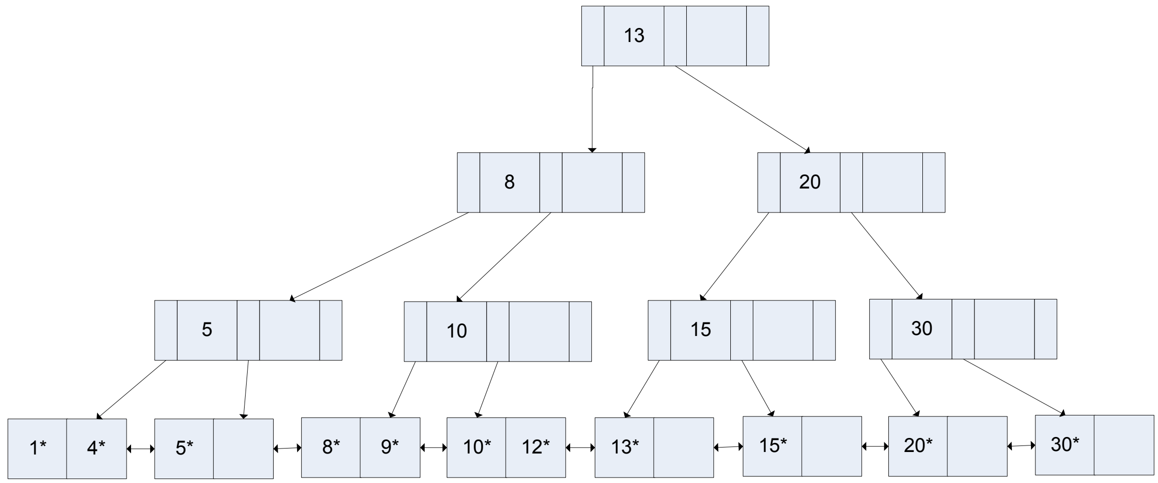
Our first delete is of 18. Simplest case is that it is not an index and in a leaf node that deleting it will not take you below d.



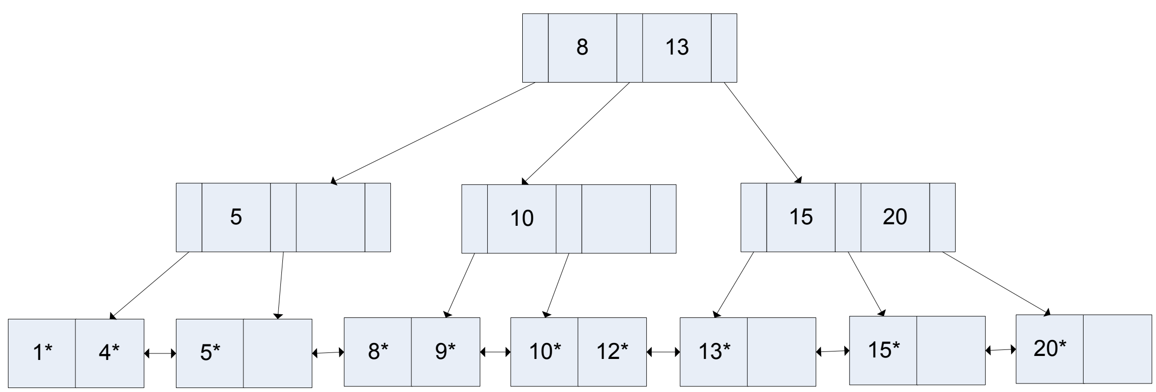
Our next delete is similar, except the index appears in a index node. In that case, the next index replaces the one in the index node. Let’s delete 25.



