mood-book

UNIT-5
STORAGE AND INDEXING

1. Data on External Storage

Disks:
Can retrieve random page at fixed cost
But reading several consecutive pages is much cheaper than
reading them in random order

Tapes:
Can only read pages in sequence
Cheaper than disks; used for archival storage

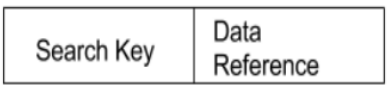
Indexing:

Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed.

An index or database index is a data structure which is used to quickly locate and access the data in a database table.

Indexes are created using some database columns.

- The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).
- The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.



Structure of an index

*Alternatives for Data Entry **k*** in Index:*

- In a data entry k* we can store:
  - Data record with key value **k,** or
  - <**k**, rid of data record with search key value **k**>, or
  - <**k**, list of rids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries
- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).

- At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
- If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

❖ Alternatives 2 and 3:

- Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
- Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

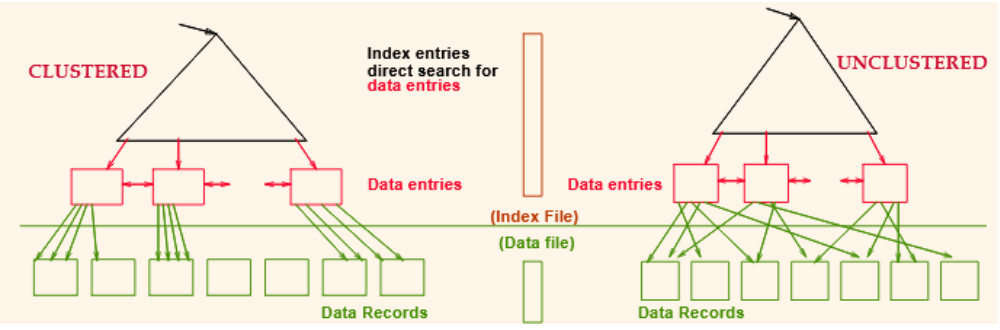
2. Types of Indexing:

- **Primary Index** − Primary index is defined on an ordered data file. The data file is ordered on a **key field**. The key field is generally the primary key of the relation.
- **Secondary Index** − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- **Clustering Index** − Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. (or)

  If order of data records is the same as, or `close to', order of data entries, then called clustered index. Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare). A file can be clustered on at most one search key. Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

  Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.

  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



Ordered Indexing is of two types −

- Dense Index
- Sparse Index

Dense Index

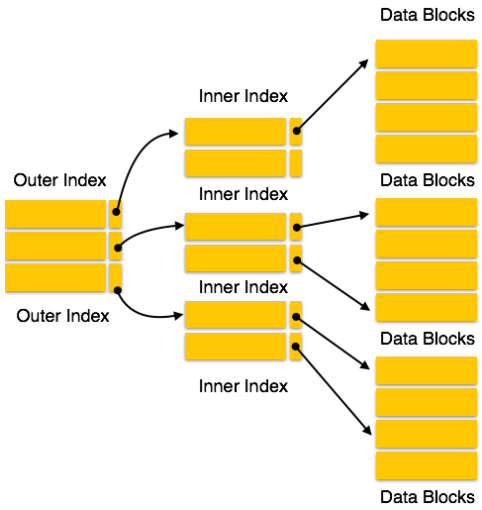
In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.

Sparse Index

In sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.

Multilevel Index

Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.



Indexes must be chosen to speed up important queries (and perhaps some updates!).

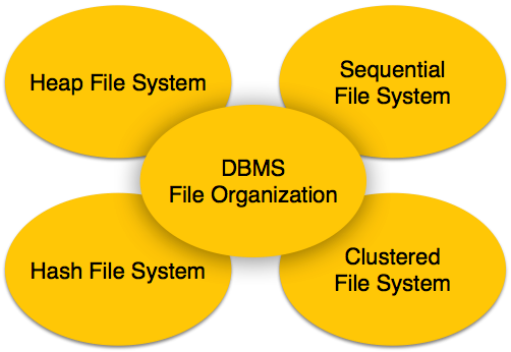
- Index maintenance overhead on updates to key fields.
- Choose indexes that can help many queries, if possible.
- Build indexes to support index-only strategies.
- Clustering is an important decision; only one index on a given relation can be clustered!
- Order of fields in composite index key can be important.

3. File organization and Indexing:

Relative data and information are stored collectively in file formats. A file is a sequence of records stored in binary format. A disk drive is formatted into several blocks that can store records. File records are mapped onto those disk blocks.

File Organization

File Organization defines how file records are mapped onto disk blocks. We have four types of File Organization to organize file records −



Heap File Organization

When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of the software to manage the records. Heap File does not support any ordering, sequencing, or indexing on its own.

Sequential File Organization

Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization, records are placed in the file in some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.

Hash File Organization

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

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.

File Operations

Operations on database files can be broadly classified into two categories −

- **Update Operations**
- **Retrieval Operations**

Update operations change the data values by insertion, deletion, or update. Retrieval operations, on the other hand, do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays a significant role. Other than creation and deletion of a file, there could be several operations, which can be done on files.

- **Open** − A file can be opened in one of the two modes, **read mode** or **write mode**. In read mode, the operating system does not allow anyone to alter data. In other words, data is read only. Files opened in read mode can be shared among several entities. Write mode allows data modification. Files opened in write mode can be read but cannot be shared.
- **Locate** − Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find (seek) operation, it can be moved forward or backward.
- **Read** − By default, when files are opened in read mode, the file pointer points to the beginning of the file. There are options where the user can tell the operating system where to locate the file pointer at the time of opening a file. The very next data to the file pointer is read.
- **Write** − User can select to open a file in write mode, which enables them to edit its contents. It can be deletion, insertion, or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allows to do so.
- **Close** − This is the most important operation from the operating system's point of view. When a request to close a file is generated, the operating system
    - removes all the locks (if in shared mode),
    - saves the data (if altered) to the secondary storage media, and
    - releases all the buffers and file handlers associated with the file.