Our next case takes the node below d. Let's delete 28. For this one we combine the leaf page (in our case it is empty) with its sibling and update the index appropriately. That gives us:

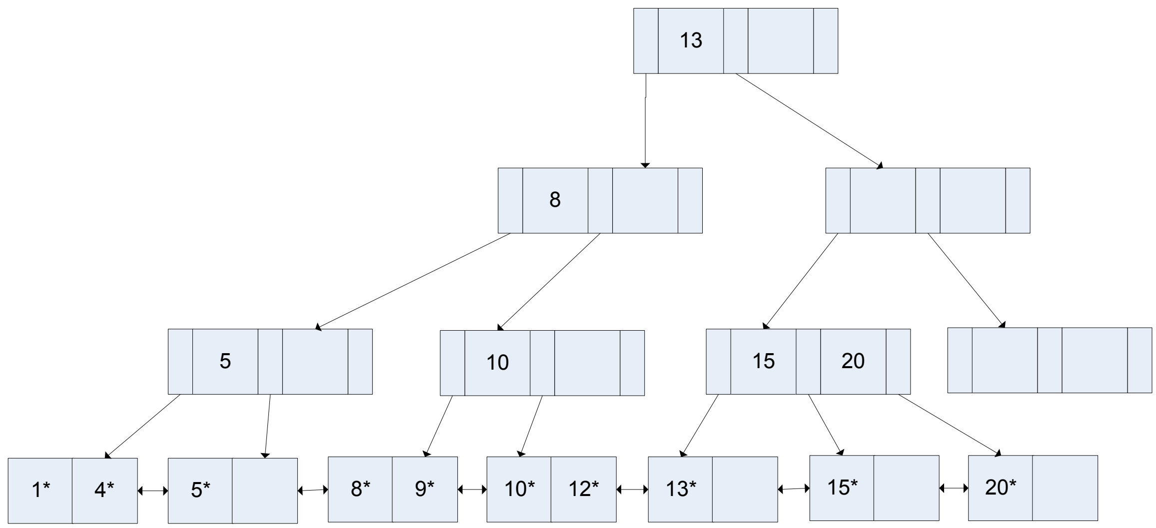


Next we delete 30. This takes us below d for the index. We combine the indexes, which has the effect of taking the index above below d. This continues to the root.

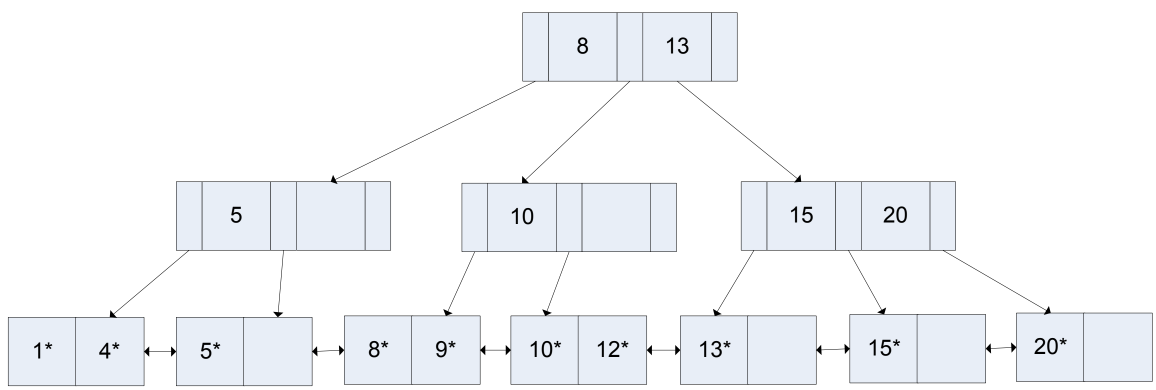


Woah. That seemed like magic. What process got us to that?

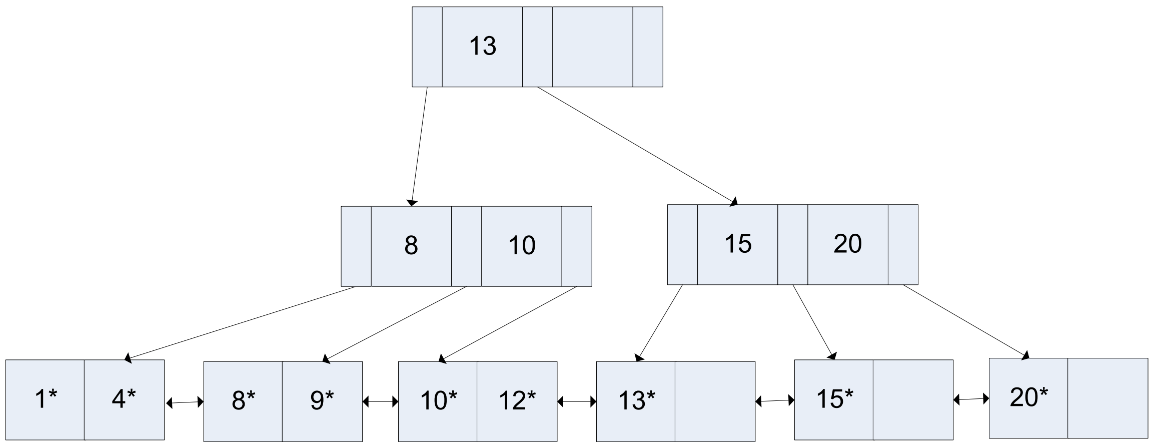
Ok – let's go through it. When we deleted 30, that took the data entry node that 30 was in below d. Now we have to merge with the sibling. When we merge – it's to the sibling on the left, which means pointer in the index above is no longer valid. We remove it, (which leaves it less than d), pull down the index from above and merge the index node with it's sibling.



Repeating the process gets us back to



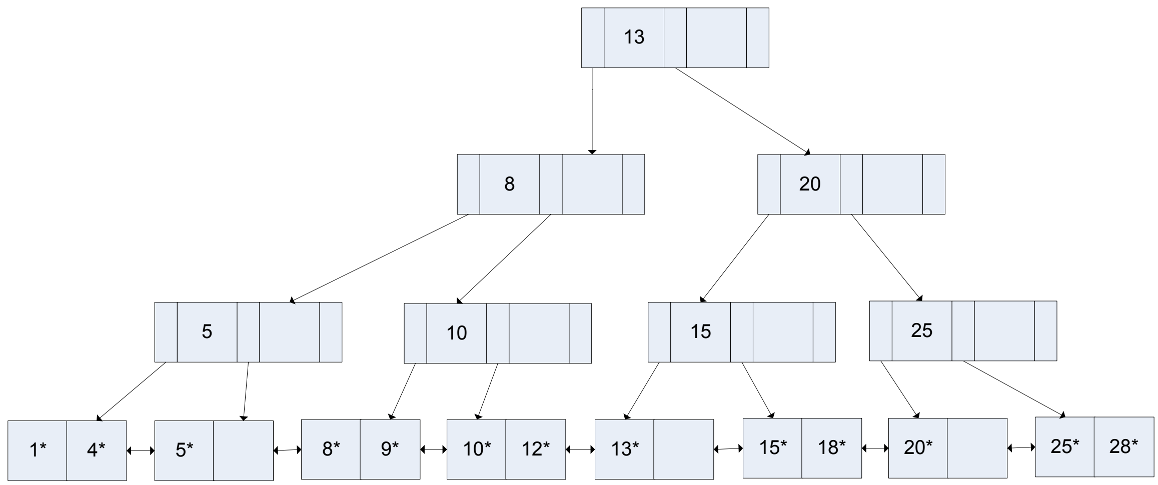
Our last example deletes 5. This takes the node and the index above it below d. We remove the leaf node and combine the index with its neighbor.