The organization of data inside a file plays a major role here. The process to locate the file pointer to a desired record inside a file various based on whether the records are arranged sequentially or clustered.

4. Index Data structures:

1. Hash Based Indexing
2. Tree based Indexing

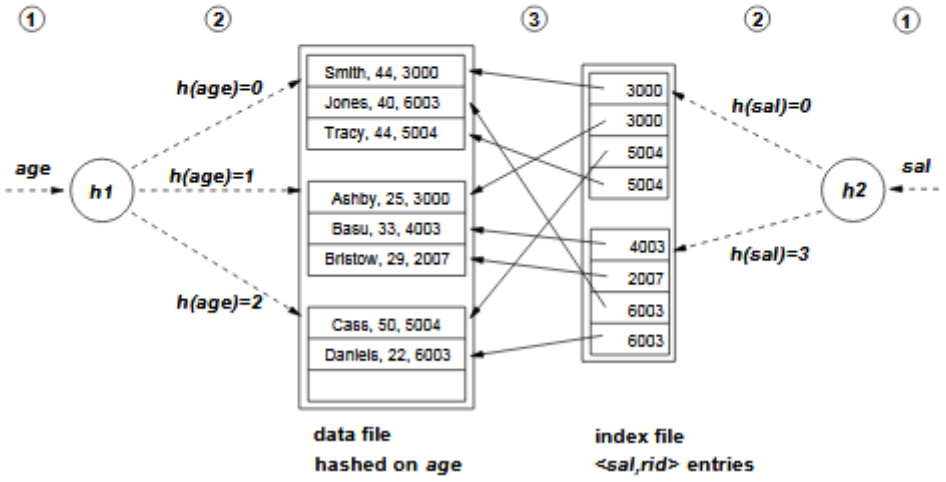
1. **hash based Indexing**: For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

Hashing uses hash functions with search keys as parameters to generate the address of a data record.

- **Bucket** − A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** − A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

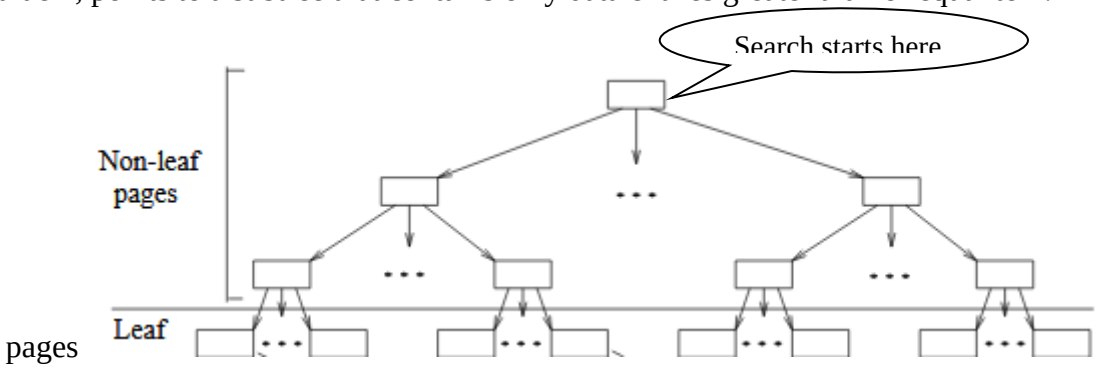
Ex:

- The data file contains ⟨*name*, *age*, *sal*⟩ records, the file itself (index entry variant ⓐ) is hashed on field *age* (hash function *h1*).
- The index file contains ⟨*sal*, *rid*⟩ index entries (variant ⓑ), pointing into the data file.
- This file organization + index efficiently supports equality searches on the *age* **and** *sal* keys.



2. **Tree Based Indexing:** An alternative to hash-based indexing is to organize records using a tree like data structure. The data entries are arranged in sorted order by search key value and a hierarchical search data structure is maintained that directs searches to the correct page of data entries.

The lowest level of the tree is called the leaf level, contains the data entries. This allows us to efficiently locate all data entries with search key values in a desired range. All searches begin at the top most node, called the root, and the contents of pages in non-leaf levels direct searches to the correct leaf page. Non leaf pages contain node pointers separated by search key values. The pointer to the left of a key value k, points to the subtree that contains only data entries less than k. The node pointer to the right of a key value k, points to a subtree that contains only data entries greater than or equal to k.



5. Comparison of File organizations:

	Sequential	**Heap/Direct**	**Hash**	**ISAM**	**B+ tree**	**Cluster**
Method of storing	Stored as they come or sorted as they come	Stored at the end of the file. But the address in the memory is random.	Stored at the hash address generated	Address index is appended to the record	Stored in a tree like structure	Frequently joined tables are clubbed into one file based on cluster key
Types	Pile file and sorted file Method		Static and dynamic hashing	Dense, Sparse, multilevel indexing		Indexed and Hash
Design	Simple Design	Simplest	Medium	Complex	Complex	Simple

Storage Cost	Cheap (magnetic tapes)	Cheap	Medium	Costlier	Costlier	Medium
Advantage	Fast and efficient when there is large volumes of data, Report generation, statistical calculations etc	Best suited for bulk insertion, and small files/tables	Faster Access No Need to Sort Handles multiple transactions Suitable for Online transactions	Searching records is faster. Suitable for large database. Any of the columns can be used as key column. Searching range of data & partial data are efficient.	Searching range of data & partial data are efficient. No performance degrades when there is insert / delete / update. Grows and shrinks with data. Works well in secondary storage devices and hence reducing disk I/O. Since all data is at the leaf node, searching is easy. All data at leaf node are sorted sequential linked list.	Best suited for frequently joined tables. Suitable for 1:M mappings
Disadvantage	Sorting of data each time for insert/delete/ update takes time and makes	Records are scattered in the memory and they are inefficiently used. Hence increases the memory	Accidental Deletion or updation of Data Use of Memory is inefficient Searching	Extra cost to maintain index. File reconstruction is needed as insert/update/delete. Does not grow with data.	Not suitable for static tables	Not suitable for large database. Suitable only for the joins on which

	system slow.	size. Proper memory management is needed. Not suitable for large tables.	range of data, partial data, non-hash key column, searching single hash column when multiple hash keys present or frequently updated column as hash key are inefficient.			clustering is done. Less frequently used joins and 1: 1 Mapping are inefficient.

Cost of Comparison of file organizations:

Operations to Compare Scan:

- Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record

We ignore CPU costs, for simplicity: B: The number of data pages

♣ R: Number of records per page

♣ D: (Average) time to read or write disk page

♣ Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated. Average-case analysis; based on several simplistic assumptions.

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	D(log 2 B) +D. # pgs w. match recs	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	D(log F 1.5B) + D. # pgs w. match recs	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log F 0.15B)	D(log F 0.15B + # match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

6. Tree Structured Indexing:

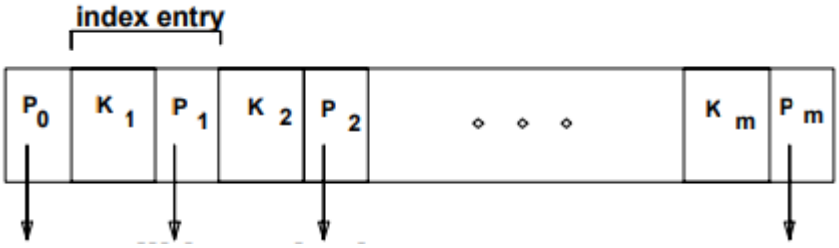
There are two index data structures, called ISAM and B+ trees, based on tree organizations. These structures provide efficient support for range searches, including sorted le scans as a special case. Unlike sorted les, these index structures support efficient insertion and deletion. They also provide support for equality selections, although they are not as efficient in this case as hash-based indexes.
tree-structured indexing techniques support both range searches and equality searches
ISAM: static structure;
B+ tree: dynamic, adjusts gracefully under inserts and deletes.

7. ISAM (Indexed Sequential access Method):

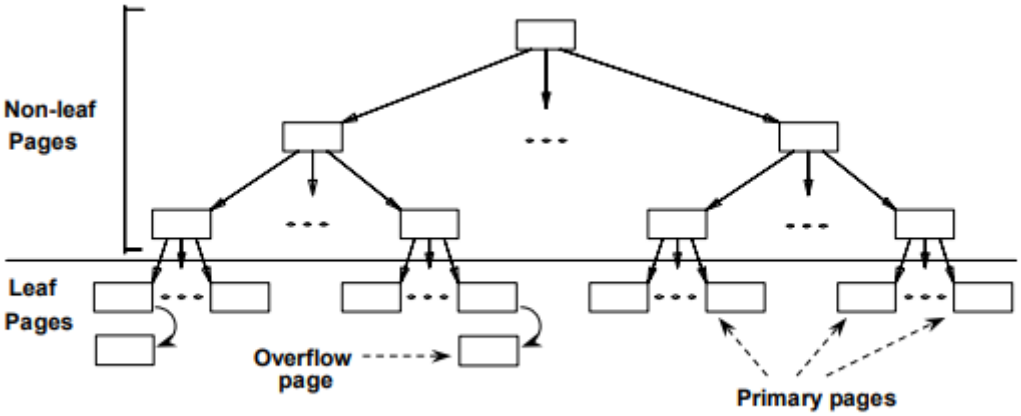
In an ISAM system, data is organized into records which are composed of fixed length fields. Records are stored sequentially, originally to speed access on a tape system. A secondary set of hash tables known as *indexes* contain "pointers" into the tables, allowing individual records to be retrieved without having to search the entire data set. This is a departure from the contemporaneous navigational databases, in which the pointers to other data were stored inside the records themselves. The key improvement in ISAM is that the indexes are small and can be searched quickly, thereby allowing the database to access only the records it needs.
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.
ISAM is very simple to understand and implement, as it primarily consists of direct, sequential access to a database file. It is also very inexpensive. The tradeoff is that each client machine must manage its own connection to each file it accesses. This, in turn, leads to the possibility of conflicting inserts into those files, leading to an inconsistent database state. This is typically

solved with the addition of a client-server framework which marshals client requests and maintains ordering. This is the basic concept behind a database management system (DBMS), which is a client layer over the underlying data store.
The indexed access method of reading or writing data only provides the desired outcome if in fact the file is organized as an ISAM file with the appropriate, previously defined keys. Access to data via the previously defined key(s) is extremely fast. Multiple keys, overlapping keys and key compression within the hash tables are supported.

Node structure of ISAM:

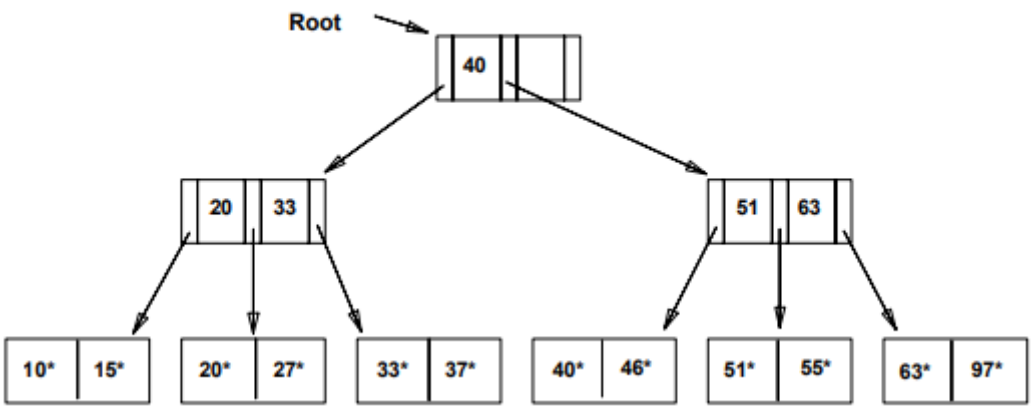


The tree can be represented as,

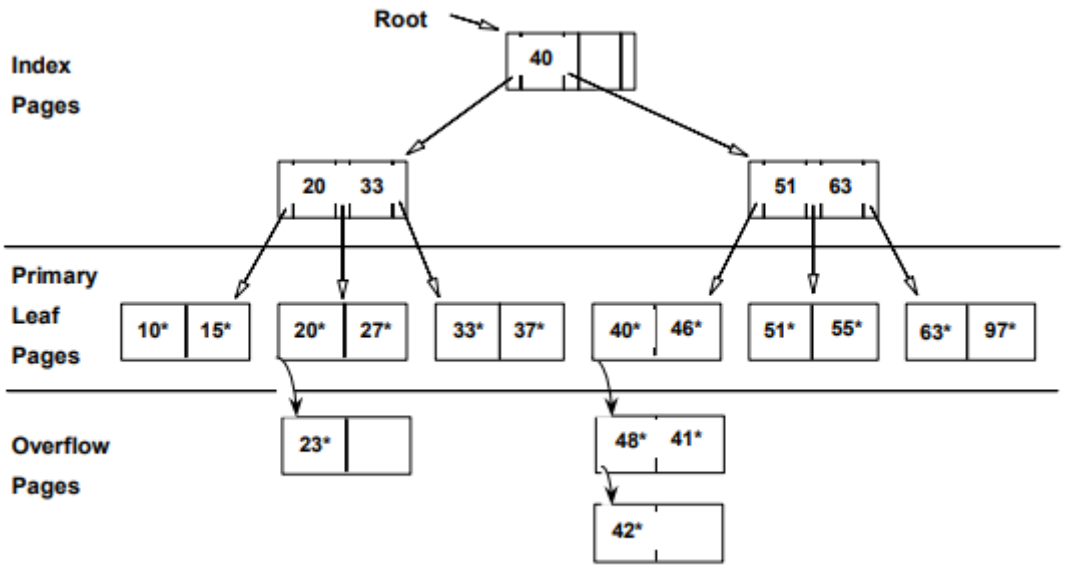


✉ *Leaf pages contain data entries.*

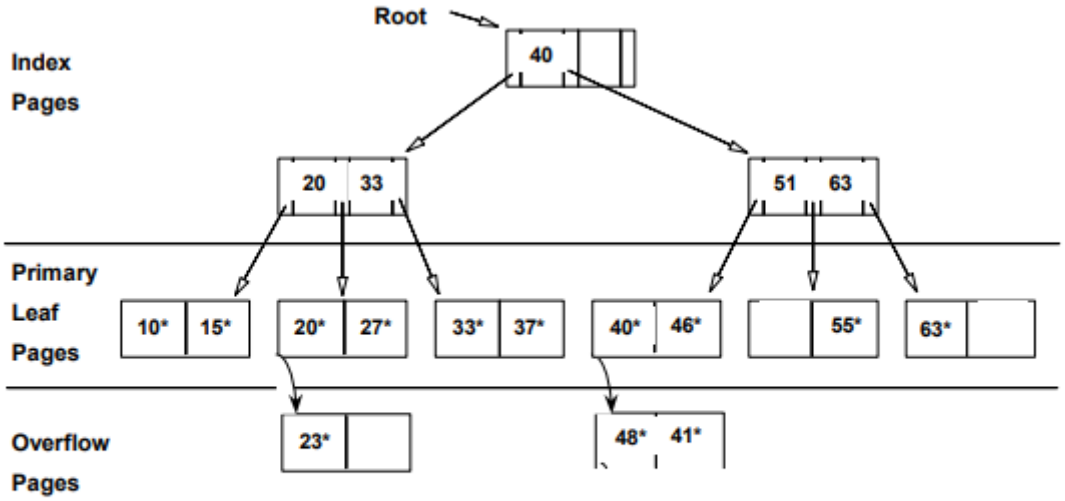
Example:
Index entries: <the search key value, block/page id>
they direct search for data entries in leaves.
Example where each node can hold 2 entries; 10* 15* 20* 27* 33* 37* 40* 46* 51* 55* 63* 97*
(* represents key entry)



After Inserting 23*,48*, 41*,42*,



Deleting 41*,52*,97*, the tree becomes...



✉ *Note that 51* appears in index levels, but not in leaf!*

8. B+ Tree:

B Trees. B Trees are multi-way trees. That is each node contains a set of keys and pointers. A B Tree with four keys and five pointers represents the minimum size of a B Tree node. A B Tree contains only data pages.

B Trees are dynamic. That is, the height of the tree grows and contracts as records are added and deleted.

B+ Trees A B+ Tree combines features of ISAM and B Trees. It contains index pages and data pages. The data pages always appear as leaf nodes in the tree. The root node and intermediate nodes are always index pages. These features are similar to ISAM. Unlike ISAM, overflow pages are not used in B+ trees.

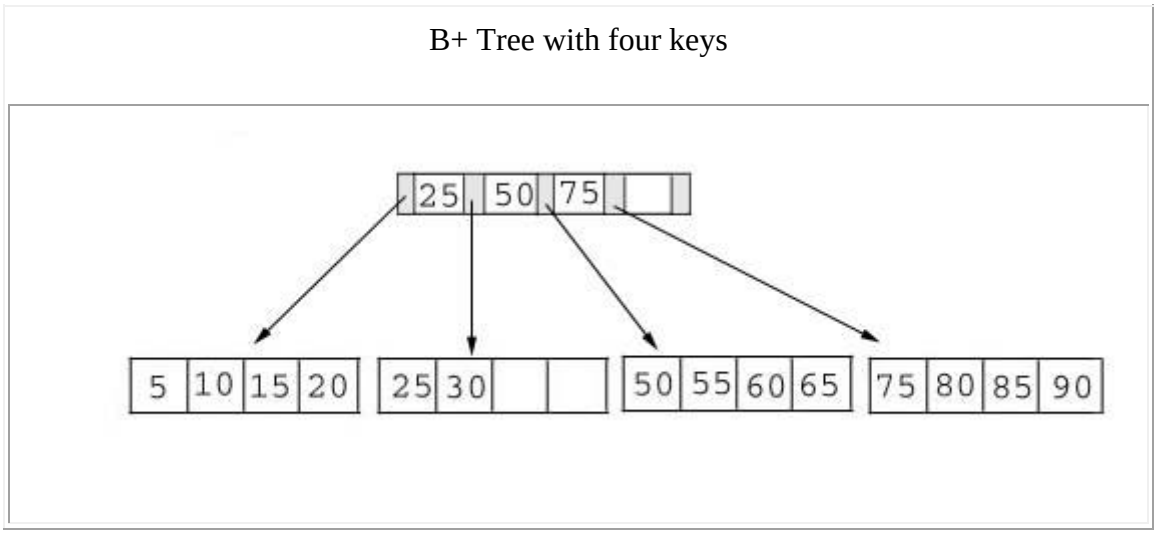
The index pages in a B+ tree are constructed through the process of inserting and deleting records. Thus, B+ trees grow and contract like their B Tree counterparts. The contents and the number of index pages reflects this growth and shrinkage.

B+ Trees and B Trees use a "fill factor" to control the growth and the shrinkage. A 50% fill factor would be the minimum for any B+ or B tree. As our example we use the smallest page structure. This means that our B+ tree conforms to the following guidelines.

Number of Keys/page	4
Number of Pointers/page	5
Fill Factor	50%
Minimum Keys in each page	2

As this table indicates each page must have a minimum of two keys. The root page may violate this rule. The following table shows a B+ tree. As the example illustrates this tree does not have a full index page. (We have room for one more key and pointer in the root page.) In addition, one of the data pages contains empty slots.

B+ Tree with four keys



Adding Records to a B+ Tree

The key value determines a record's placement in a B+ tree. The leaf pages are maintained in sequential order AND a doubly linked list (not shown) connects each leaf page with its sibling page(s). This doubly linked list speeds data movement as the pages grow and contract.

We must consider three scenarios when we add a record to a B+ tree. Each scenario causes a different action in the insert algorithm. The scenarios are:

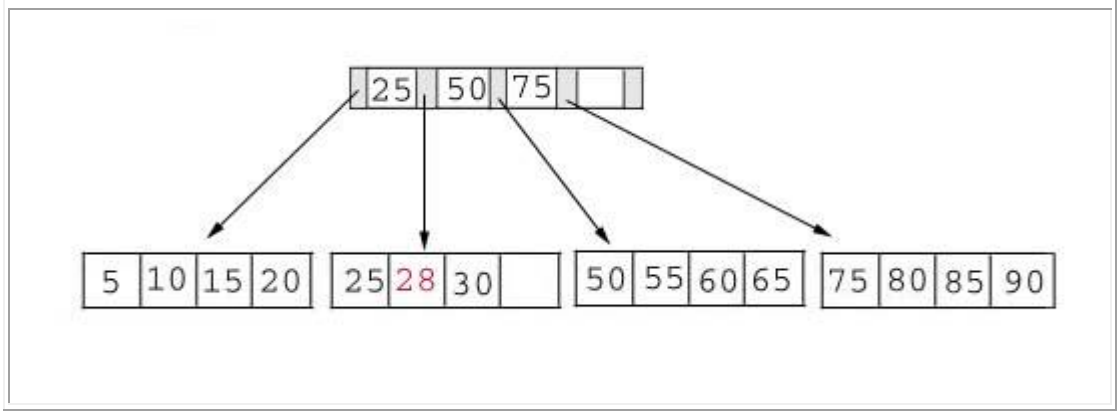
The insert algorithm for B+ Trees

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.

Illustrations of the insert algorithm

The following examples illustrate each of the **insert** scenarios. We begin with the simplest scenario: inserting a record into a leaf page that is not full. Since only the leaf node containing 25 and 30 contains expansion room, we're going to insert a record with a key value of 28 into the B+ tree. The following figures shows the result of this addition.

Add Record with Key 28



Adding a record when the leaf page is full but the index page is not

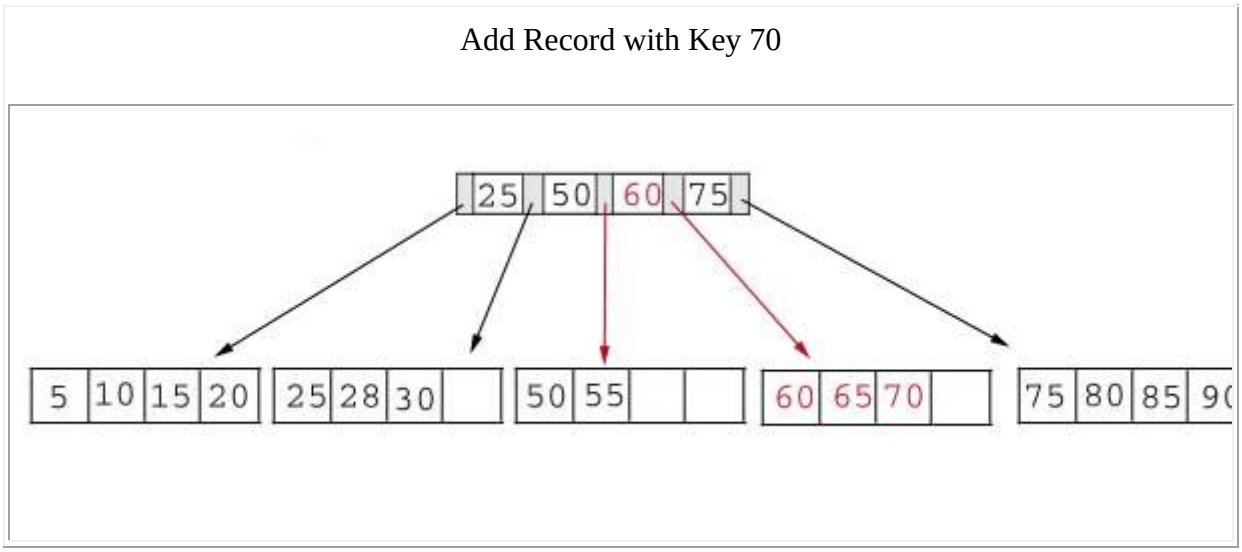
Next, we're going to insert a record with a key value of 70 into our B+ tree. This record should go in the leaf page containing 50, 55, 60, and 65. Unfortunately this page is full. This means that we must split the page as follows:

Left Leaf Page	Right Leaf Page
50 55	60 65 70

The middle key of 60 is placed in the index page between 50 and 75.

The following table shows the B+ tree after the addition of 70.

Add Record with Key 70



Adding a record when both the leaf page and the index page are full

As our last example, we're going to add a record containing a key value of 95 to our B+ tree. This record belongs in the page containing 75, 80, 85, and 90. Since this page is full we split it into two pages:

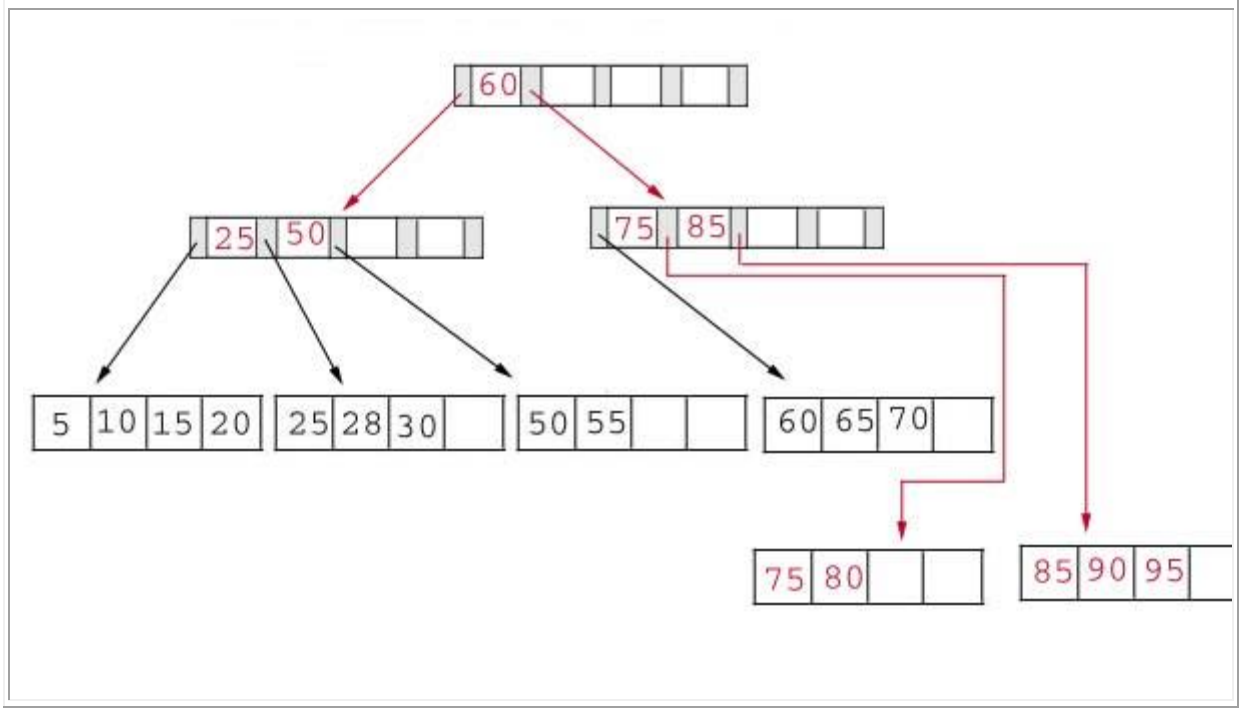
Left Leaf Page	Right Leaf Page
75 80	85 90 95

The middle key, 85, rises to the index page. Unfortunately, the index page is also full, so we split the index page:

Left Index Page	Right Index Page	New Index Page
25 50	75 85	60

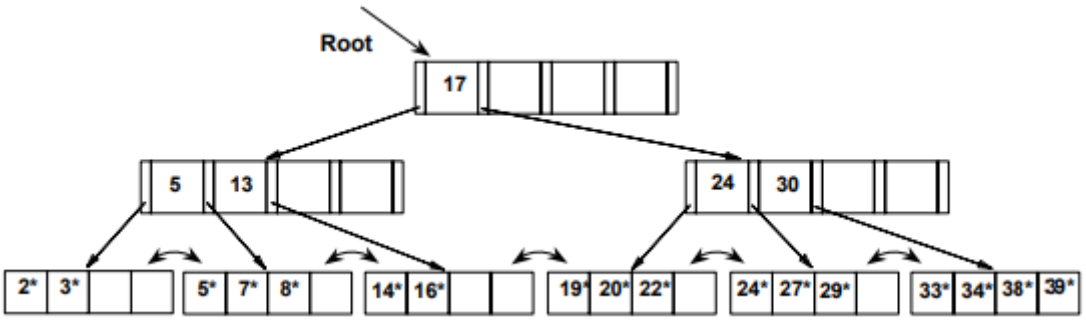
The following table illustrates the addition of the record containing 95 to the B+ tree.

Add Record with Key 95

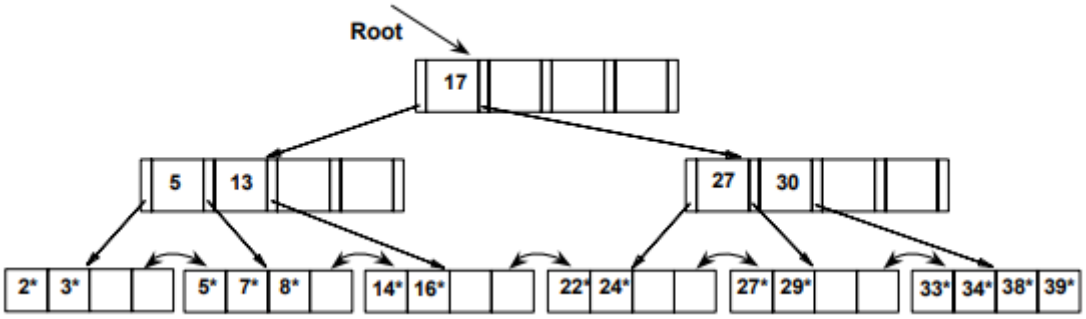


Deletion in B+ Tree:

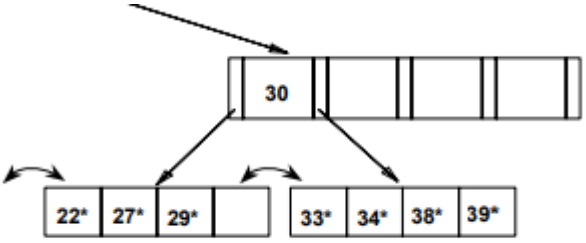
Consider the tree,



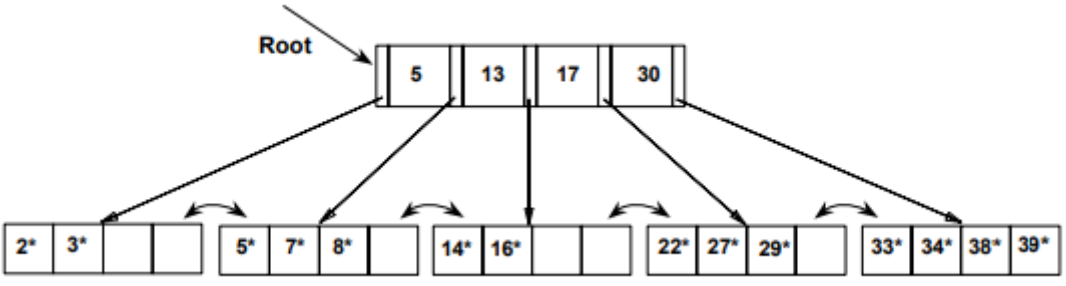
Deleting 19*,20*



Deleting 24*,



Tree appears like,



Deleting Keys from a B+ tree

We must consider three scenarios when we delete a record from a B+ tree. Each scenario causes a different action in the delete algorithm. The scenarios are:

The delete algorithm for B+ Trees

Leaf Page Below Fill Factor	Index Page Below Fill Factor	Action
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 either by deleting the parent or merging the parent level indexes.
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 the correct fill factor or you reach the root page.

9. Hash Based Indexing:

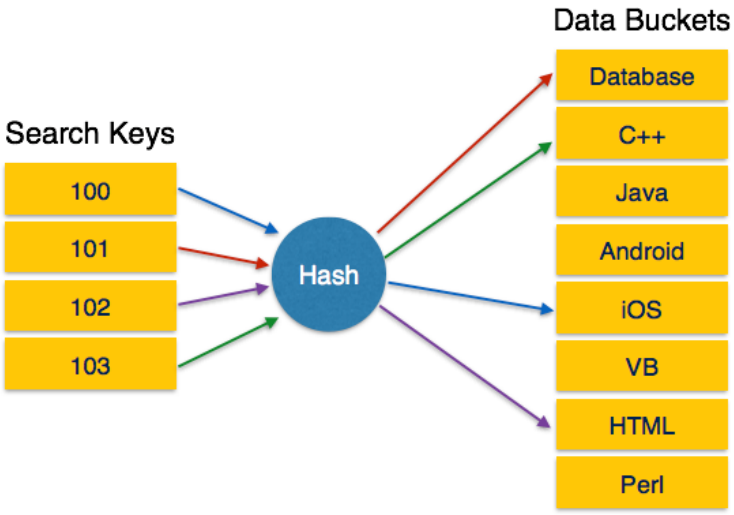
Hash Organization

- **Bucket** − A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** − A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values.

The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.



Operation

- **Insertion** − When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

  Bucket address = h(K)

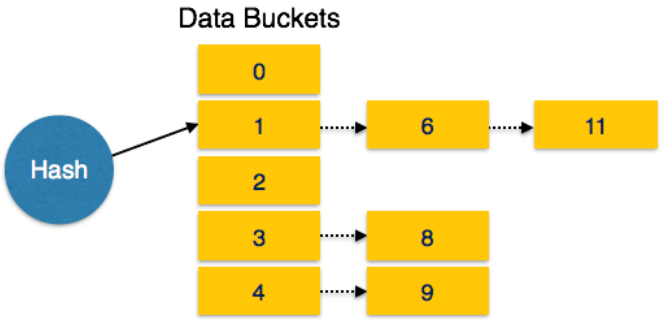
- **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

- **Delete** − This is simply a search followed by a deletion operation.

Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **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 called **Closed Hashing**.



Disadvantages of Static hashing:

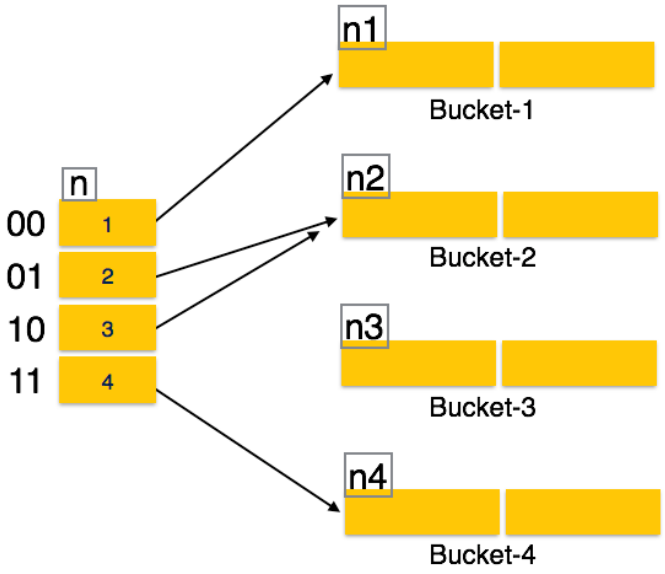
- In static hashing, function h maps searchkey values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
- If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
- If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
- If database shrinks, again space will be wasted.
  One solution: periodic reorganization of the file with a new hash function
  Expensive, disrupts normal operations

- Better solution: allow the number of buckets to be modified dynamically.

Extendible hashing/ Dynamic Hashing:

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. 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 − that is, when all the buckets are full − then the depth value is increased linearly and twice the buckets are allocated.

Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

Mian points on Extendable hashing – one form of dynamic hashing

- Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$.
- At any time use only a prefix of the hash function to index into a table of bucket addresses.
- Let the length of the prefix be *i* bits, $0 \le i \le 32$.
  - Bucket address table size = $2^{i.}$ Initially $i = 0$
  - Value of *i* grows and shrinks as the size of the database grows and shrinks.
- Multiple entries in the bucket address table may point to a bucket.
- Thus, actual number of buckets is $< 2^i$
  - The number of buckets also changes dynamically due to merging and splitting of buckets.

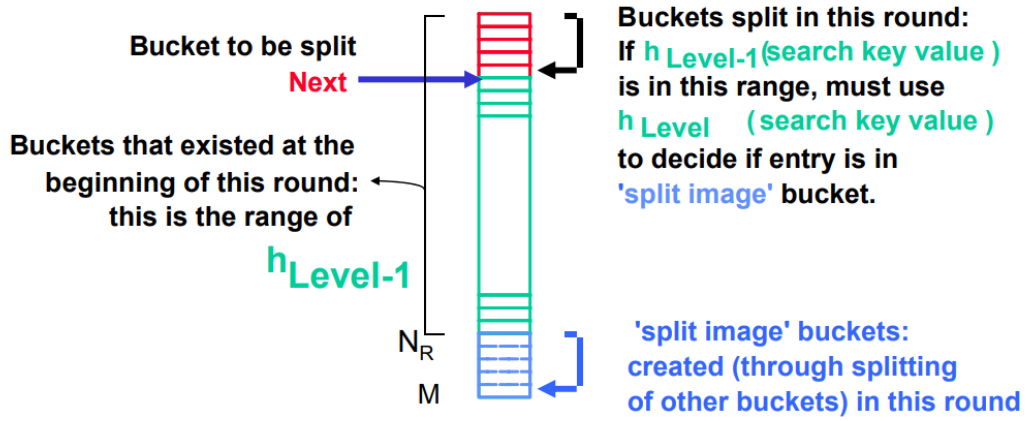
Linear Hashing:

This is another dynamic hashing scheme, an alternative to Extendible Hashing.

LH handles the problem of long overflow chains without using a directory, and handles duplicates.

Idea: Use a family of hash functions h0, h1, h2, ... – hi(key) = h(key) mod(2iN); N = initial # buckets – h is some hash function (range is 0 to 2|MachineBitLength|) – If N = 2d0, for some d0, hi consists of applying h and looking at the last di bits, where di = d0 + i. – hi+1 doubles the range of hi (similar to directory doubling)

Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin. – Splitting proceeds in 'rounds'. Round ends when all NR initial (for round R) buckets are split. – Buckets 0 to Next-1 have been split; Next to NR yet to be split. – Current round number is Level.



Ex: Initial Layout

The Linear Hashing scheme has m initial buckets labelled 0 through m−1, and an initial hashing function h0(k) = f(k) % m that is used to map any key k into one of the m buckets (for simplicity assume h0(k) = k % m), and a pointer p which points to the bucket to be split next whenever an overflow page is generated (initially p = 0). An example is shown in Figure 1. **Bucket Split**: When the first overflow occurs (it can occur in any bucket), bucket 0, which is pointed by p, is split (rehashed) into two buckets: the original bucket 0 and a new bucket m. A new empty page is also added in the overflown bucket to accommodate the overflow. The search values originally mapped into bucket 0

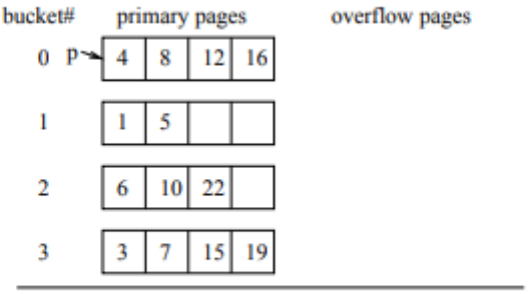


Figure 1: An initial Linear Hashing. Here $m = 4$, $p = 0$, $h_0(k) = k \ \% \ 4$.

function h0) are now distributed between buckets 0 and m using a new hashing function h1.

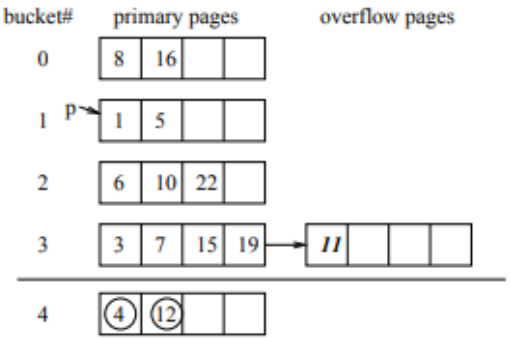


Figure 2: The Linear Hashing after inserting 11 into Figure 1. Here $p = 1$, $h_0(k) = k \ \% \ 4$, $h_1(k) = k \ \% \ 8$.

As an example, Figure 2 shows the layout of the Linear Hashing of Figure 1 after inserting a new record with key 11. The circled records are the existing records that are moved to the new bucket. In more detail, the record is inserted into bucket 11 % 4 = 3. The bucket overflows and an overflow page is introduced to accommodate the new record. Bucket 0 is split and the records originally in bucket 0 are distributed between bucket 0 and bucket 4, using a new hash function h1(k) = k % 8. The next bucket overflow, such as triggered by inserting two records in bucket 2 or four records in bucket 3 in Figure 2, will cause a new split that will attach a new bucket m+1 and the contents of bucket 1 will be distributed using h1 between buckets 1 and m + 1. A crucial property of h1 is that search values that were originally mapped by h0 to some bucket j must be remapped either to bucket j or bucket j + m. This is a necessary property for Linear Hashing to work. An example of such hashing function is: h1(k) = k % 2m. Further bucket overflows will cause additional bucket splits in a linear bucket-number order (increasing p by one for every split).