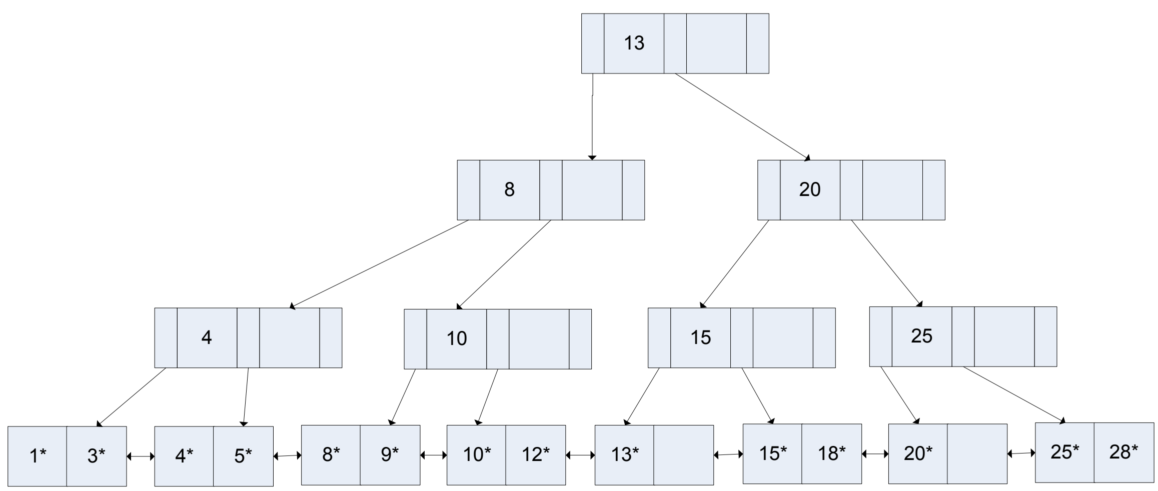
In this case, deleting 5 caused a merge with the data entry node containing (8,9). Eliminating the index node with 5 forced a merge with its sibling and pulled 8 down out from the parent node.

Rotation

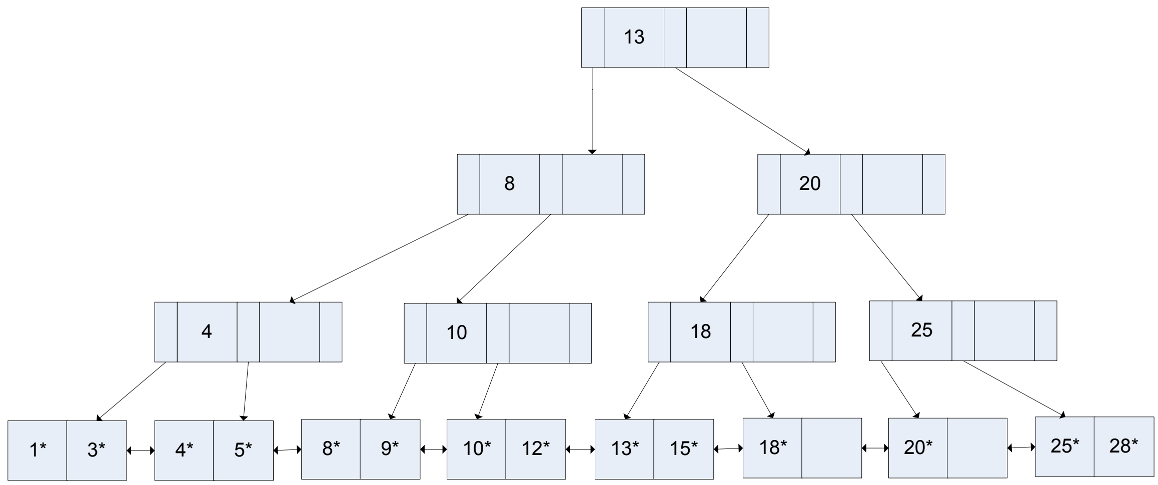
It is also possible to rebalance a tree to reduce the number of splits – called rotation. If you are trying to insert, and a leaf page is full, but its sibling isn't – you can move an index to a sibling and avoid splitting. Let's go back to a tree from our insert example:



We want to add 3 – but in this case we check the sibling to see if it has room. It does, so we move a record to it adjusting the index. Now we have :



The same concept works with deletes. If we took the above tree and deleted 13, you can re-distribute from the sibling:



and then do the delete: