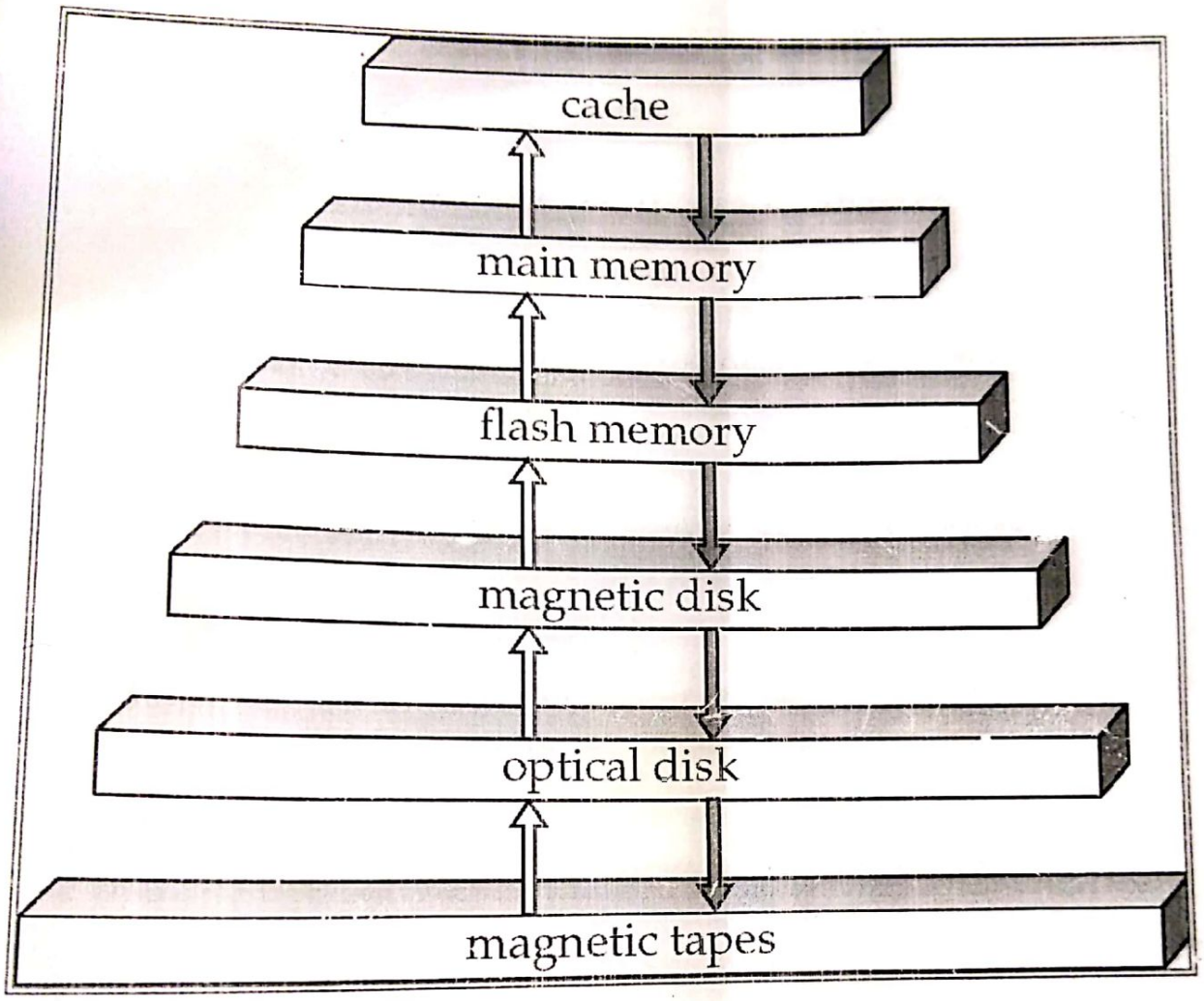
Classification of Physical Storage Media

Volatile storage: loses contents when power is switched off
Non-volatile storage: Contents persist even when power is switched off.



Cache – fastest and most costly form of storage; volatile; managed by the computer system hardware.

Main memory:

- Fast access (10s to 100s of nanoseconds)
- Generally too small (or too expensive) to store the entire database
- It is Volatile

Flash memory

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
- It supports only a limited number (10K – 1M) of write/erase cycles.
- Widely used in embedded devices such as digital cameras
- It is a type of EEPROM (Electrically Erasable Programmable Read-Only Memory)

Magnetic-disk

- Data is stored on disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
- Much slower access than main memory

Optical storage -Non-volatile, data is read optically from a disk using a laser- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms

- Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)- Reads and writes are slower than with magnetic disk
- **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic

loading/unloading of disks available for storing large volumes of data

Tape storage

- Non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB tapes available)

Improvement of Reliability via Redundancy

Redundancy – store extra information that can be used to rebuild information lost in a disk failure

E.g., **Mirroring** (or **shadowing**)

- Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
  - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
- Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired

Improvement in Performance via Parallelism

Two main goals of parallelism in a disk system:

1. Multiple accesses, increase throughput
2. Parallelize large accesses to reduce response time.

Improve transfer rate by striping data across multiple disks.

Bit-level striping – split the bits of each byte across multiple disks

- In an array of eight disks, write bit *i* of each byte to disk *i*.
- Each access can read data at eight times the rate of a single disk.

Block-level striping – with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1

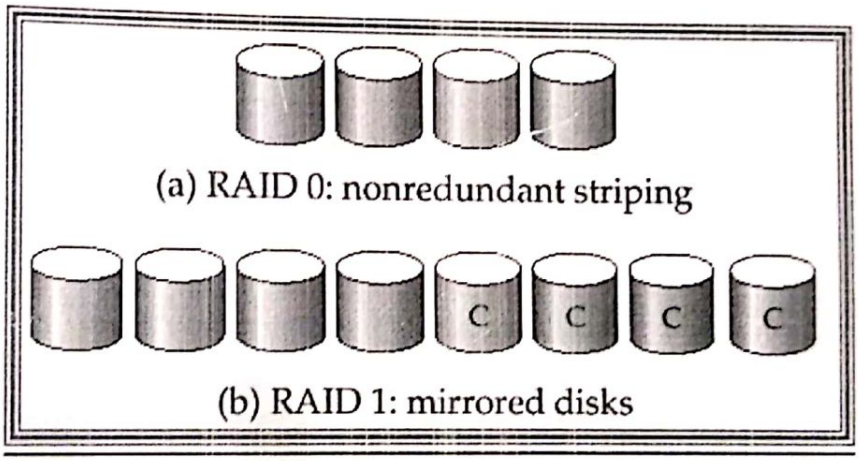
- Requests for different blocks can run in parallel if the blocks reside on different disks

Block – a contiguous sequence of sectors from a single track -sizes range from 512 bytes to several kilobytes

RAID(Redundant Arrays of Independent Disks)

RAID Level 0: Block striping; non-redundant.

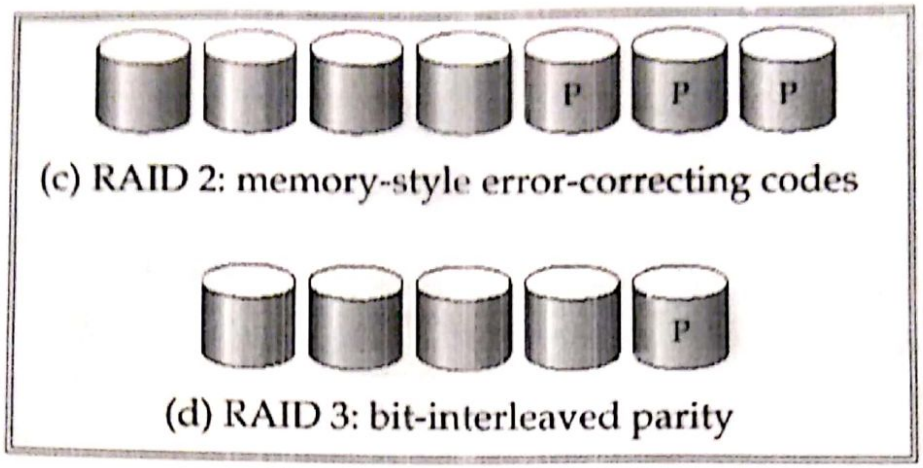
- Used in applications where data lose is not critical.



RAID Level 1: Mirrored disks with block striping

Popular for applications such as storing log files in a database system.

RAID Level 2: Memory-Style Error-Correcting-Codes (ECC) with bit striping.



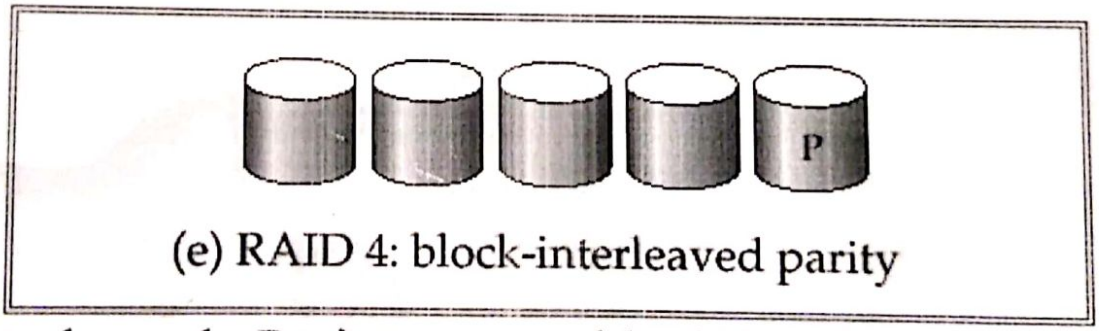
(c) RAID 2: memory-style error-correcting codes

(d) RAID 3: bit-interleaved parity

RAID Level 3: Bit-Interleaved Parity- a single parity bit is enough for error correction, since we know which disk has failed

- When writing data, corresponding parity bits must also be computed and written to a parity bit disk
- To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)

RAID Level 4:



(e) RAID 4: block-interleaved parity

Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from *N* other disks.

- When writing data block, corresponding block of parity bits must also be computed and written to parity disk

- To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.

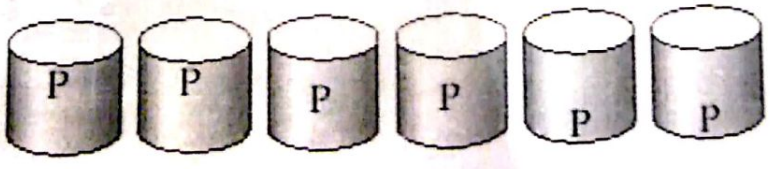
RAID Level 5: Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.

- E.g., with 5 disks, parity block for nth set of blocks is stored on disk (n *mod* 5) + 1, with the data blocks stored on the other 4 disks.

(f) RAID 5: block-interleaved distributed parity

0	1	2	3	4
P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

RAID Level 6: P+Q Redundancy scheme; similar to Level 5 but stores extra redundant information to guard against multiple disk failures.



(g) RAID 6: P + Q redundancy

Storage Access

A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.

Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.

Buffer – portion of main memory available to store copies of disk blocks.

Buffer manager – subsystem responsible for allocating buffer space in main memory.

Buffer Manager

Programs call on the buffer manager when they need a block from disk.

1. If the block is already in the buffer, buffer manager returns the address of the block in main memory
2. If the block is not in the buffer, the buffer manager
   a) Allocates space in the buffer for the block
      i) Replacing (throwing out) some other block, if required, to make space for the new block.
      ii) Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
   b) Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester

Buffer-Replacement Policies

1. Most operating systems replace the block **least recently used** (LRU strategy)
2. **Pinned block** – memory block that is not allowed to be written back to disk.
3. **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
4. Most recently used (MRU) strategy – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.

Buffer manager can use statistical information regarding the probability that a request will reference a particular relation

E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer

5. Buffer managers also support **forced output of blocks** for the purpose of recovery

File Organization

The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.

1. Fixed-Length Records

- Assume record size is fixed
- Each file has records of one particular type only
- Different files are used for different relations

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Insertion: Store record i starting from byte $n * (i - 1)$, where n is the size of each record.

Deletion of record i:

1. move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

With Record 2 Deleted and All Records Moved

2. Move record *n* to *i*

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600

With Record 2 deleted and Final Record Moved

3. Do not move records, but link all free records on a *free list*

header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

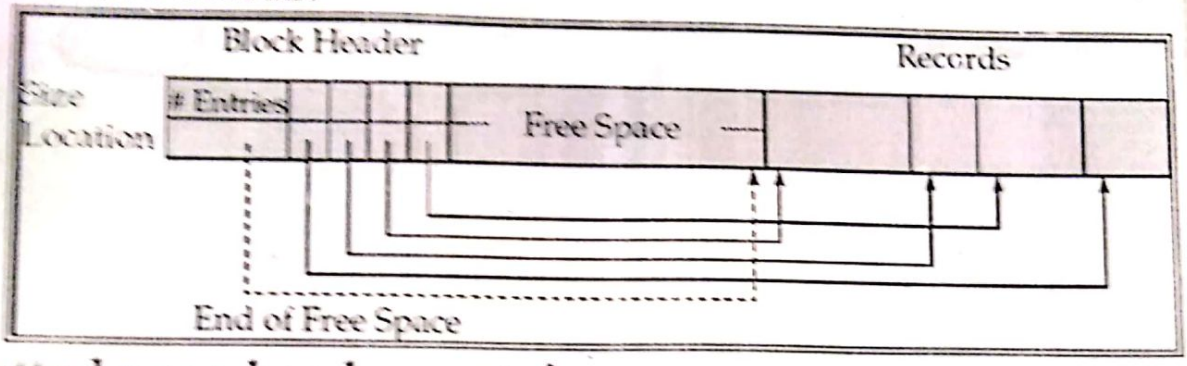
Store the address of the first deleted record in the file header. Use this first record to store the address of the second deleted record, and so on. Stored addresses as pointers since they "point" to the location of a record.

More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)

Variable-Length Records

Variable-length records arise in database systems in several ways:

Storage of multiple record types in a file.
Record types that allow variable lengths for one or more fields.



Slotted page header contains:

- number of record entries
- end of free space in the block
- location and size of each record

Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.

Pointers should not point directly to record — instead they should point to the entry for the record in header.

Organization of Records in Files

Records of each relation may be stored in a separate file.

Heap – a record can be placed anywhere in the file where there is space

Sequential – store records in sequential order, based on the value of the search key of each record

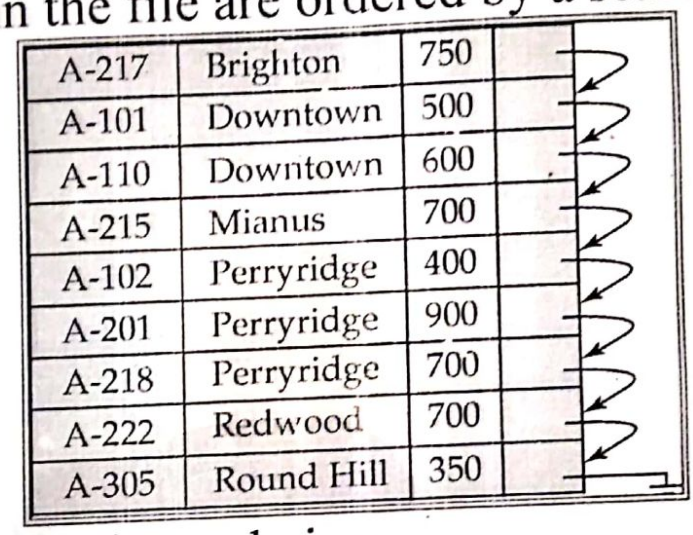
Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

In a **multi table clustering file organization** records of several different relations can be stored in the same file

Sequential File Organization

Suitable for applications that require sequential processing of the entire file

The records in the file are ordered by a search-key



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

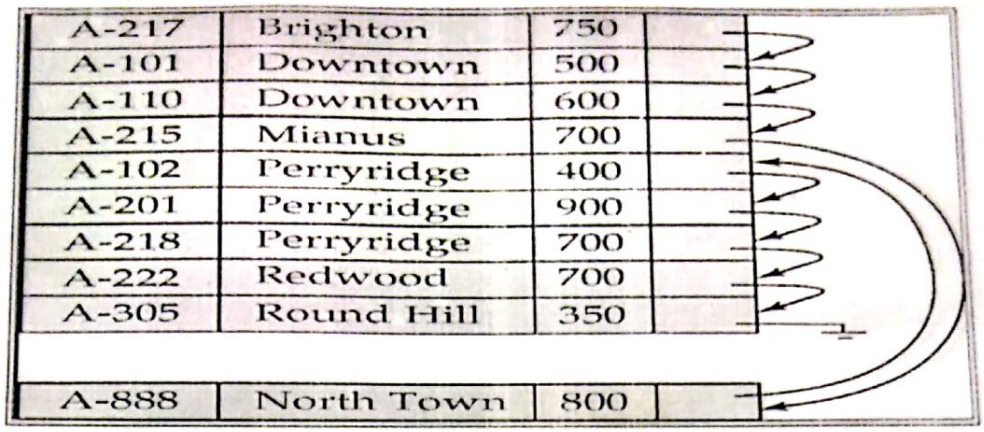
Deletion – use pointer chains

Insertion –locate the position where the record is to be inserted

1. if there is free space insert there
2. if no free space, insert the record in an overflow block
3. In either case, pointer chain must be updated

Need to reorganize the file from time to time to restore sequential order

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	
A-888	North Town	800	



Multi table Clustering File Organization

Store several relations in one file using a **multi table clustering** file organization

customer_name	*account_number*
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

Depositor

customer_name	*customer_street*	*customer_city*
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

Customer

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

Multi table clustering organization of customer and depositor

Data Dictionary Storage

Data dictionary (also called system catalog) stores metadata; that is, data about data, such as

1. Information about relations
   - names of relations
   - names and types of attributes of each relation
   - names and definitions of views
   - integrity constraints
2. User and accounting information, including passwords
3. Statistical and descriptive data
   - number of tuples in each relation
4. Physical file organization information
   - How relation is stored (sequential/hash/...)
   - Physical location of relation
5. Information about indices
6. Catalog structure
   - Relational representation on disk
   - Specialized data structures designed for efficient access, in memory

A possible catalog representation

Relation_metadata = (relation_name, number_of_attributes, storage_organization, location)

Attribute_metadata = (attribute_name, relation_name, domain_typc, position, length)

User_metadata = (user_name, encrypted_password, group)

Index_metadata = (index_name, relation_name, index_type, index_attributes)

View_metadata = (view_name, definition)

Indexing and Hashing

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

Search Key	Pointer

- Index files are typically much smaller than the original file

Two basic kinds of indices:

1. Ordered indices: search keys are stored in sorted order

2. Hash indices: search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

i) Access types supported efficiently. E.g.,
records with a specified value in the attribute [point Q
or records with an attribute value falling in a specified range of values. [range Query]

ii)Access time

iii) Insertion time

iv) Deletion time

v)Space overhead

Ordered Indices

In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.

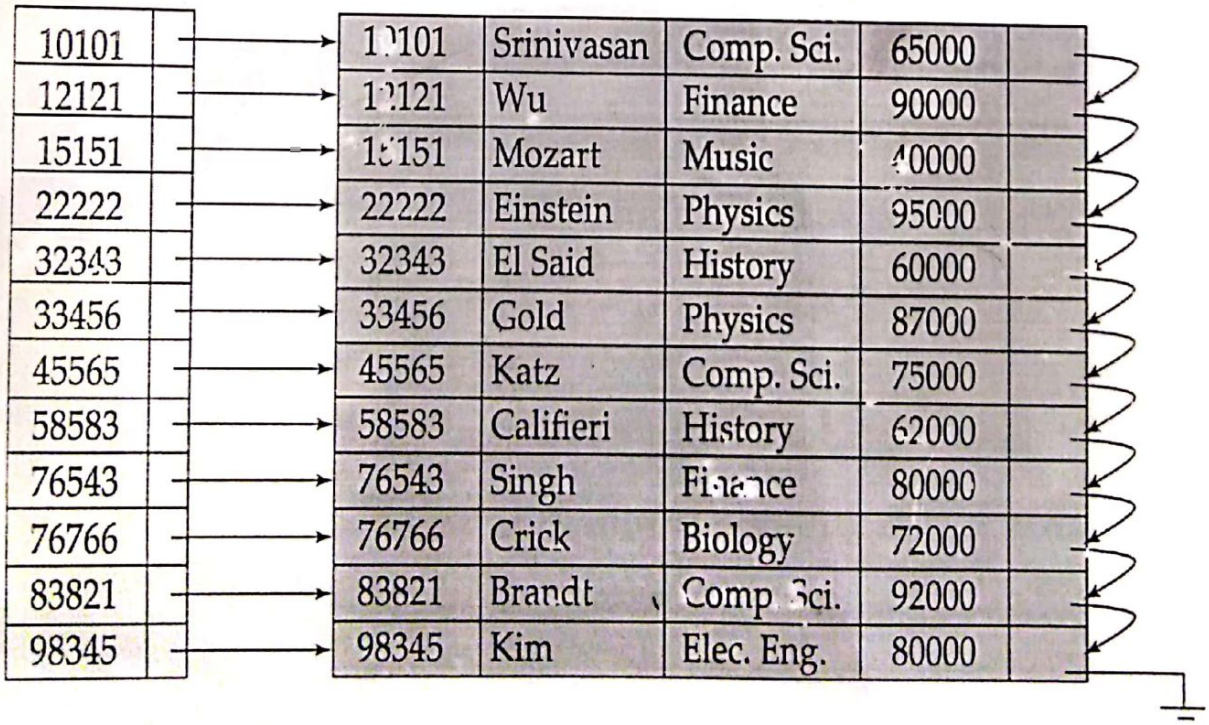
Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file. It also called as **clustering index**. The search key of a primary index is usually but not necessarily the primary key.

Secondary index: an index whose search key specifies an order different from the sequential order of the file. It also called as non-clustering index.

Index-sequential file: ordered sequential file with a primary index.

Dense Index Files

Dense index — Index record appears for every search-key value in the file. E.g. index on *ID* attribute of *instructor* relation

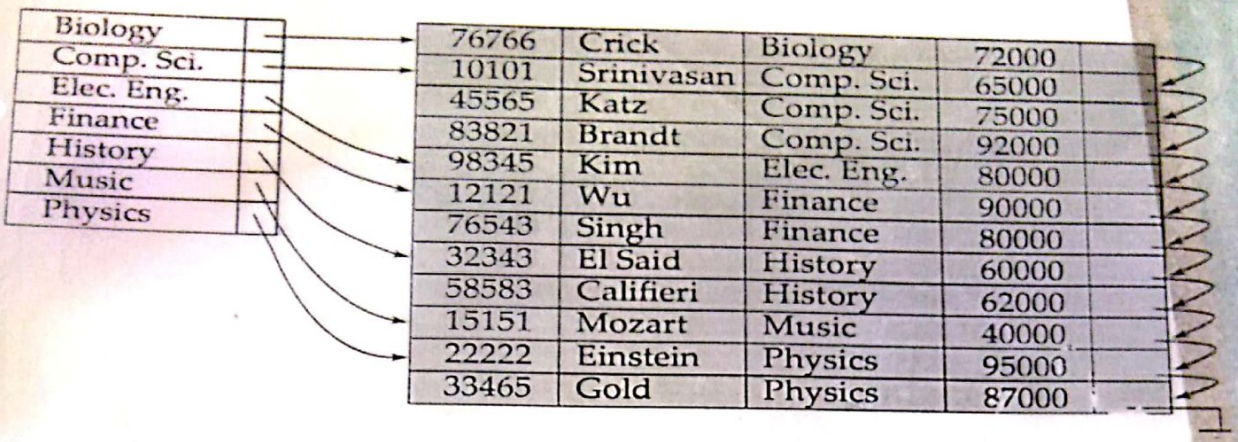


10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

B+ tree – Insert Adams, Lamport (n = 4)

Delete Srinivasan, Singh, Wu, Gold

Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

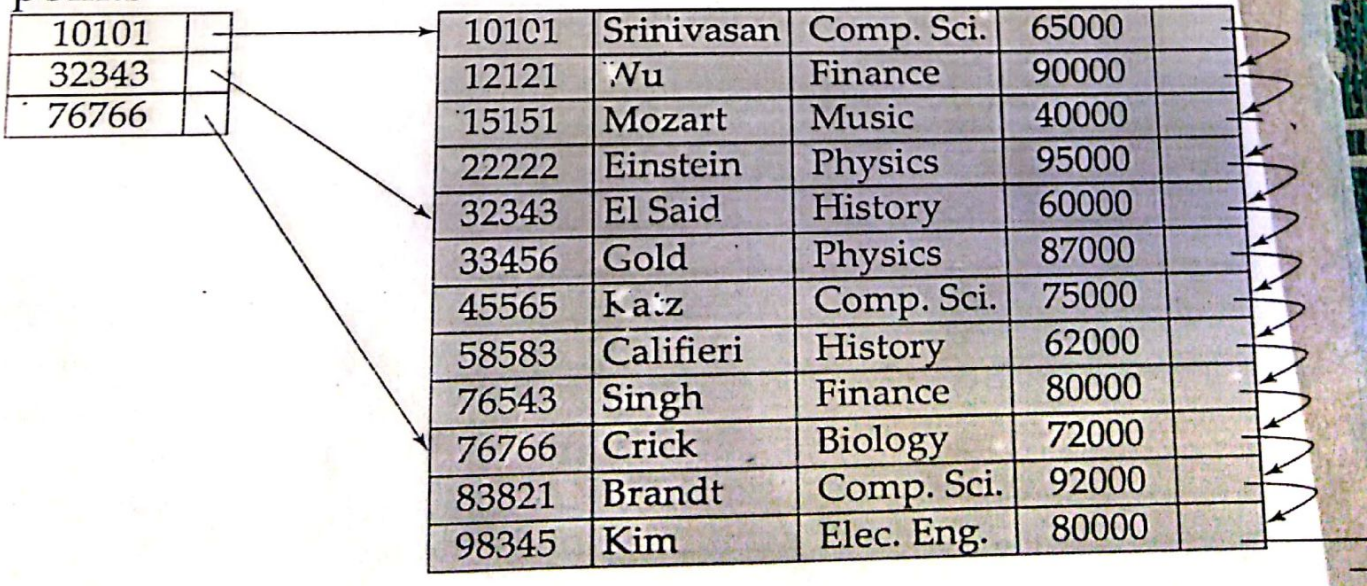
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

Sparse Index Files

Sparse Index: contains index records for only some search-key values. Applicable when records are sequentially ordered on search-key

To locate a record with search-key value K we:

- Find index record with largest search-key value $< K$
- Search file sequentially starting at the record to which the index record points

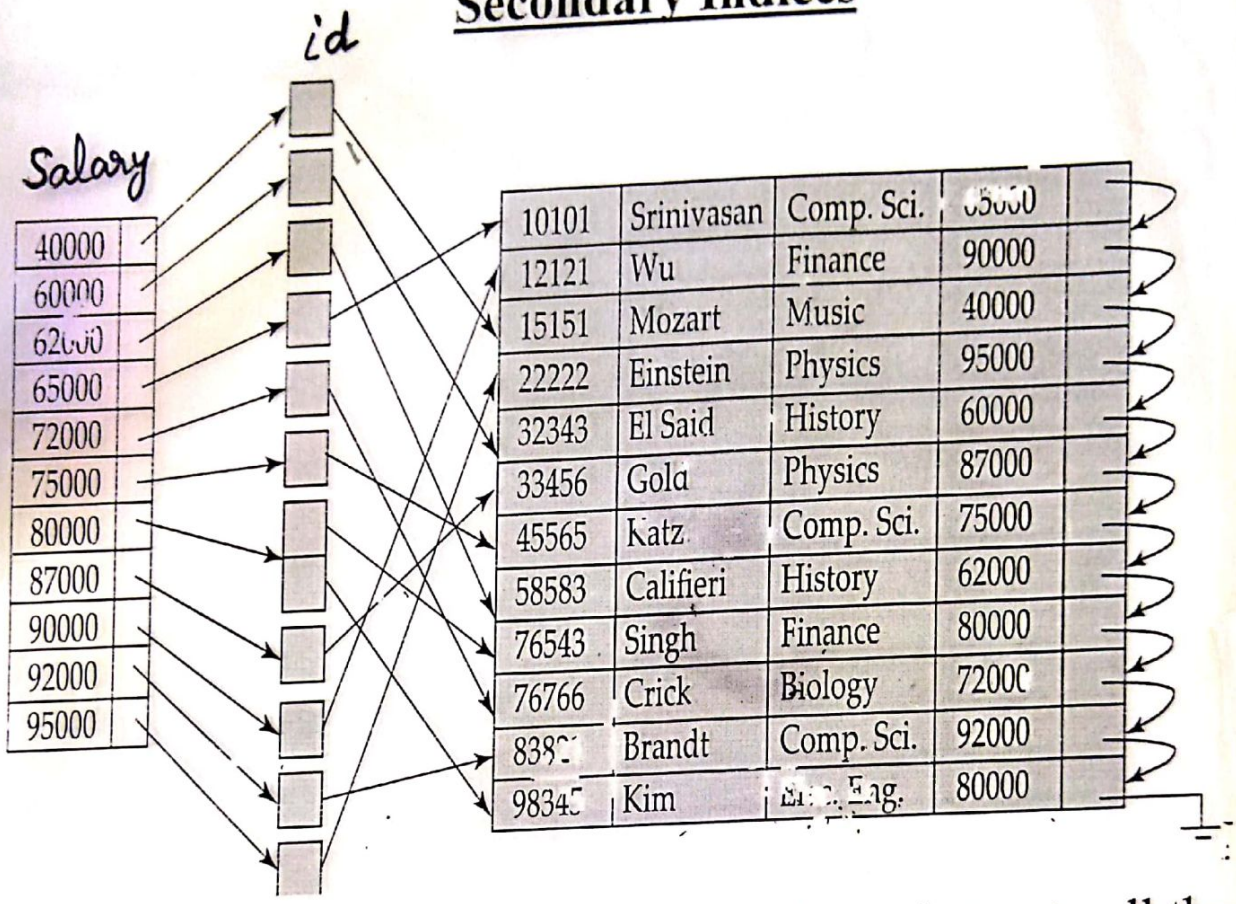


10101
32343
76766

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Compared to dense indices:
Less space and less maintenance overhead for insertions and deletions. Generally slower than dense index for locating records.

Secondary Indices



Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary indices have to be dense

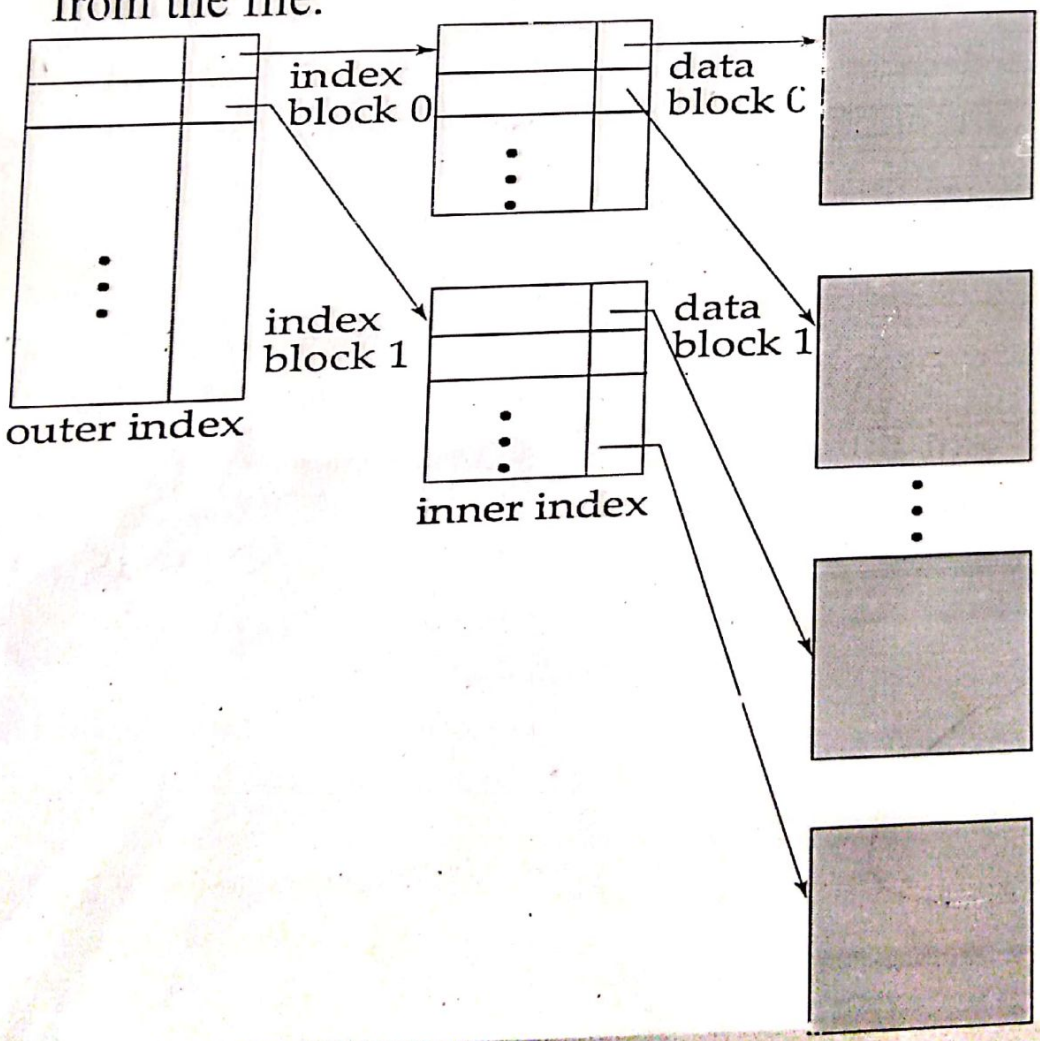
Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- But, updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated.

- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive

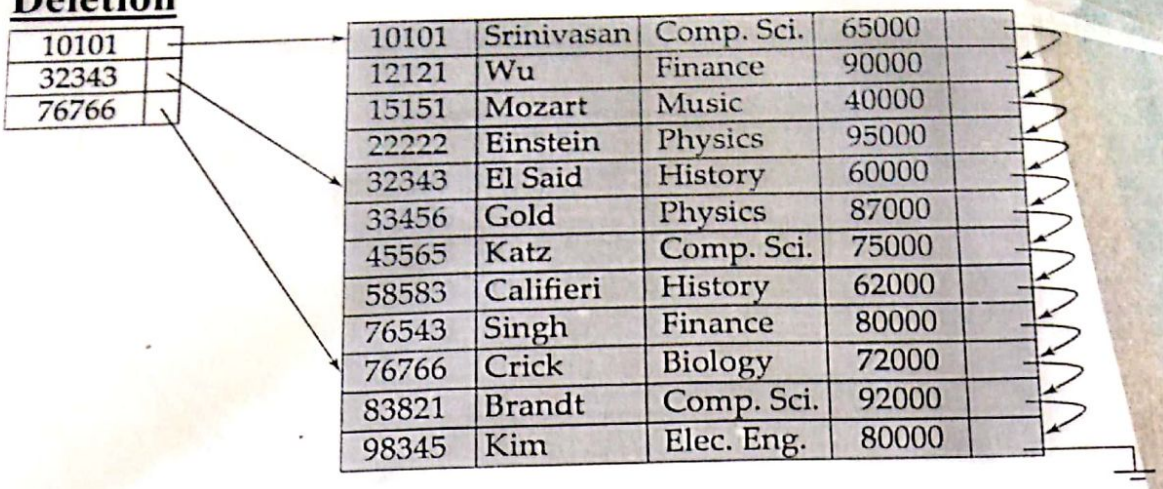
Multilevel Index (combination of primary & sparse index)

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion
- from the file.



Index Update:

Deletion



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

Single-level index entry deletion:

Dense indices – deletion of search-key is similar to file record deletion.

Sparse indices – If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).

If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

Single-level index insertion:

Perform a lookup using the search-key value appearing in the record to be inserted.

Dense indices – if the search-key value does not appear in the index, insert it.

Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new

block is created. If a new block is created, the first search-key value appearing in the new block is inserted into the index.

Multilevel insertion and deletion: algorithms are simple extensions of the single-level algorithms

B^+-Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files.

Disadvantage of indexed-sequential files:

- Performance degrades as file grows, since many overflow blocks get created.
- Periodic reorganization of entire file is required.

Advantage of B^+-tree index files:

- Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
- Reorganization of entire file is not required to maintain performance.

(Minor) disadvantage of B^+-trees:

Extra insertion and deletion overhead, space overhead.

- Advantages of B^+-trees outweigh disadvantages, so B^+-trees are used extensively

B^+-tree is a rooted tree satisfying the following properties:

All paths from root to leaf are of the same length

Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.

A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values

Special cases:

If the root is not a leaf, it has at least 2 children.

If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

Hashing

Two types of Hashing – Static and Dynamic

Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*.
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

A hash function h maps a search-key value K to an address of a bucket is

- Commonly used hash function *hash value mod nB* where nB is the no. of buckets.
- E.g. h(Brighton) = (2+18+9+7+8+20+15+14) mod 10 = 93 mod 10 = 3

- Hash file organization of *instructor* file, using *deptname* as k
- There are 10 buckets
- The hash function returns the sum of the binary representations of the characters modulo 10

E.g. h(Music) = 1 h(History) = 2
h(Physics) = 3 h(Elec. Eng.) = 3

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.

Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.

Handling of Bucket Overflows

Bucket overflow can occur because of

- Insufficient buckets
- Skew in distribution of records. This can occur due to two reasons:
  - i). Multiple records have same search-key value
  - ii). Chosen hash function produces non-uniform distribution of key values

Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets.***

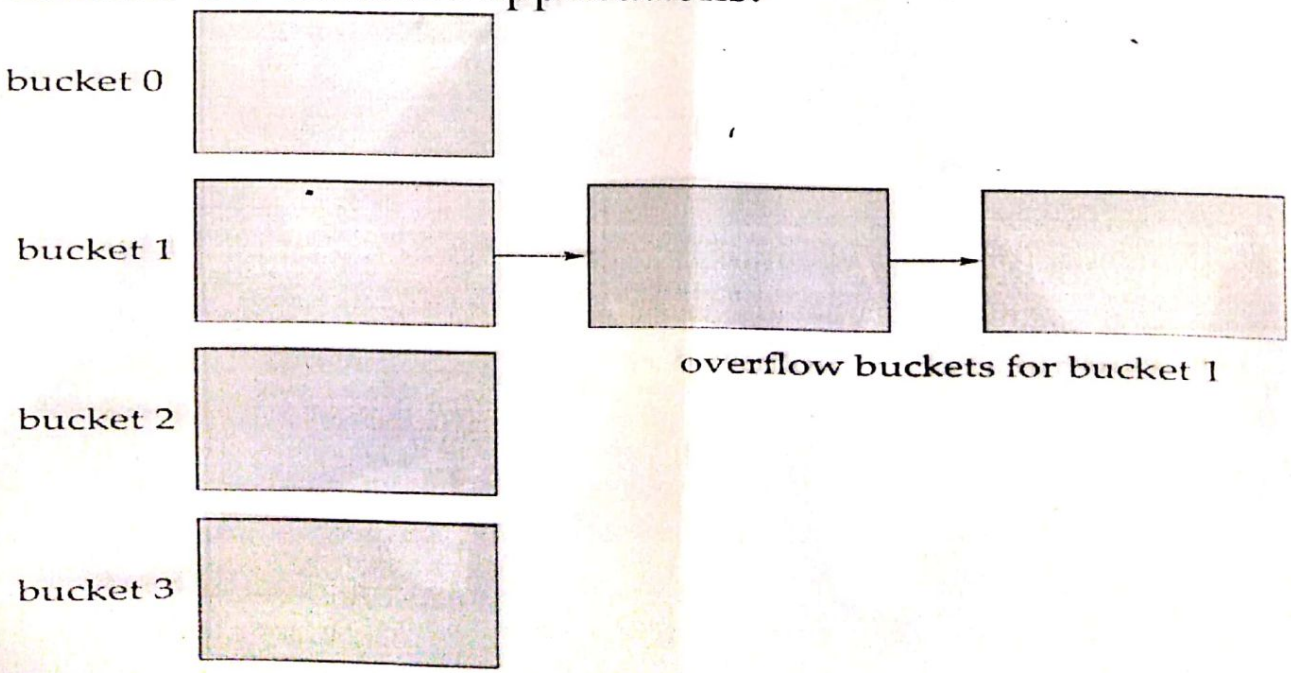
Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list. The scheme is called **closed hashing.** (DBMS)

An alternative, called **open hashing**, the set of buckets is fixed, and there is no overflow buckets, is not suitable for database applications. If the bucket is full, the system inserts records in some other buckets.

i). To use the next bucket (in cyclic order) that has space. This policy is called linear probing.

ii). Compute further hash functions.

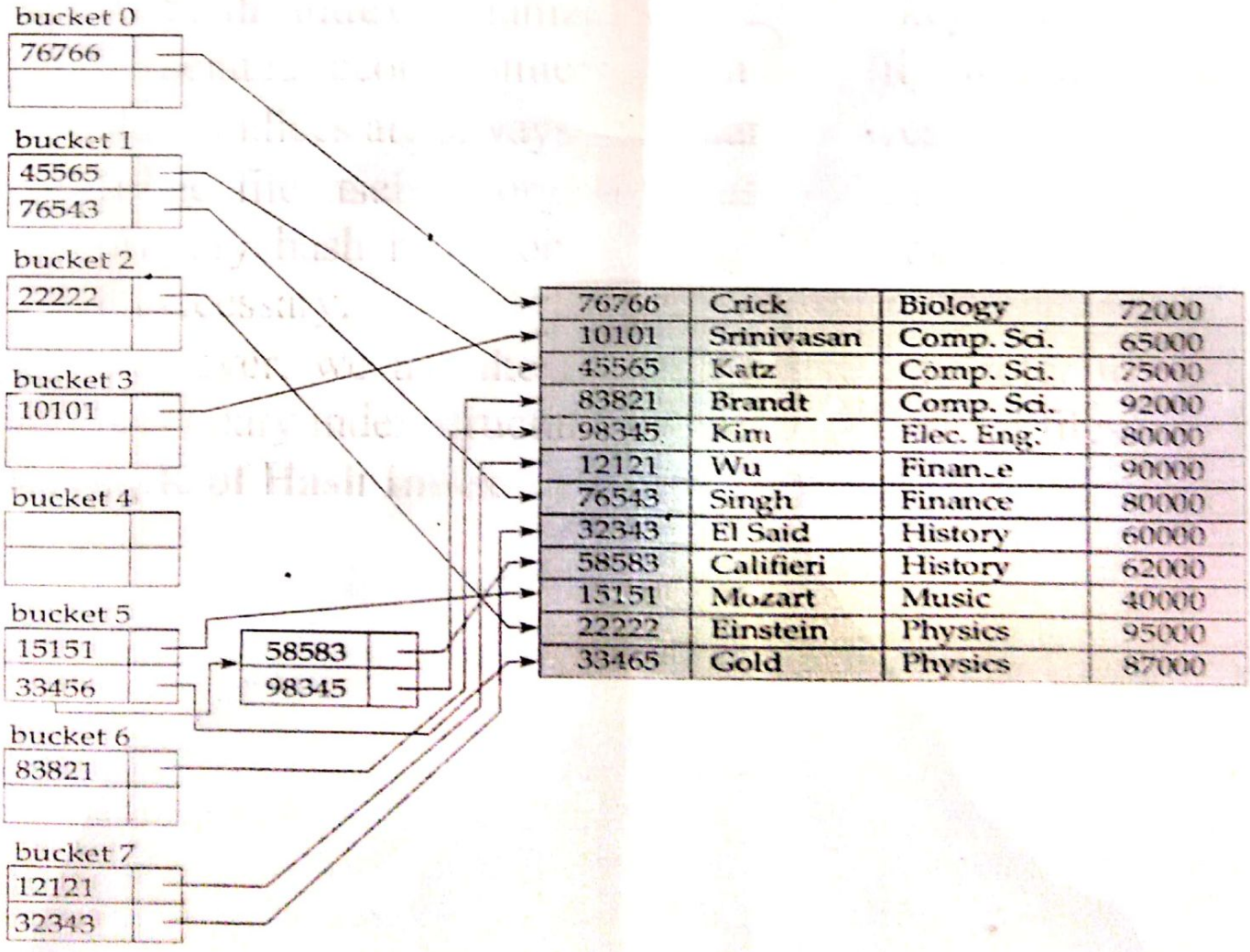
Open hashing is used to construct symbol tables for compilers and assemblers. The deletion under open hashing is troublesome. Compilers and assemblers perform only lookup and insertion operations on their symbol tables. But, in a database system, it is important to be able to handle insertion and as well as deletion. So, open hashing is not suitable for database applications.



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Hash indices are always secondary indices
- If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
- However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index



Deficiencies of Static Hashing

In static hashing, function h maps search-key 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 re-organization of the file with a new hash function -Expensive, disrupts normal operations

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

Dynamic Hashing

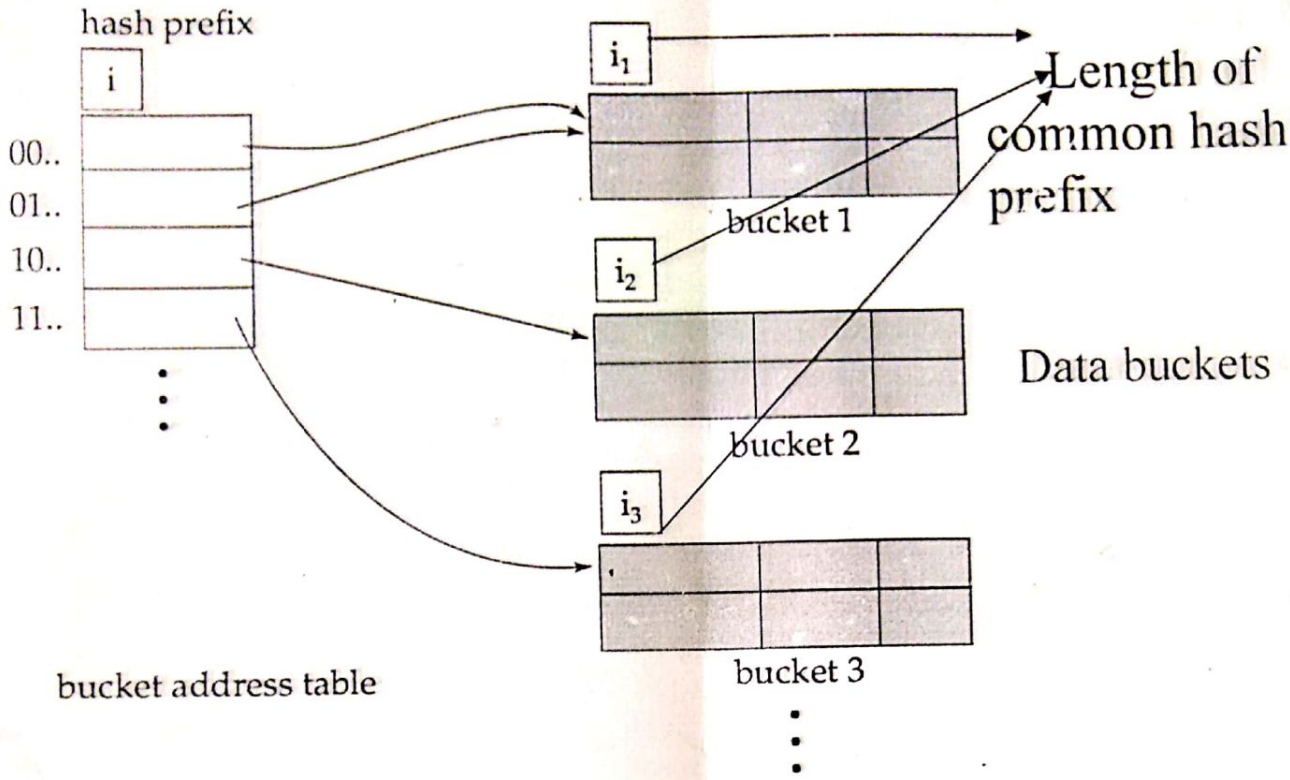
- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
- Hash function generates values over a large range — typically b-bit integers, with $b = 32$.
- At any time use only a prefix of the hash function 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 coalescing and splitting of buckets.

General Extendable Hash Structure

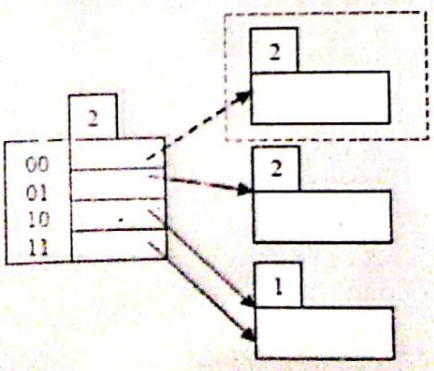


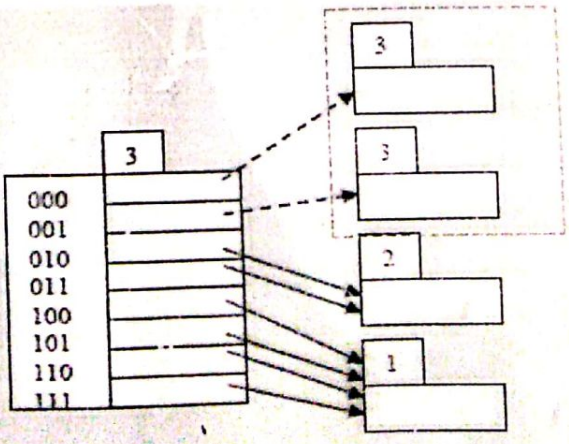
- Hash function returns **b** bits
- Only the prefix **i** bits are used to hash the item
- There are 2^i entries in the bucket address table
- Let i_j be the length of the common hash prefix for data bucket **j**, there is 2^{i-i_j} entries in bucket address table points to **j**

To split a bucket *j* when inserting record with search-key value K_j:

Splitting (Case 1 $i = i_j$)

- Only one entry in bucket address table points to data bucket j
- i++; split data bucket j to j, z; $i_j = i_z = i$; rehash all items previously in j;

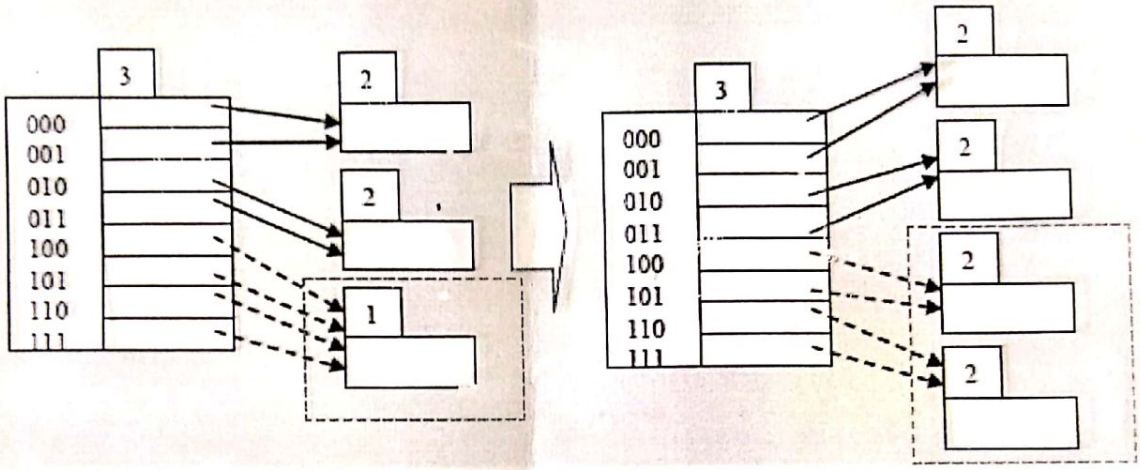




Splitting (Case 2 i > ij)

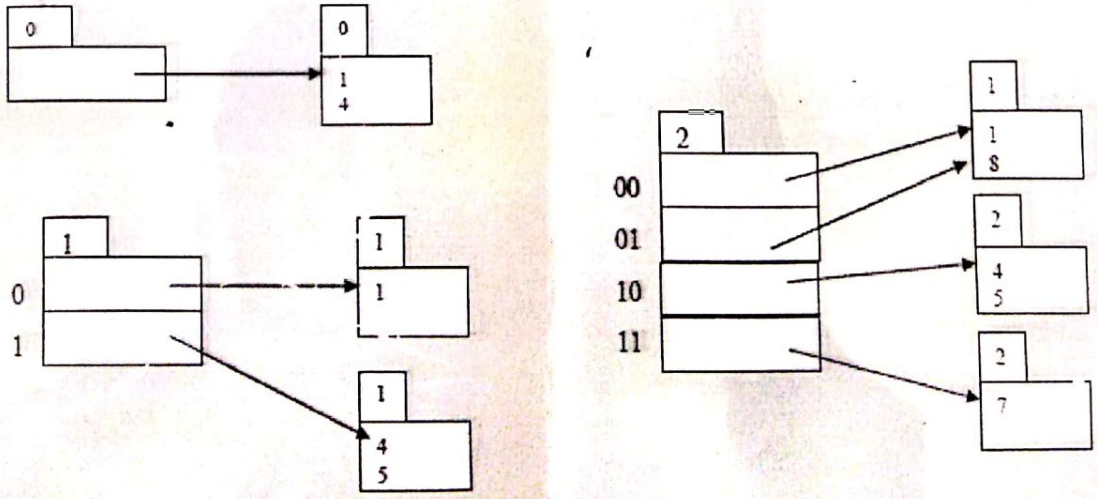
– More than one entry in bucket address table point to data bucket j

– split data bucket j to j, z; Adjust the pointers previously point to j to j and z; rehash all items previously in j;



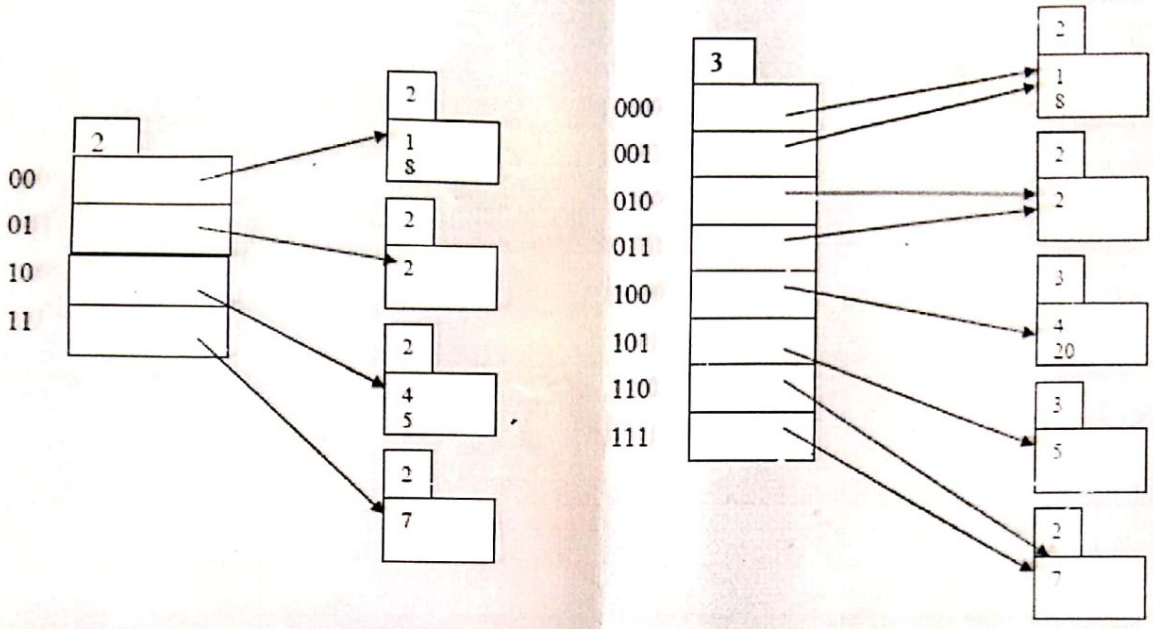
- Example 5: Suppose the hash function is $h(x) = x \bmod 8$ and each bucket can hold at most two records. Show the extendable hash structure after inserting 1, 4, 5, 7, 8, 2, 20.

1	4	5	7	8	2	20
001	100	101	111	000	010	100



inserting 1, 4, 5, 7, 8, 2, 20

1	4	5	7	8	2	20
001	100	101	111	000	010	100

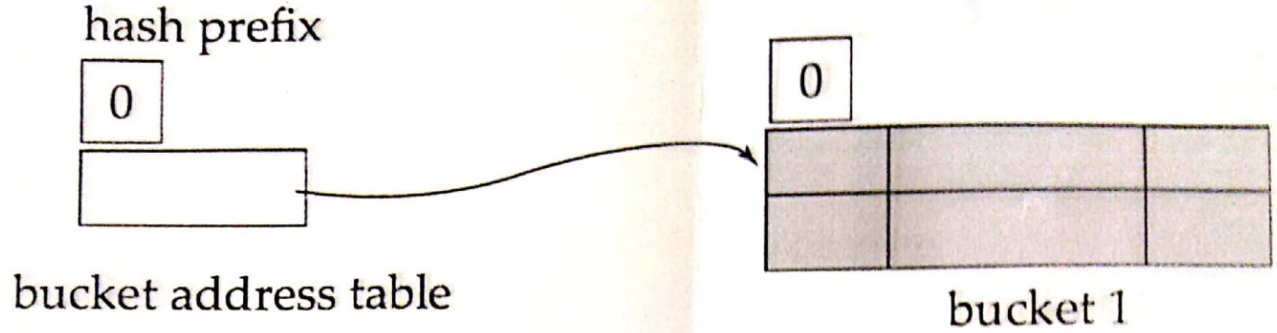


Use of Extendable Hash Structure: Example

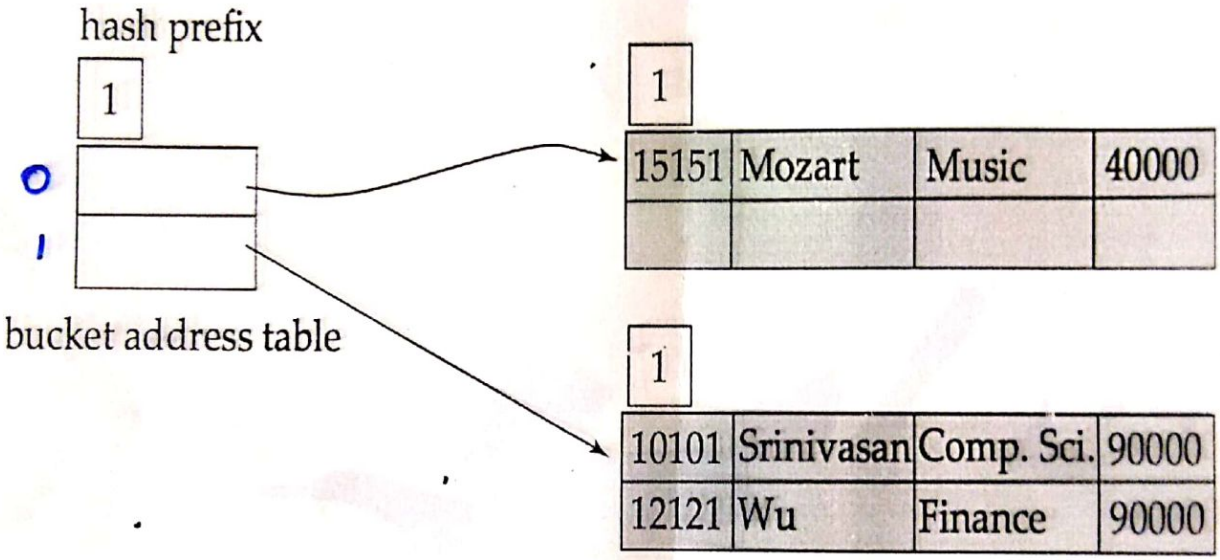
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Biology – 0010
CompSci – 1111
Elect.Eng – 0100
Finance – 1010
History – 1100
Music – 0011
Physics – 1001

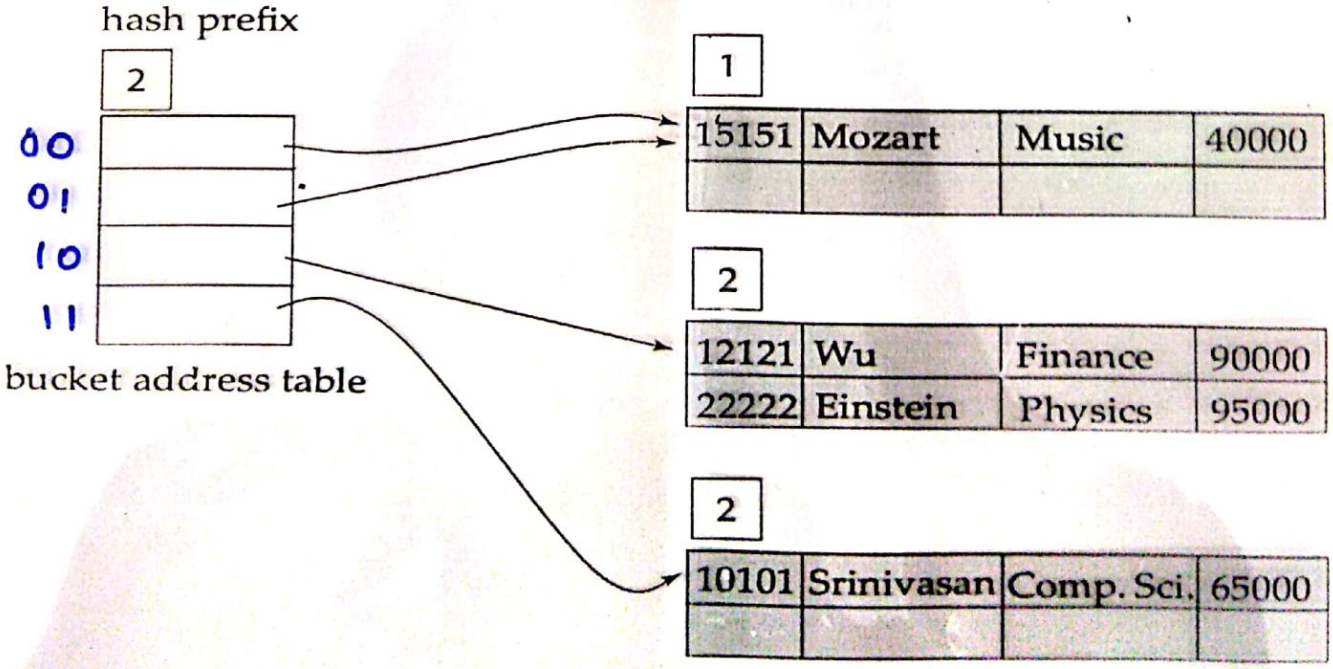
Initial Hash structure; bucket size = 2



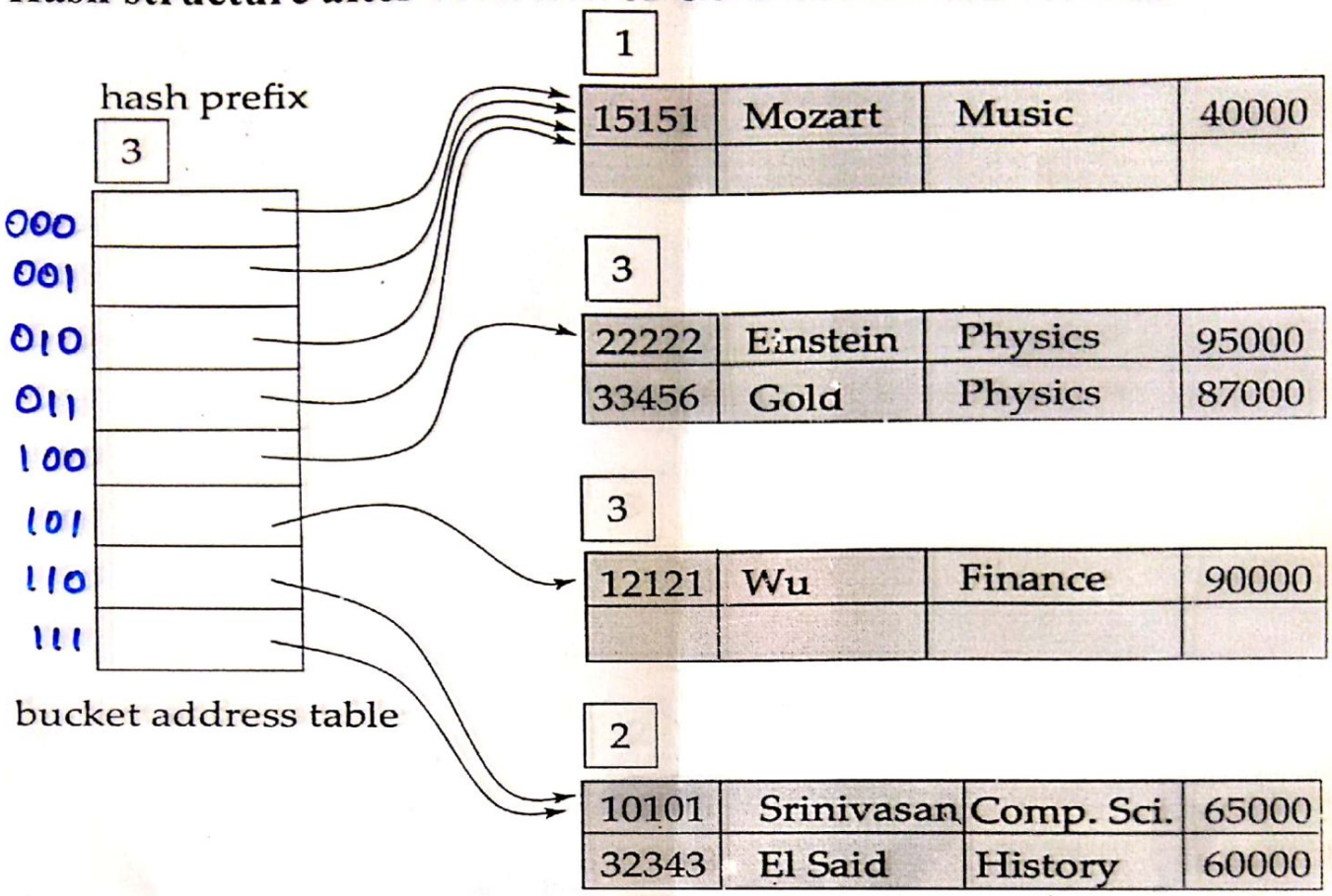
Hash structure after insertion of 'Mozart', 'Srinivasan' and Wu records



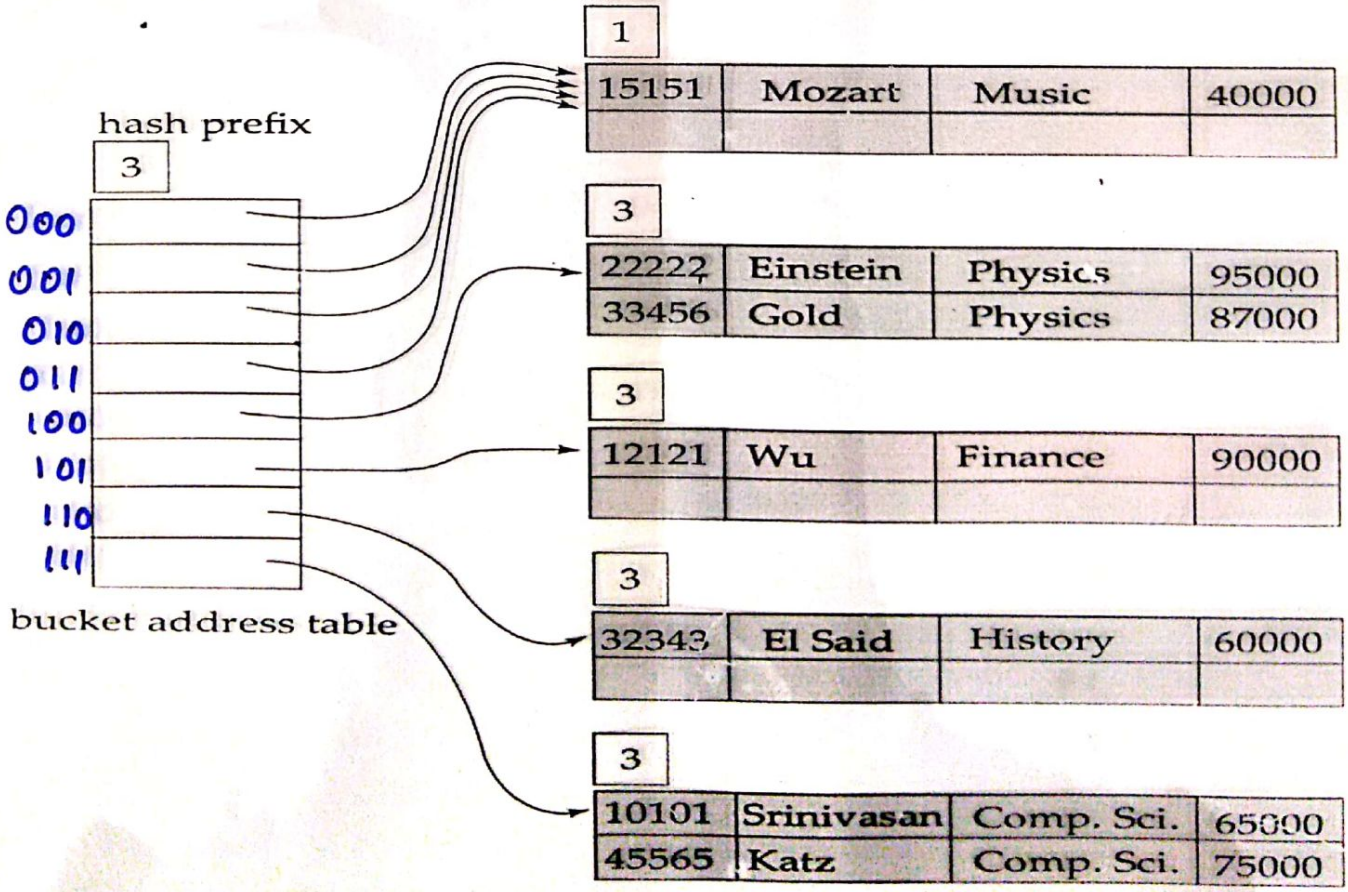
Hash structure after insertion of Einstein record



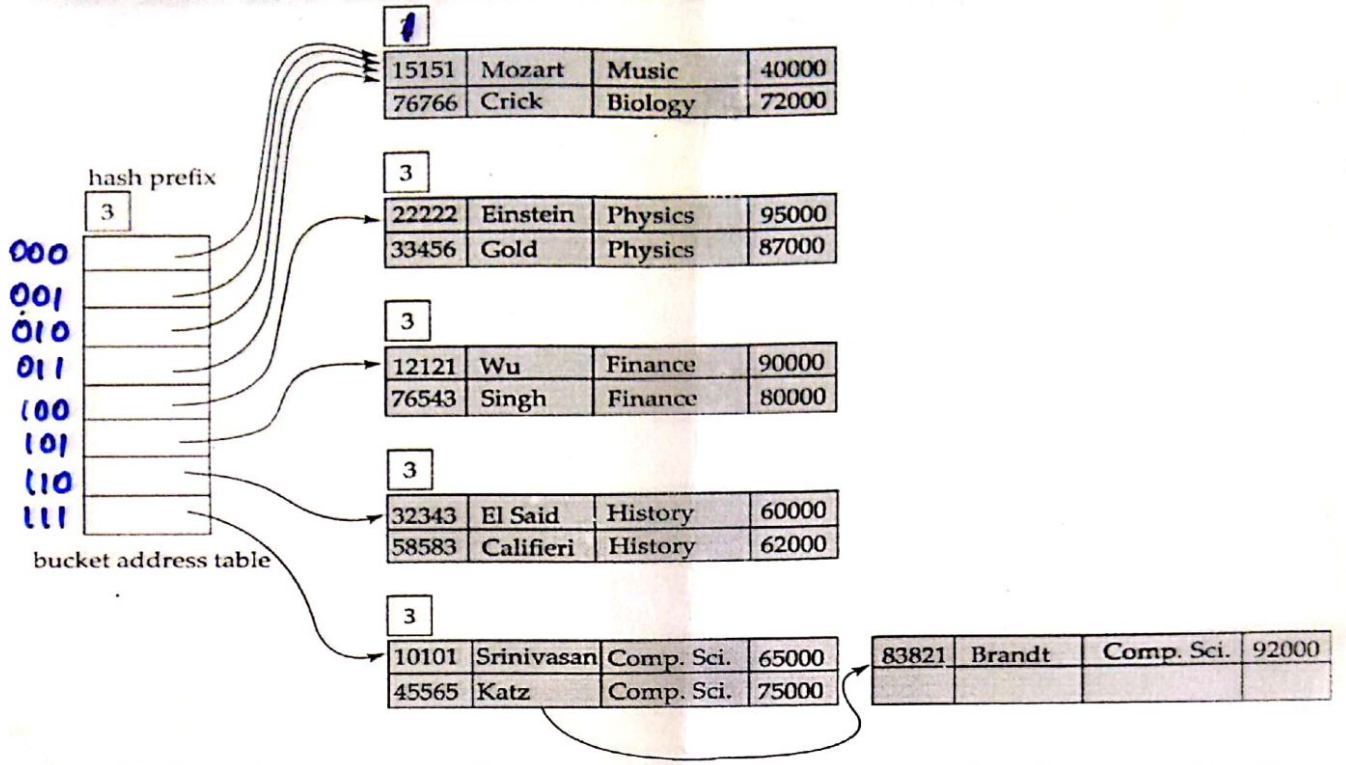
Hash structure after insertion of Gold and ElSaid records



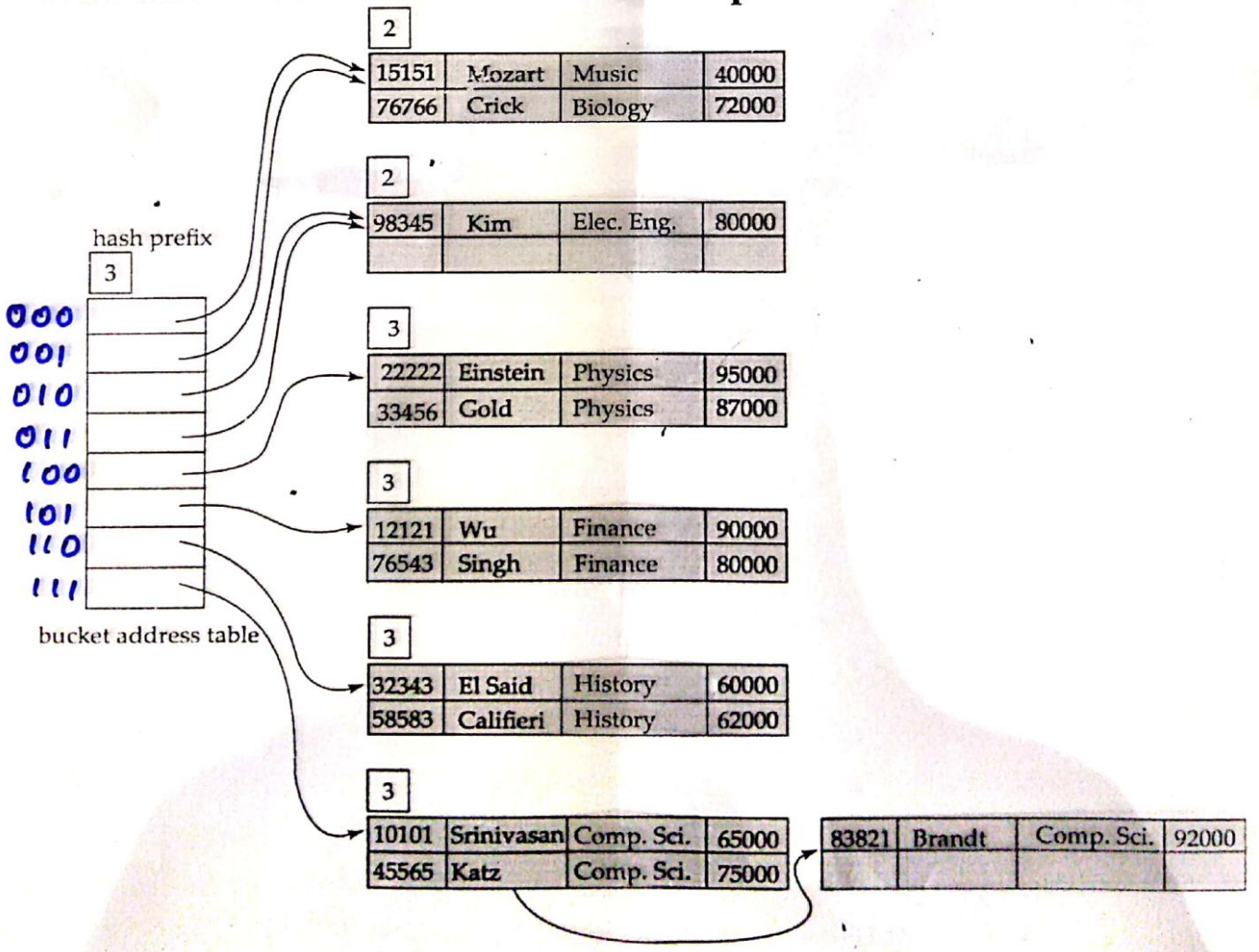
Hash structure after insertion of Katz record



And after insertion of eleven records



And after insertion of Kim record in previous hash structure



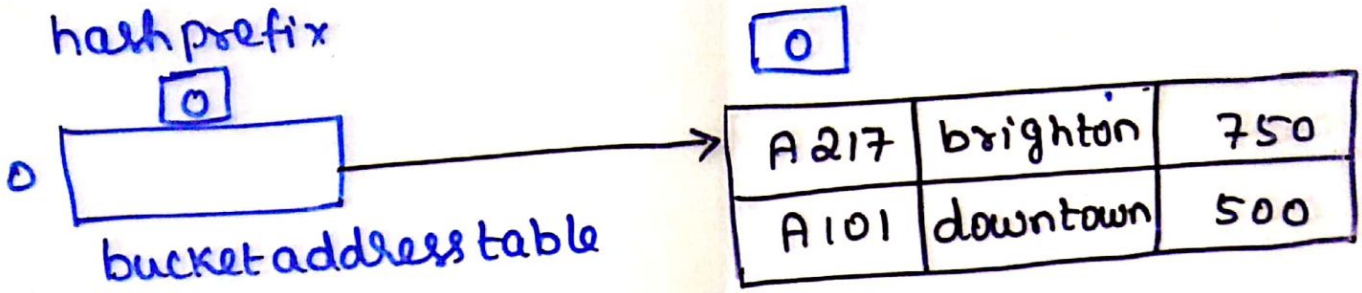
Suppose that we are using extendable hashing on a account file that contains the following records. The 32 bit hash values on branchname is also given. Show the step by step extendable hash structure for account file and buckets can hold two records.

acc #	bname	balance
A217	brighton	750
A101	Downtown	500
A110	Downtown	600
A215	Mianus	700
A102	Perryridge	400
A201	Perryridge	900
A218	Perryridge	700
A222	Redwood	700
A305	Roundhill	350

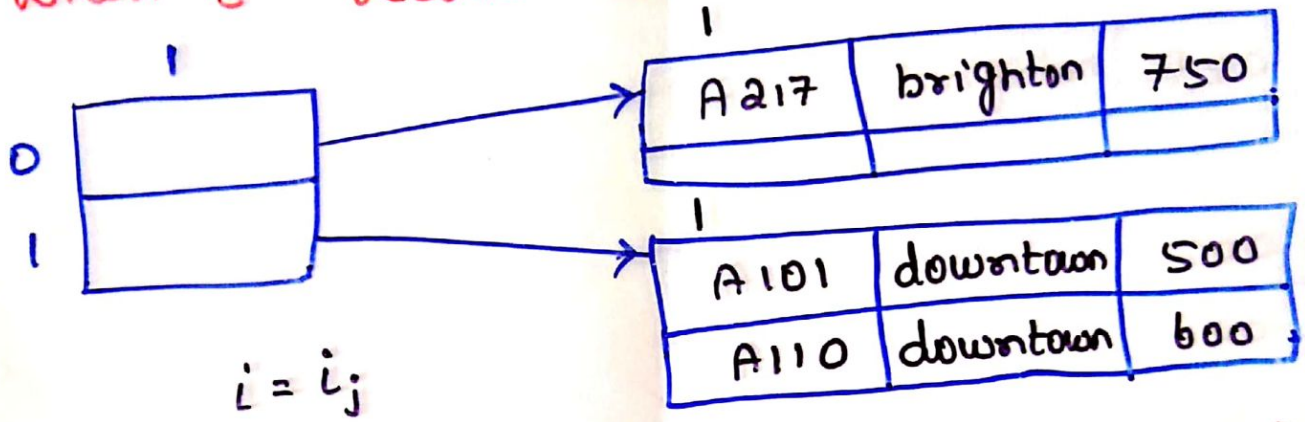
Hash function for branchname

bname	h(branchname)
brighton	0010 - - - - - -
Downtown	1010 - - - - -
Mianus	1100 - - - - - -
Perryridge	1111 - - - -
Redwood	0011 - - - -
Roundhill	1101 - - - - - -

Initial extendable hash function

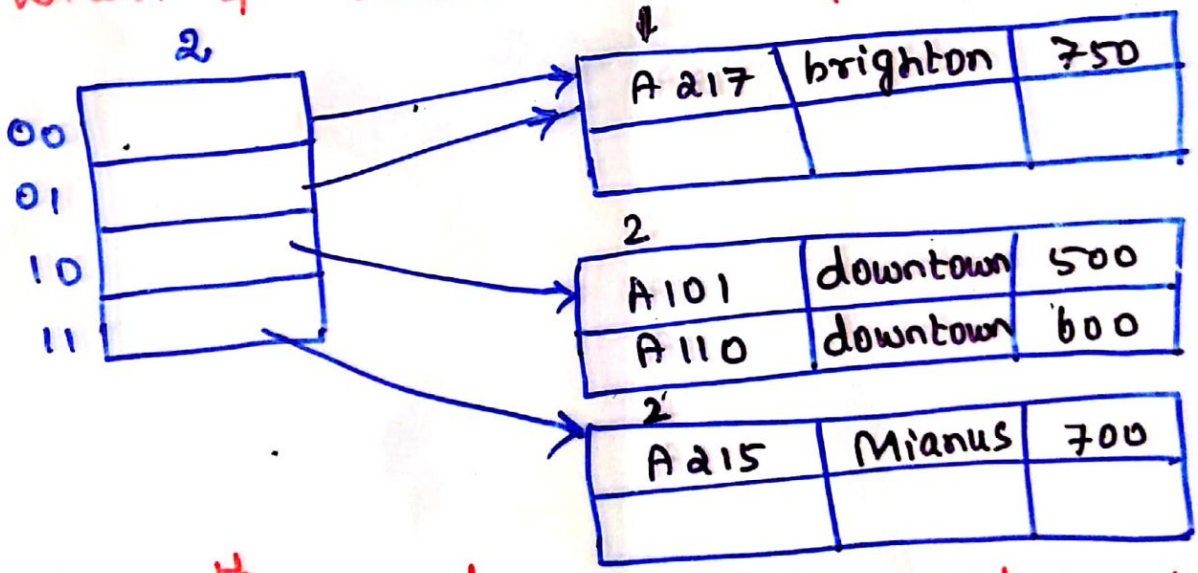


When 3rd record enters,

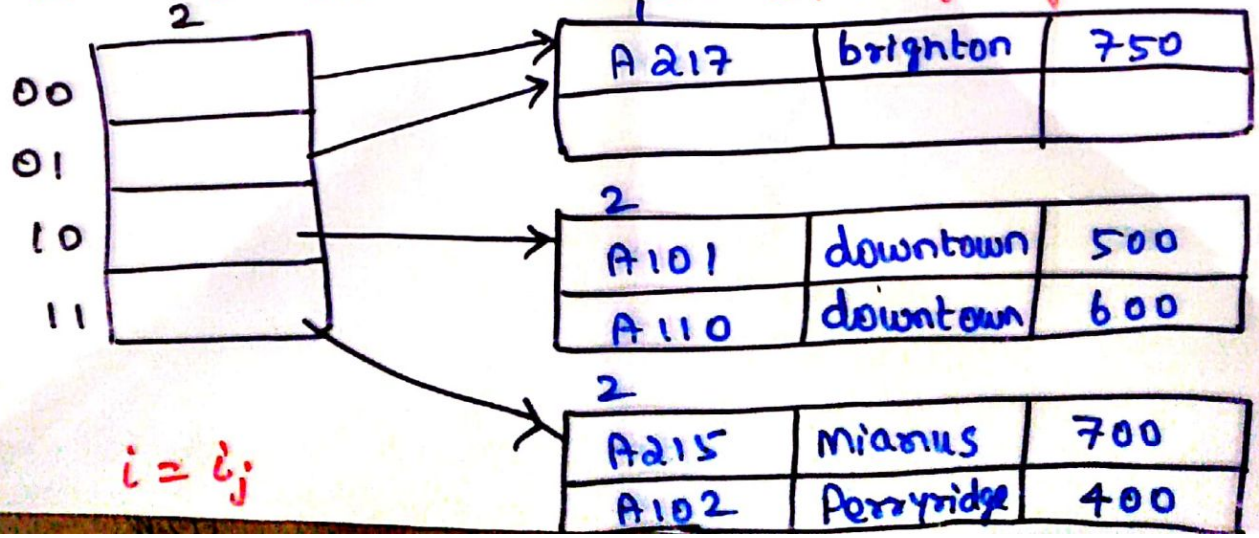


$i = i_j$

When 4th record enters (Mianus – 1100)

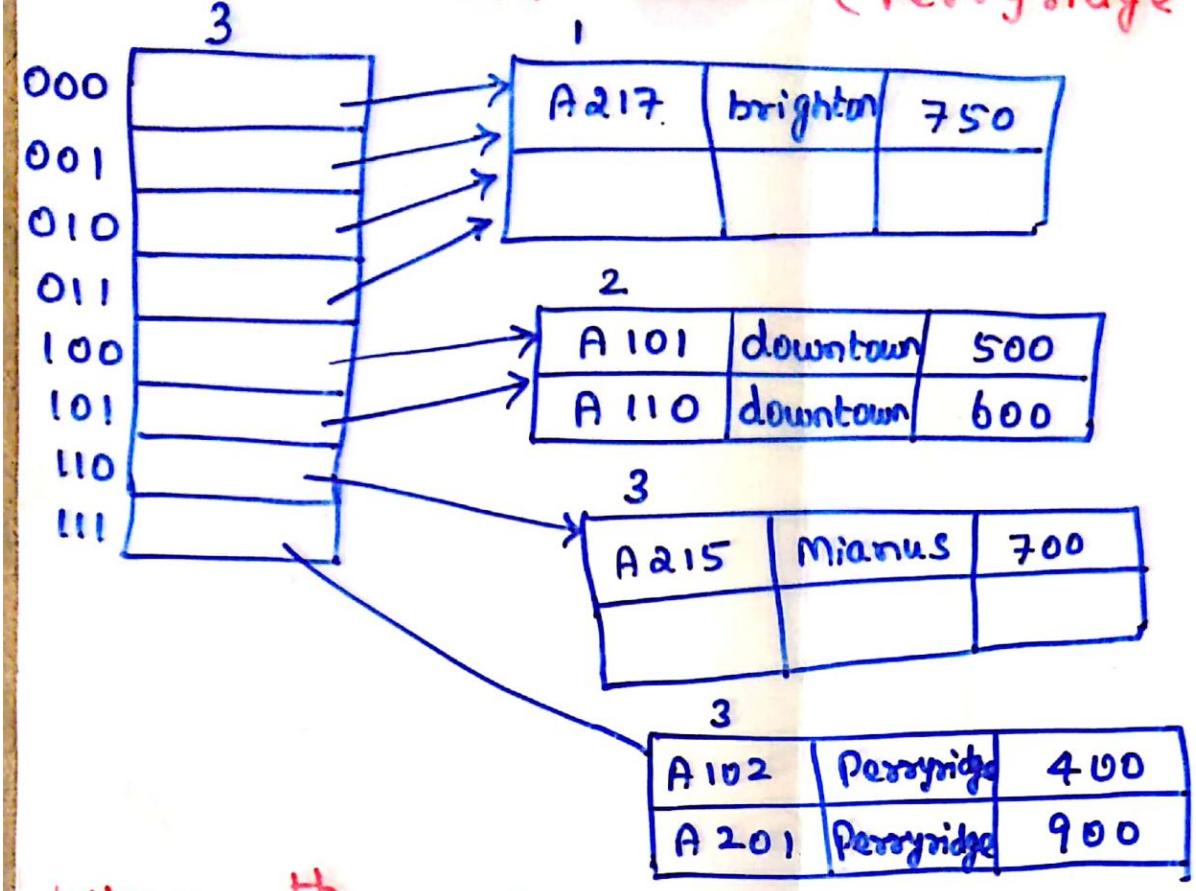


When 5th record enters (Perryridge – 1111)

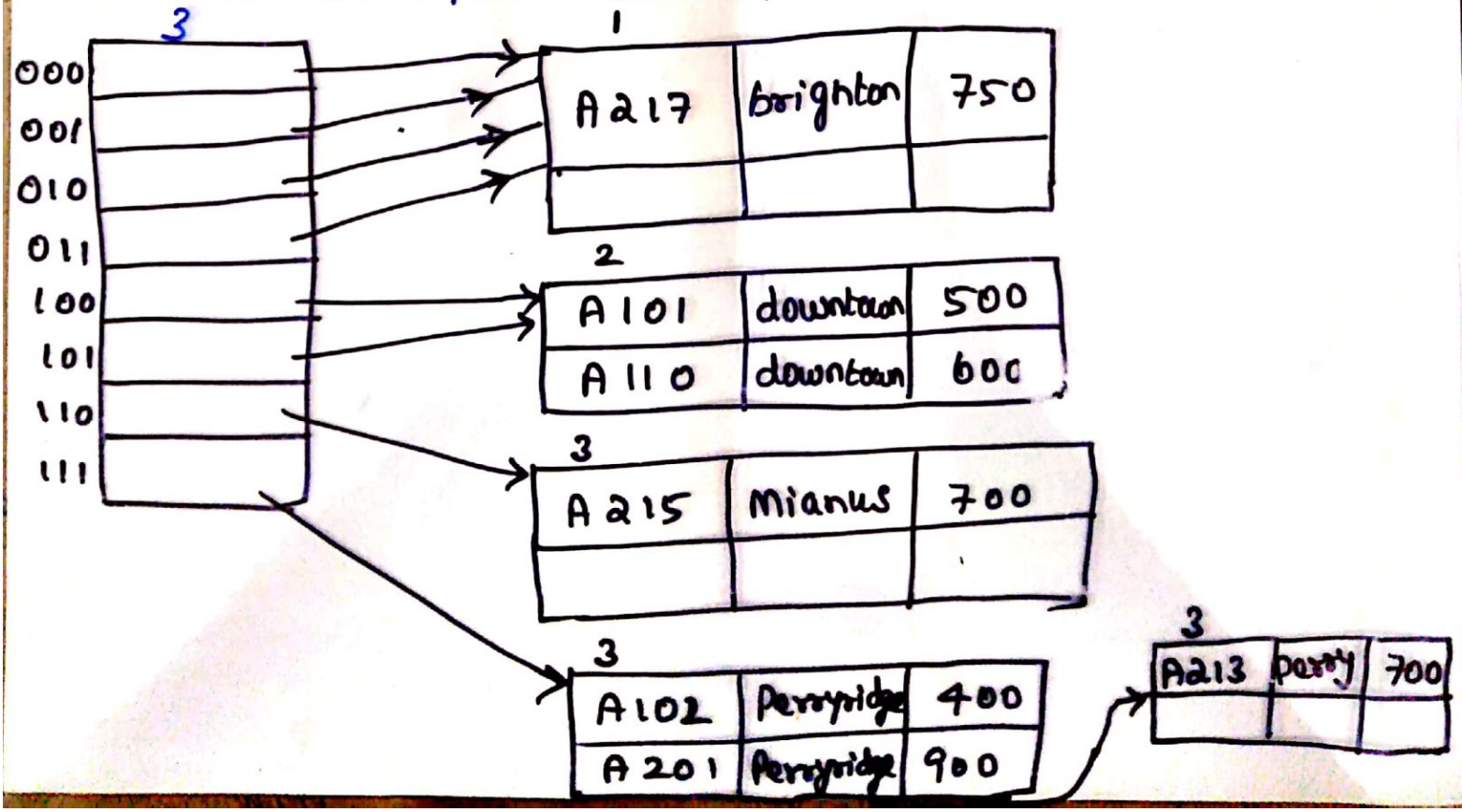


$i = i_j$

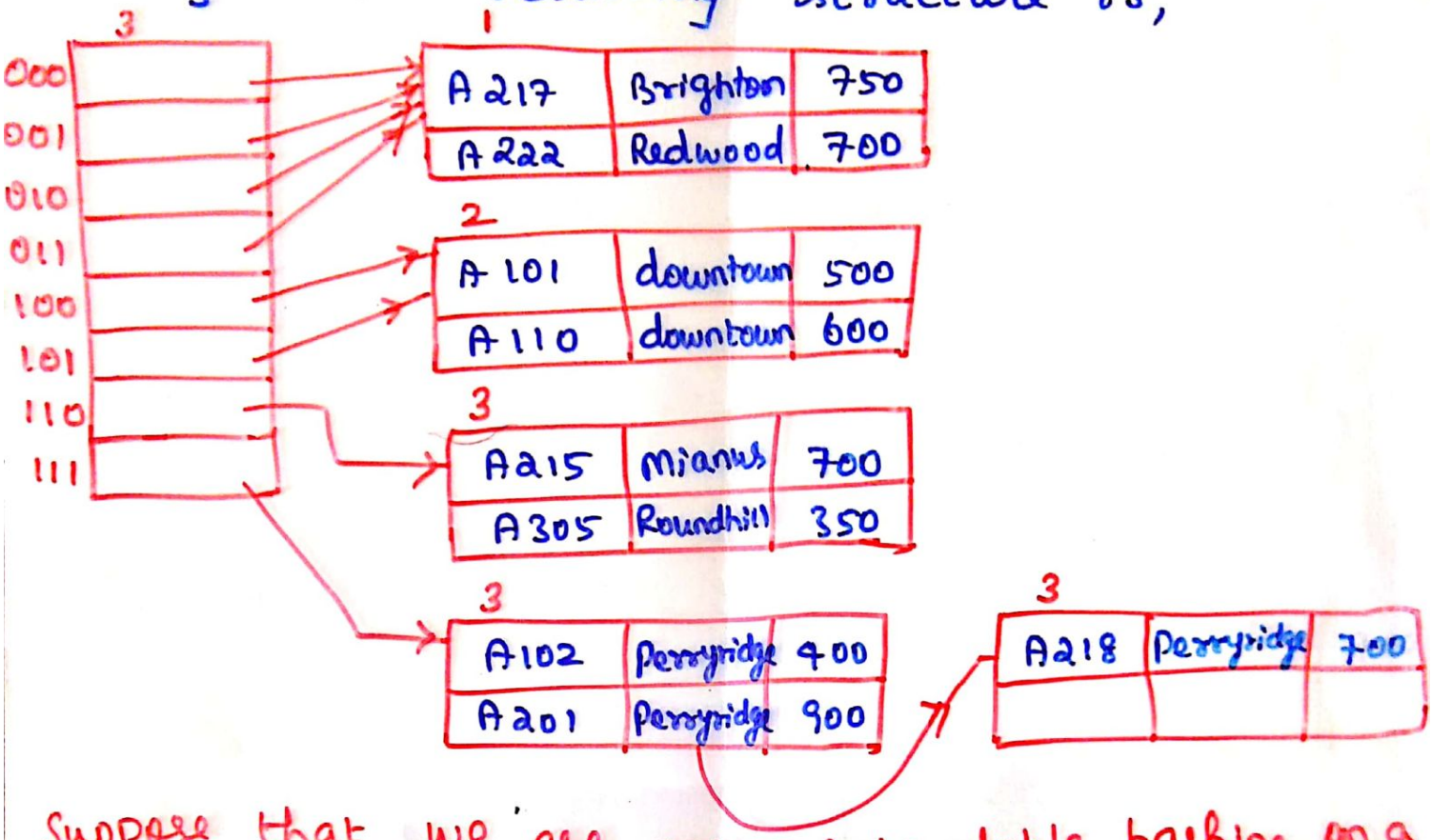
When 6th record enters (Perryridge – 1111)



When 7th record enters (Perryridge – 1111) – It leads to overflow. This overflow cannot handled by Using the number of bits, because these 3 records have same hash value. So, the system uses an overflow bucket.



When 8th record (Redwood - 0011) enters, there is free space in $i = i_1$, When 9th record (Roundhill - 1101) enters, there is free space in $i = i_3$. The resulting structure is,



Suppose that we are using extendable hashing on a file that contains records with the following search key values.

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

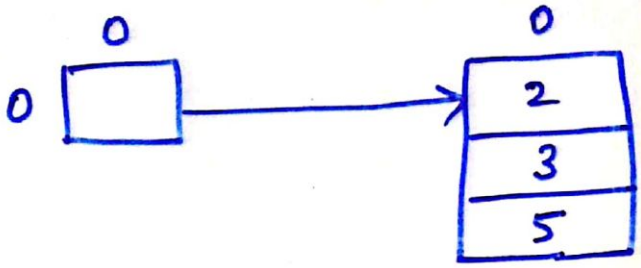
Show the extendable hash structure for this file if the hash function is $h(x) = x \bmod 8$ and buckets can hold three records.

Also find the result of each of the following steps.

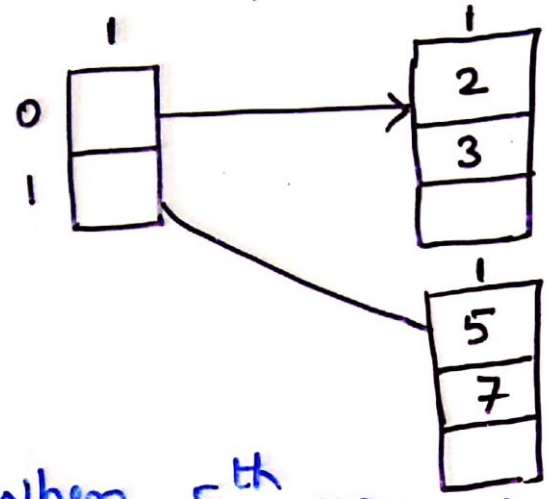
a. Delete 11 b. delete 31 c. insert 1 d. insert 15

x	2	3	5	7	11	17	19	23	29	31
h(x)	2	3	5	7	3	1	3	7	5	7
binary	010	011	101	111	011	001	011	111	101	111

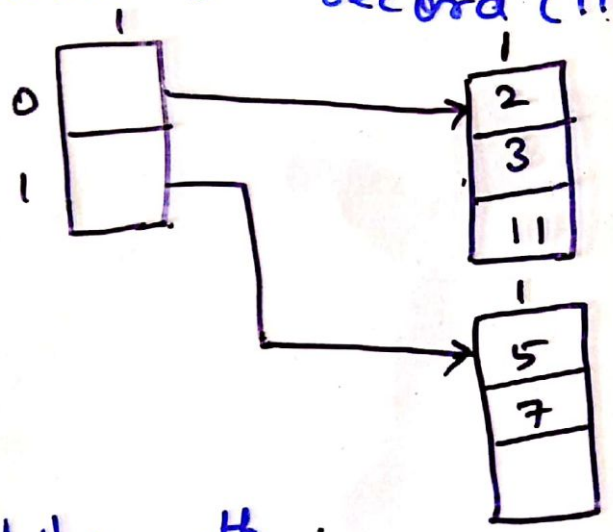
Initial extendable hash structure,



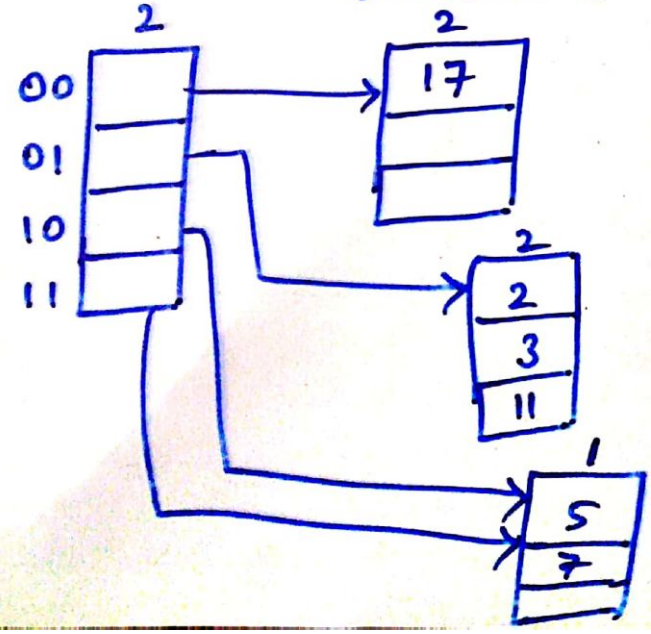
When 4th record (7) enters,



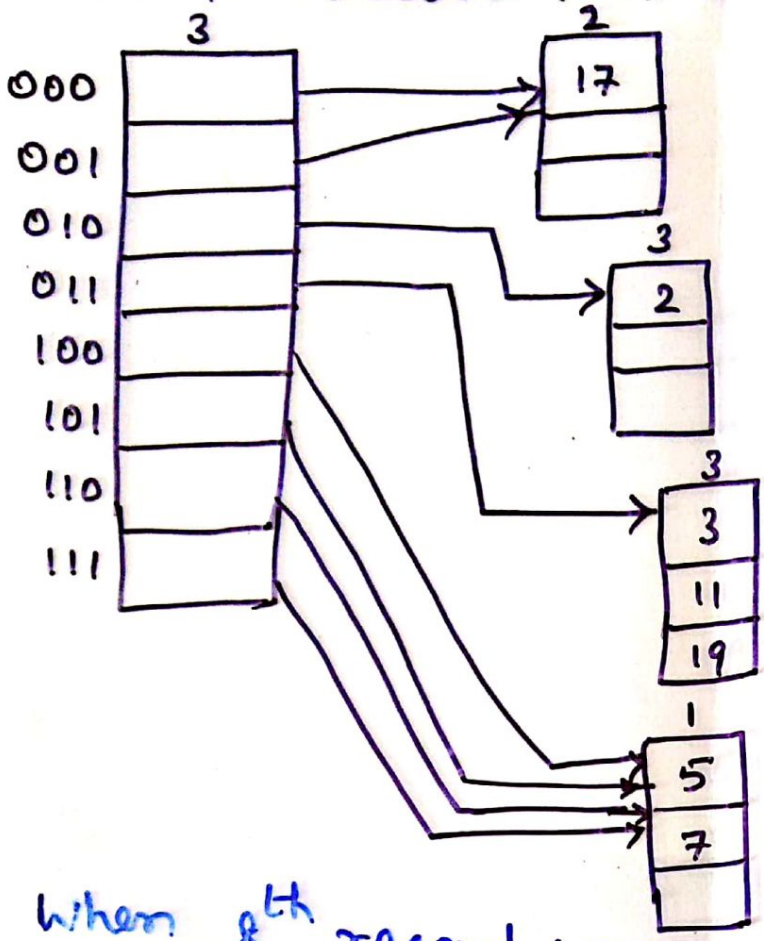
When 5th record (11) enters,



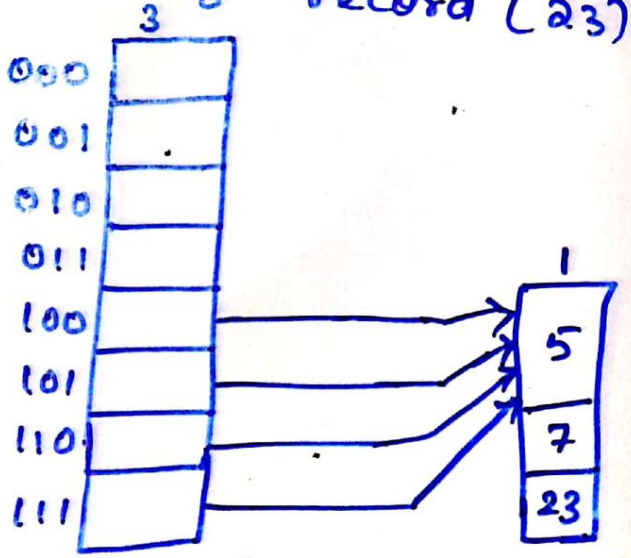
When 6th record (17) enters



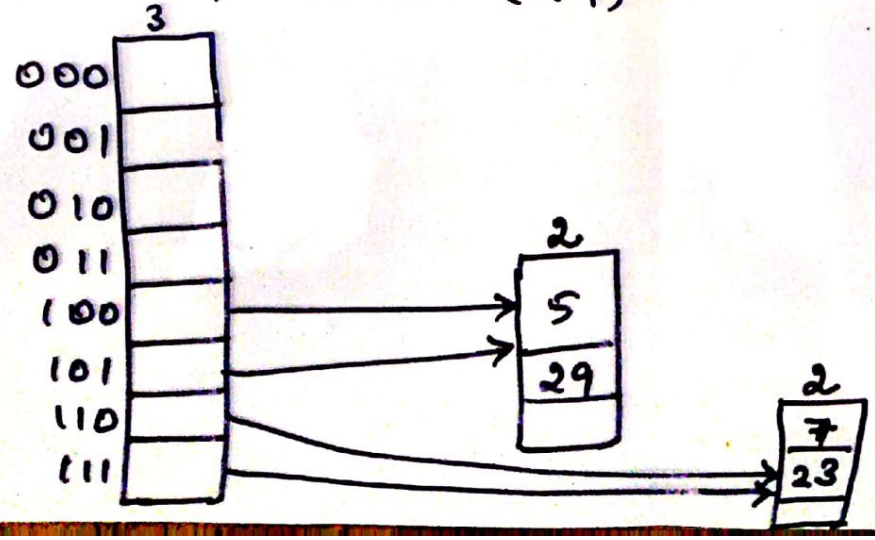
When 7th record (19) enters

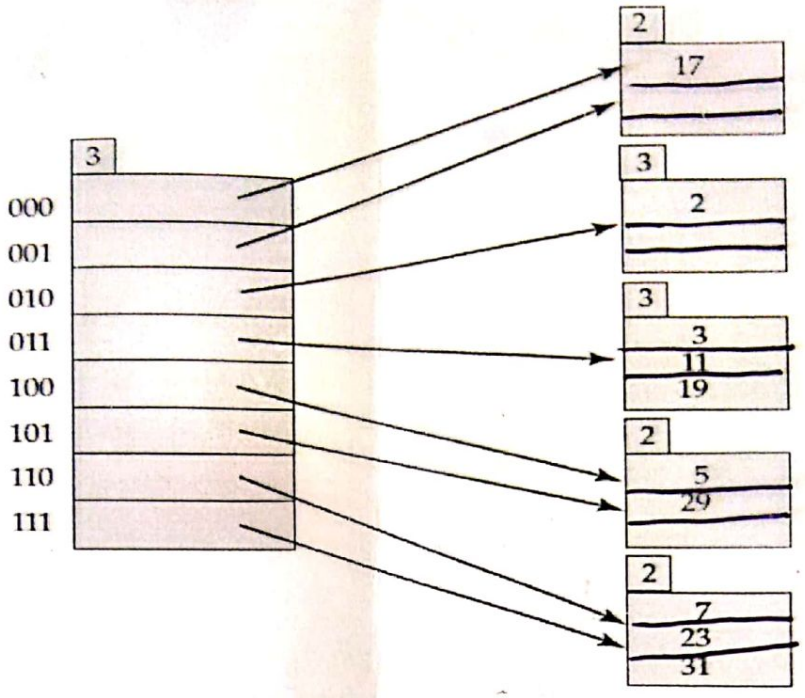


When 8th record (23) enters,

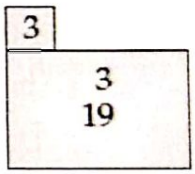


When 9th record (29) enters

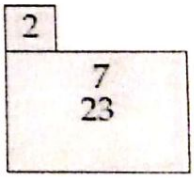




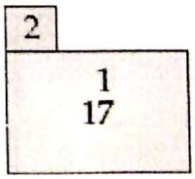
Delete 11, change the third bucket to:



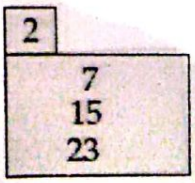
Delete 31, change the last bucket to:



Insert 1: change the first bucket to:



Insert 15: change the last bucket to:



Lookup, insertion and deletion in dynamic hashing

- Each bucket j stores a value i_j, All the entries that point to the same bucket have the same values on the first i_j bits.

To locate the bucket containing search-key K_j:

1. Compute $h(K_j) = X$
2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket

To insert a record with search-key value K_j

Step1: Follow same procedure as look-up and locate the bucket, say j.

Step 2: If there is room in the bucket j insert record in the bucket.

Step3: Else the bucket must be split and insertion re-attempted

Step4: Overflow buckets used instead in some cases

To delete a key value,

- Locate it in its bucket and remove it.
- The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
- Coalescing of buckets can be done
- Decreasing bucket address table size is also possible
- Decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Advantages: i) Performance does not degrade as the file grows.
ii) It saves memory space.

DisAdvantages: i) Lookup involves an additional level of indirection, since the system access the bucket address table before accessing the bucket itself. ii) Implementation of extendable hashing involves additional complexity.

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially
- Applicable on attributes that take on a relatively small number of distinct values

E.g. gender, country, state, …

E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)

- A bitmap is simply an array of bits

In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute

- Bitmap has as many bits as records
- In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

- Bitmap indices are useful for queries on multiple attributes, not particularly useful for single attribute queries
- Queries are answered using bitmap operations

Intersection (and), Union (or), Complementation (not)

Bitmaps can be combined using the logical operations AND, OR, NOT, Minus.

A	B	A AND B	A OR B	NOT A	NOT B	A MINUS B
0	0	0	0	1	1	0
0	1	0	1	1	0	0
1	0	0	1	0	1	1
1	1	1	1	0	0	0

- **A MINUS B is equivalent to A AND NOT B**

Problem:

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

1. Construct a bitmap index on the attribute salary, dividing salary values into 4 ranges: below 50000, 50000 to below 60000, 60000 to below 70000 and above.

4-1 -b

Bitmap for *salary*, with S_1, S_2, S_3 and S_4 representing the given intervals in the same order

S_1	0	0	1	0	0	0	0	0	0	0	0	0
S_2	0	0	0	0	0	0	0	0	0	0	0	0
S_3	1	0	0	0	1	0	0	1	0	0	0	0
S_4	0	1	0	1	0	1	1	0	1	1	1	1

2. Consider a query that requests all instructors in the Finance department with salary of 80000 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

The question is a bit trivial if there is no bitmap on the *dept_name* attribute. The bitmap for the *dept_name* attribute is:

Comp. Sci	1	0	0	0	0	0	1	0	0	0	1	0
Finance	0	1	0	0	0	0	0	0	1	0	0	0
Music	0	0	1	0	0	0	0	0	0	0	0	0
Physics	0	0	0	1	0	1	0	0	0	0	0	0
History	0	0	0	0	1	0	0	1	0	0	0	0
Biology	0	0	0	0	0	0	0	0	0	1	0	0
Elec. Eng.	0	0	0	0	0	0	0	0	0	0	0	1

Intersection of Finance department bitmap and S_4 bitmap of *salary*.

S_4	0	1	0	1	0	1	1	0	1	1	1	1
Finance	0	1	0	0	0	0	0	0	1	0	0	0
$S_4 \cap$ Finance	0	1	0	0	0	0	0	0	1	0	0	0

- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap

  E.g. 100110 AND 110011 = 100010
  100110 OR 110011 = 110111
  NOT 100110 = 011001

(ex)Males with income level L1: 10010 AND 10100 = 10000

- Can then retrieve required tuples, Counting number of matching tuples is even faster

Bitmap indices generally very small compared with relation size E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation. If number of distinct attribute values is 8, bitmap is only 1% of relation size

Index Definition in SQL

Create an index

create index <index-name> **on** <relation-name> (<attribute-list>)

E.g.: **create index** *b-index* **on** *branch(branch_name)*

Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key. Not really required if SQL **unique** integrity constraint is supported

To drop an index

drop index <index-name>

Most database systems allow specification of type of index, and clustering.

Bitmap Indices

This method is used for very large tables with less unique value columns and is accessed number of time with various retrieval queries. In this method we will have

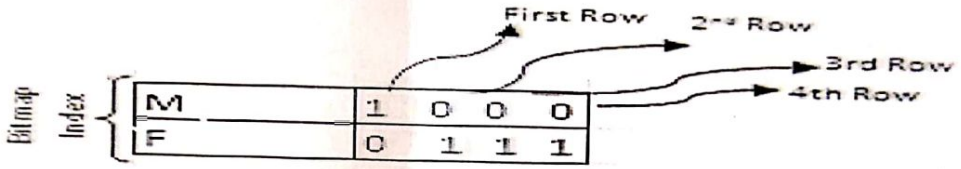
1. As many bits as the number of rows in the table for each of less unique value columns. For example, if the STUDENT table has 10K records, then we will have 10K bits - one bit for each row.
2. Number of bitmap indices created on the column will be equal to number of distinct values in the column. For example, for GENDER column we will have two bitmap indices created - one for male and one for female, and for semester column we will have four bitmap indices created – 1, 2, 3, and 4.
3. If we have any matching value in the column for the row, then that row bit will have '1', else '0'. That means, for GENDER column we will have 2 bitmap indices – one for male and one for female. The bit value for the 'male' bit map index will be 1, if that row has GENDER as 'M', else '0'.

STUDENT

STUDENT_ID	STUDENT_NAME	ADDRESS	AGE	GENDER	SEMESTER
100	Joseph	Alaiedon Township	20	M	1
101	Allen	Fraser Township	21	F	1
102	Chris	Clinton Township	20	F	2
103	Patty	Troy	22	F	4

1. As per rule 1, we will have four rows in the table and hence we will have bits – one bit for each row.

2. The GENDER column has only two values - 'M' and 'F'. Hence we will have two bitmap indices – one for 'M' and one for 'F'.
3. Now the bitmap index for GENDER column is as below. Here Bitmap index 'M' has value '1000' indicating first row has gender as 'M' and rest of the rows do not have gender as 'M'. Similarly, bitmap index 'F' is '0111'



Similarly bitmap index for Semester can be as follows:

SEMESTER	
1	1 1 0 0
2	0 0 1 0
3	0 0 0 0
4	0 0 0 1

Find the female students who are in the second semester. Here this query uses two columns to filter the records, where both of them have less unique value columns.

SELECT * FROM STUDENT WHERE GENDER = 'F' AND SEMESTER =2; [The query will search the bitmap index for both of this columns and perform logical 'AND' operation on those indexes to get the actual address of the result.]

Bitmap Index for F: 0 1 1 1 **AND**
Bitmap Index for SEM 2: 0 0 1 0
Result: **0 0 1 0** his implies third row has the result for the query.

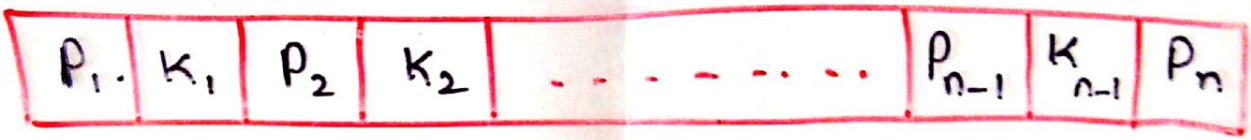
B^+ Tree indexing

* The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data.

* B^+ tree index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.

* A B^+ tree index takes the form of balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.

Structure of a B^+ tree



A typical node of a B^+ tree contains upto $n-1$ search key values $K_1, K_2 \ldots K_{n-1}$ and n pointers $P_1, P_2, \ldots P_n$. The search key values within a node are kept in sorted order.

* nodes are classified as leaf nodes, non leaf nodes and root node.

Leaf nodes: Each leaf node can hold upto $n-1$ values to a minimum of $\lceil (n-1)/2 \rceil$ values.

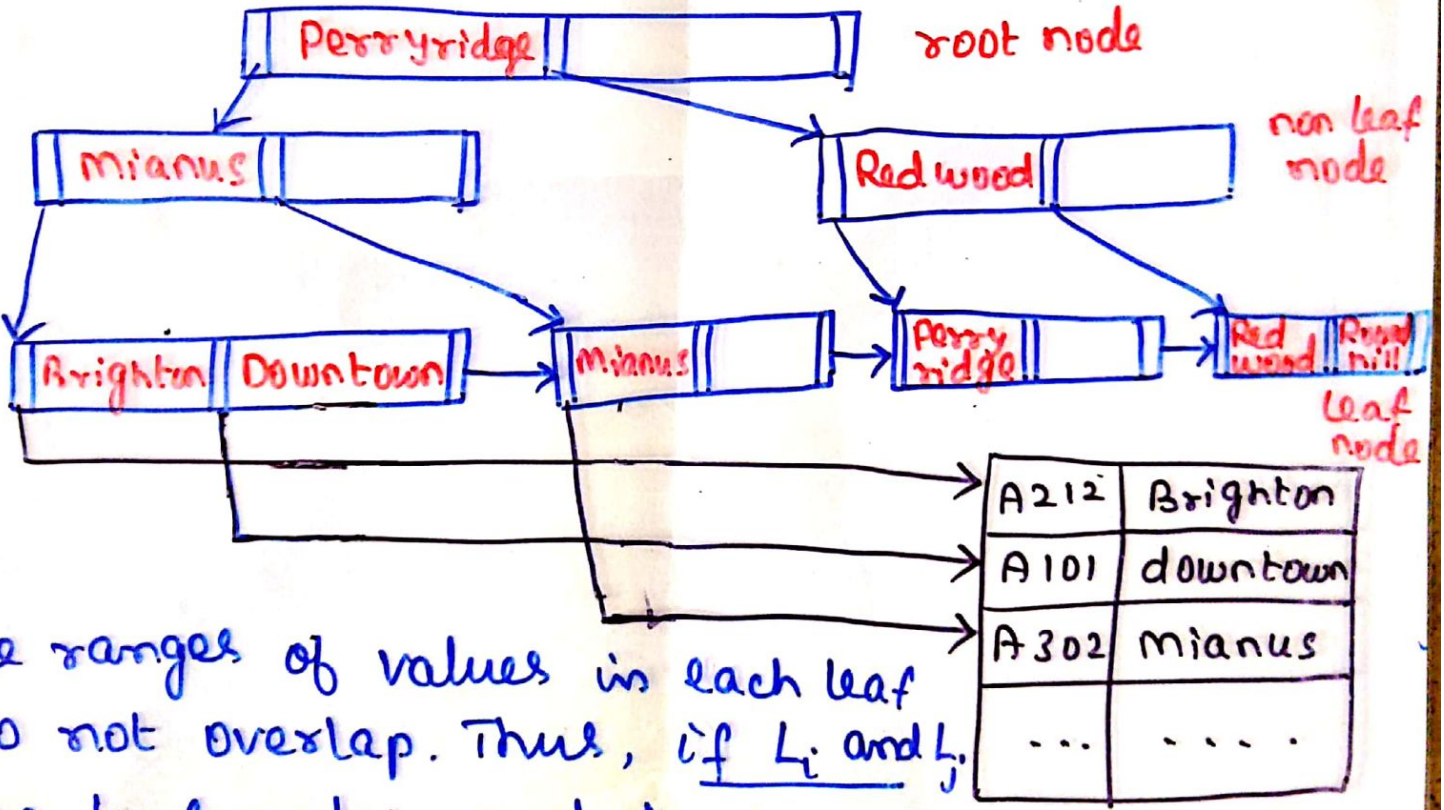
(ex) if $n = 4$ in B^+ tree, each leaf must contain atleast 2 values and at most 3 values.

root node: The root node can hold fewer than $\lceil n/2 \rceil$ pointers. However it must hold atleast

two pointers.

Non leaf nodes! It form a multilevel (sparse) index on the leaf nodes. The structure of non leaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A non leaf node may hold upto n pointers and must hold atleast $\lceil n/2 \rceil$ pointers. Non leaf nodes are also referred to as internal nodes. The number of pointers in a node is called the fanout of the node.

(ex) B+ tree for account file if n = 3



The ranges of values in each leaf do not overlap. Thus, if L_i and L_j are leaf nodes and $i < j$, then every search key value in L_i is less than every search key value in L_j. If the B+ tree index is to be a dense index, every search key value must appear in some leaf node. Since there is a linear order on the leaves based on the search key values, use pointers P_n to chain

together the leaf nodes in search key order. This ordering allow for efficient sequential processing of the file.

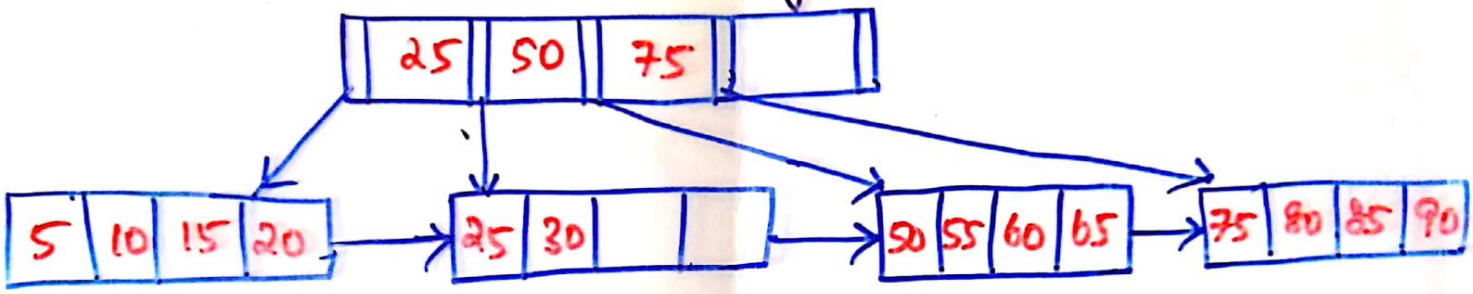
Adding Records to a B^+ tree

* consider three scenarios when add a record to a B^+ tree. Each scenario causes a different action in the insert algorithm.

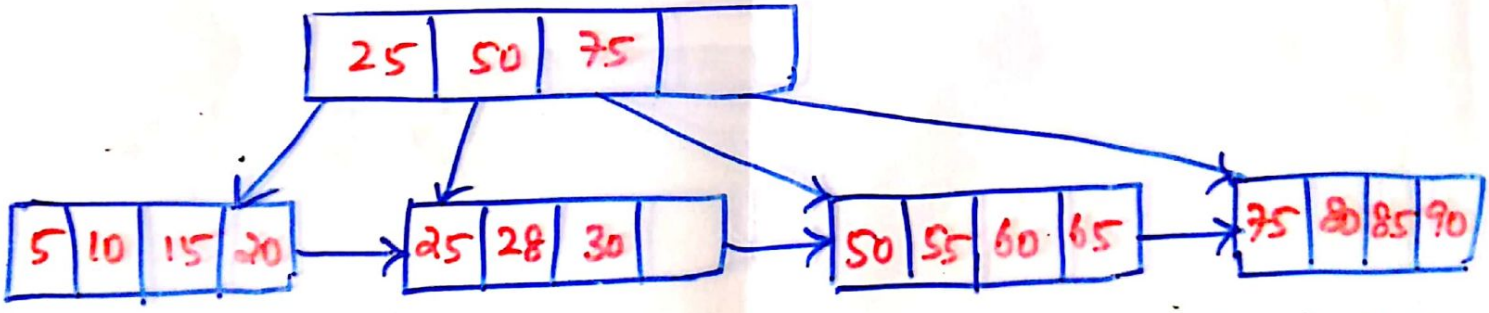
Leaf node full	index node full	Action
NO	NO	Place the record in sorted position in the appropriate leaf node
yes	no	1. Split the leaf node. 2. Place Middle key value in the index node in sorted order. 3. Left Leaf node contains records with keys below the middle key. 4. Right leaf node contains records with keys equal to or greater than the middle key.
yes	yes	1. Split the leaf node. 2. Records with keys < middle key go to the left leaf node. 3. Records with keys >= middle key go to the right leaf node. 4. Split the index node. 5. Keys < middle key go to the left index node.

6. Keys > middle key go to the right index page.
7. The middle key goes to the next (higher level) index.
8. If the next level index page is full, continue splitting the index node.

(ex) Add record with key 28 if n = 5



1) leaf node is not full, so insert a record 28

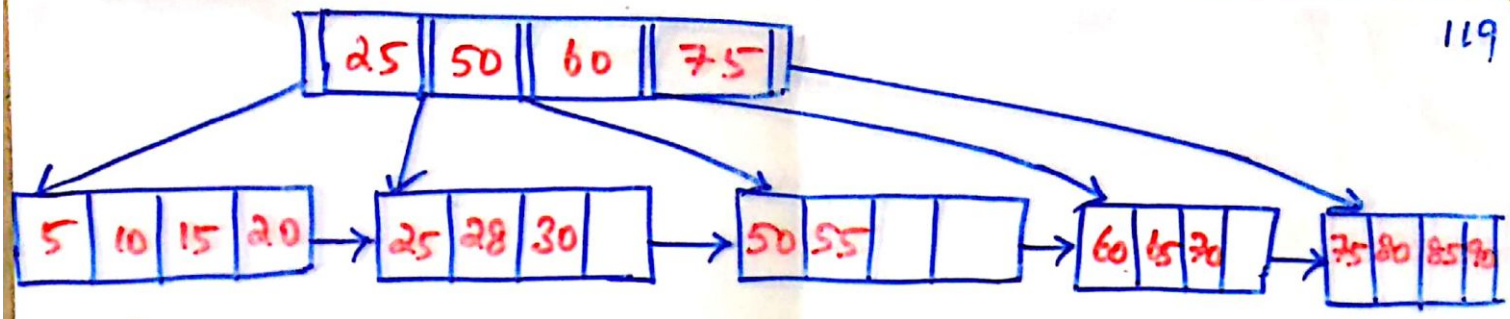


2) Adding a record when the leaf node is full but the index node is not (Insert a record 70)

* This record 70 should go in the leaf node containing 50, 55, 60 and 65. But this node is full. So, we must split the node as,

Left leaf node	Right leaf node
50 55	60 65 70

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



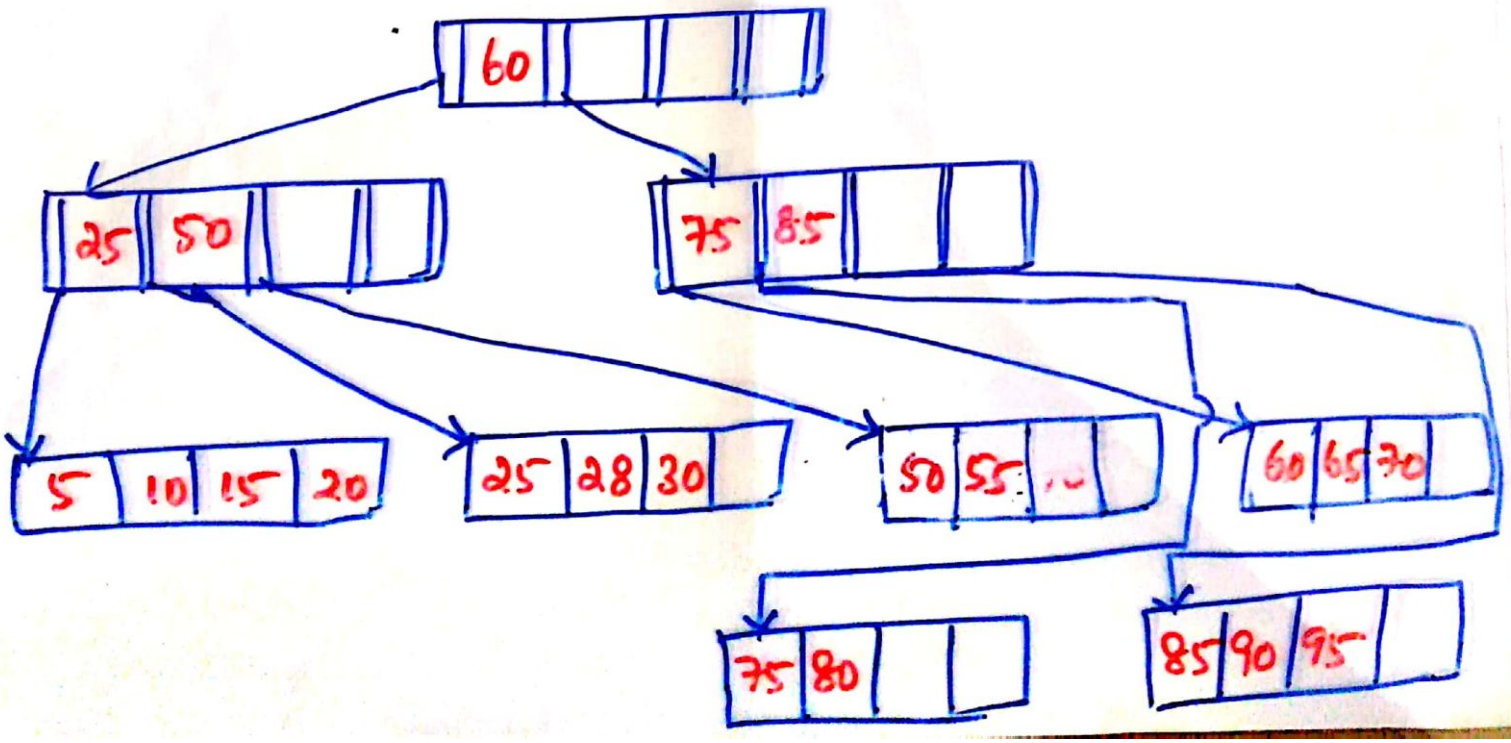
Adding a record when both the leaf node and the index node are full: (Insert a record 95)

* The record 95 should go in the leaf node containing 75, 80, 85 and 90. Since this node is full, split into two nodes.

Left leafnode	Right leafnode
75 80	85 90 95

* The middle key 85 rises to the index node. But, the index node is also full, so split the index node as

Left index node	Right index node	new index node go to the next higher level
25 50	75 85	60



Deleting Keys from a B^+ tree

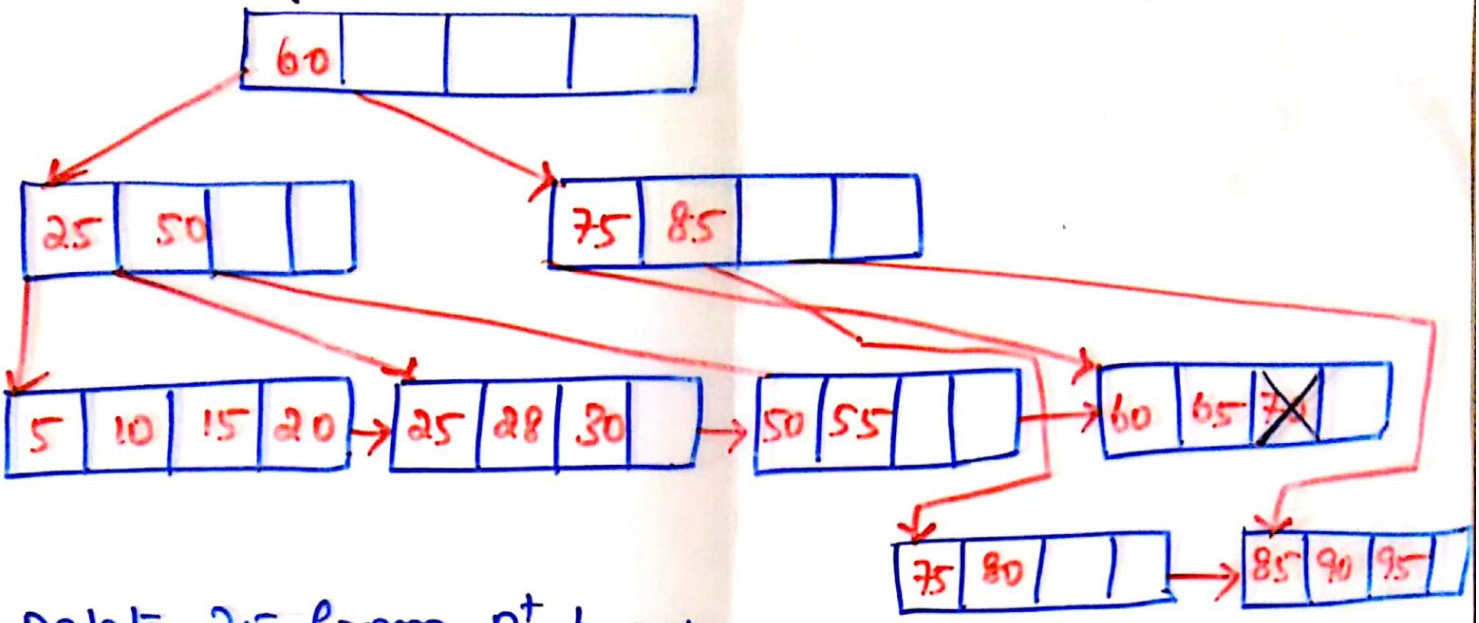
Consider three scenarios to delete a record from a B^+ tree. Each scenario causes a different action in the delete algorithm.

Leaf node below fill factor	index node below fill factor	Action
NO	NO	Delete the record from the leaf node. 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 node and its sibling. Change the index node to reflect the change.
yes	yes	1. Combine the leaf node and its sibling 2. Adjust the index node to reflect the change. 3. Combine the index node with its sibling. Continue combining index nodes until to reach a node with the correct fill factor (or) reach the root node.

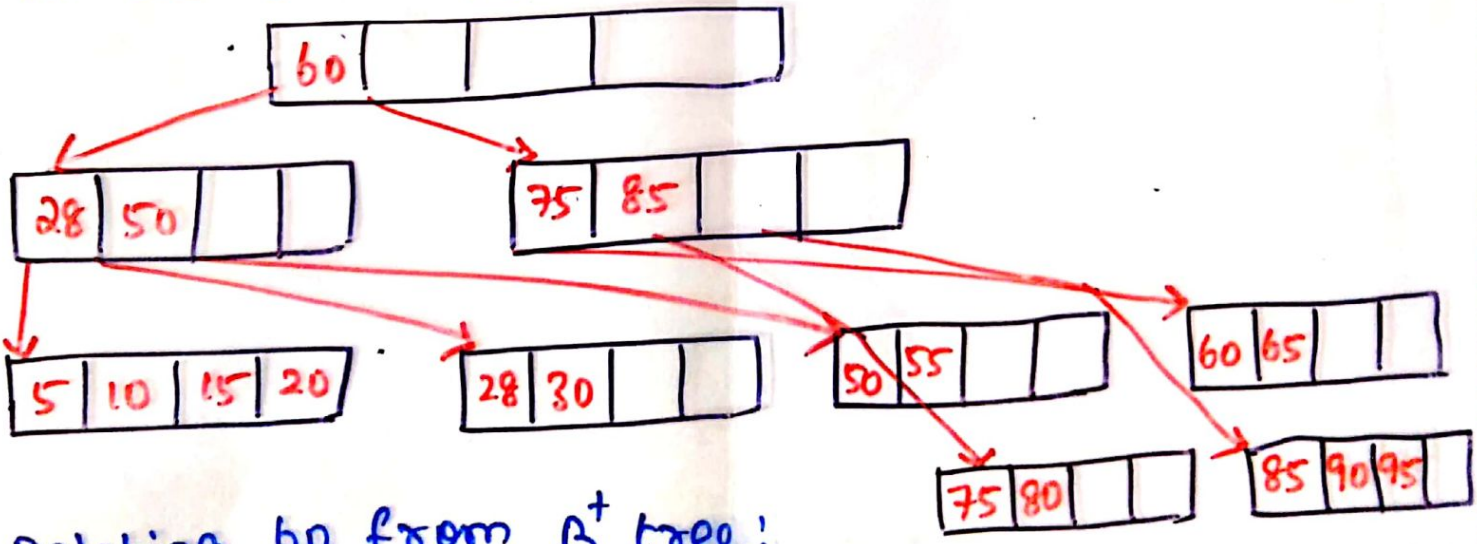
fill factor - To control the growth and the shrinkage. A 50% fill factor would be minimum for any B^+ tree.

Delete 70 from the B^+ tree! This record is in a leaf node containing 60, 65 and 70. This node will contain 2 records after the deletion. (fill factor is 50% or 2 records).

So, simply delete 70 from the leaf node.



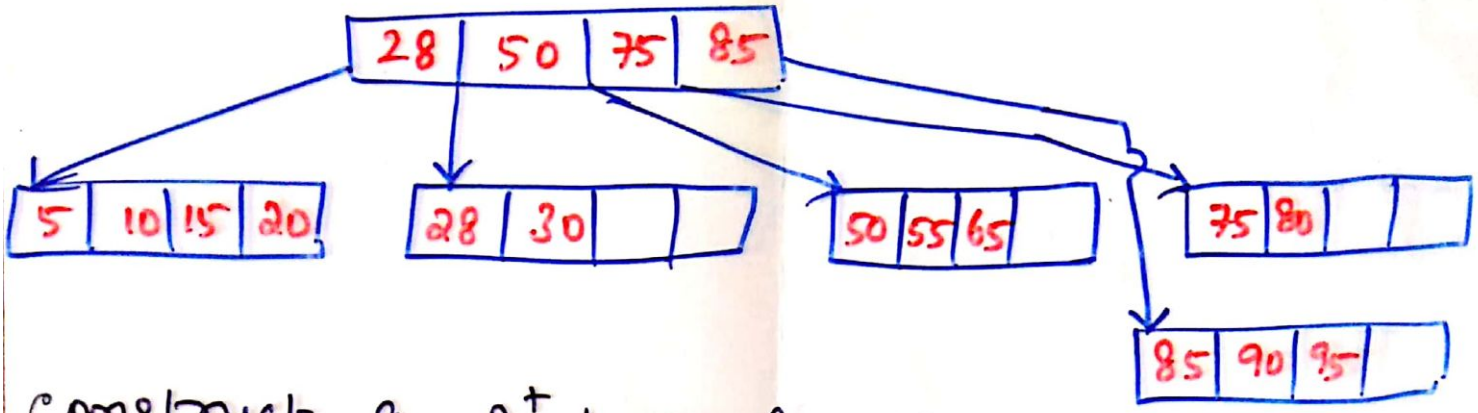
Delete 25 from B^+ tree! This record is found in the leaf node containing 25, 28, 30. The fill factor will be 50 %. (or) 2 records after the deletion. But 25, appear in the index node. Thus, when delete 25, it must replace with 28 in the index node.



Deleting 60 from B^+ tree!

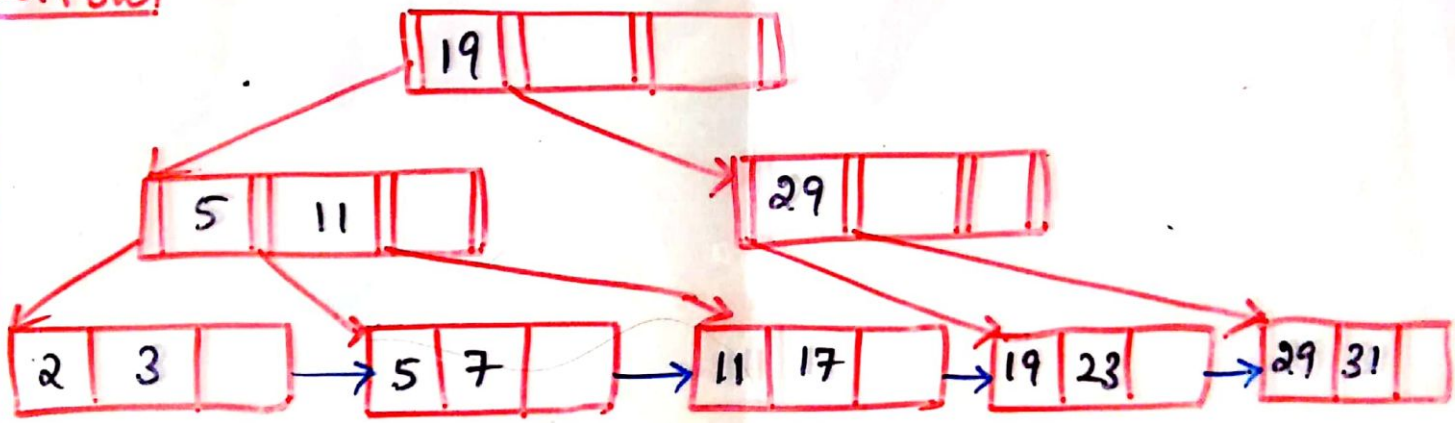
1. The leaf node containing 60, 65 will be below the fill factor after the deletion. So, Combine leaf nodes.
2. With recombined nodes, the index node will be reduced by one key. Hence it will also fall below the fill factor. So, Combine index nodes.
3. 60 appear in root node, so it will be removed.

After deletion of 60,

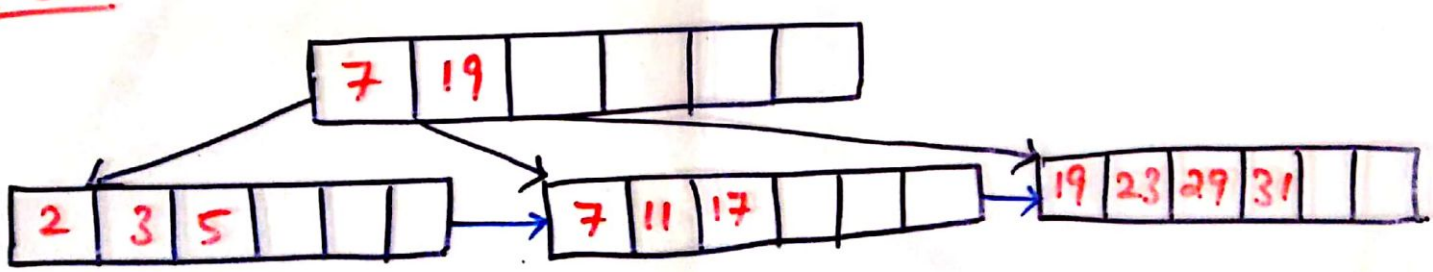


Construct a B^+ tree for the following set of key values. (2, 3, 5, 7, 11, 17, 19, 23, 29, 31). Assume that the tree is initially empty and values are added in ascending order. Construct B^+ trees for the cases where the number of pointers that will fit in one node as follows.
a. four b. six c. eight

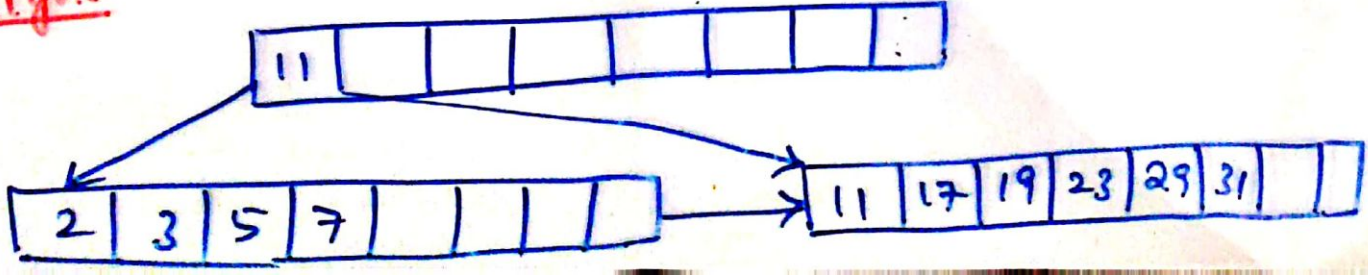
a. four



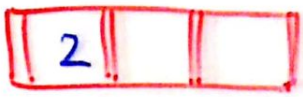
b. six



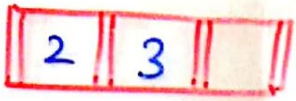
c. eight



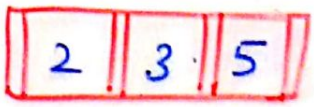
Step 1



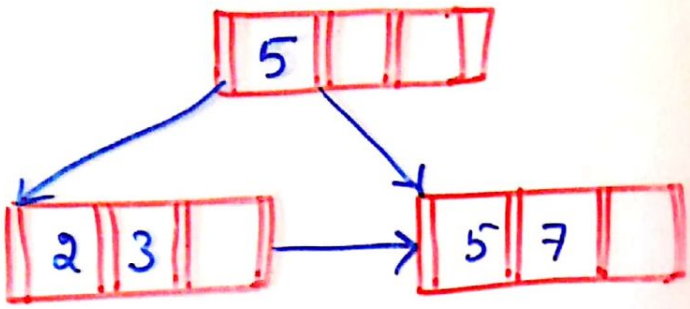
Step 2



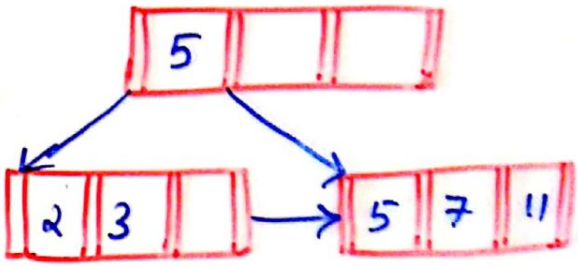
Step 3



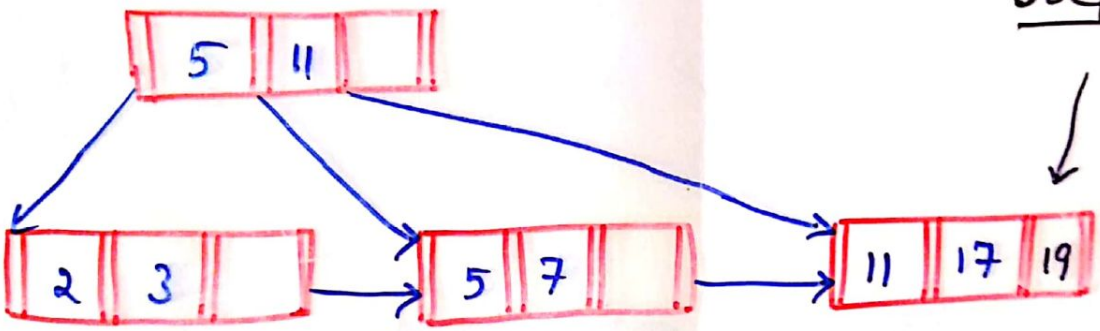
Step 4



Step 5

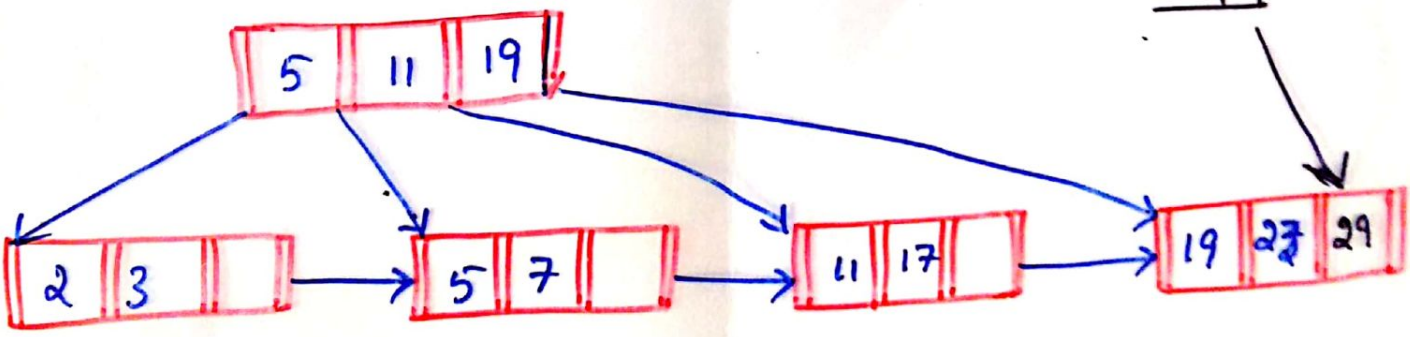


Step 6



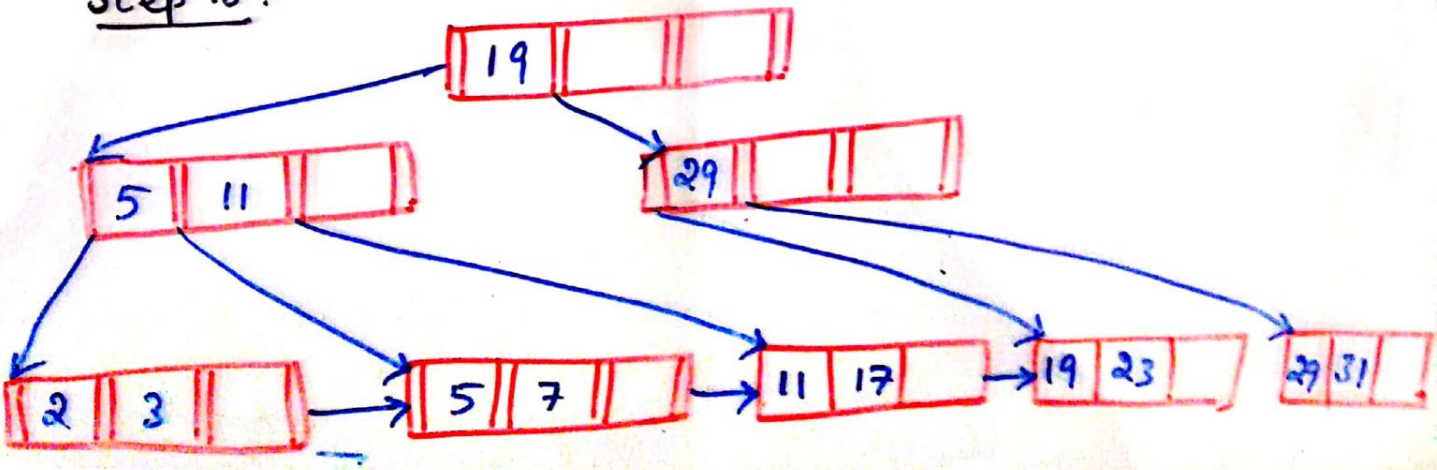
Step 7

Step 8 !



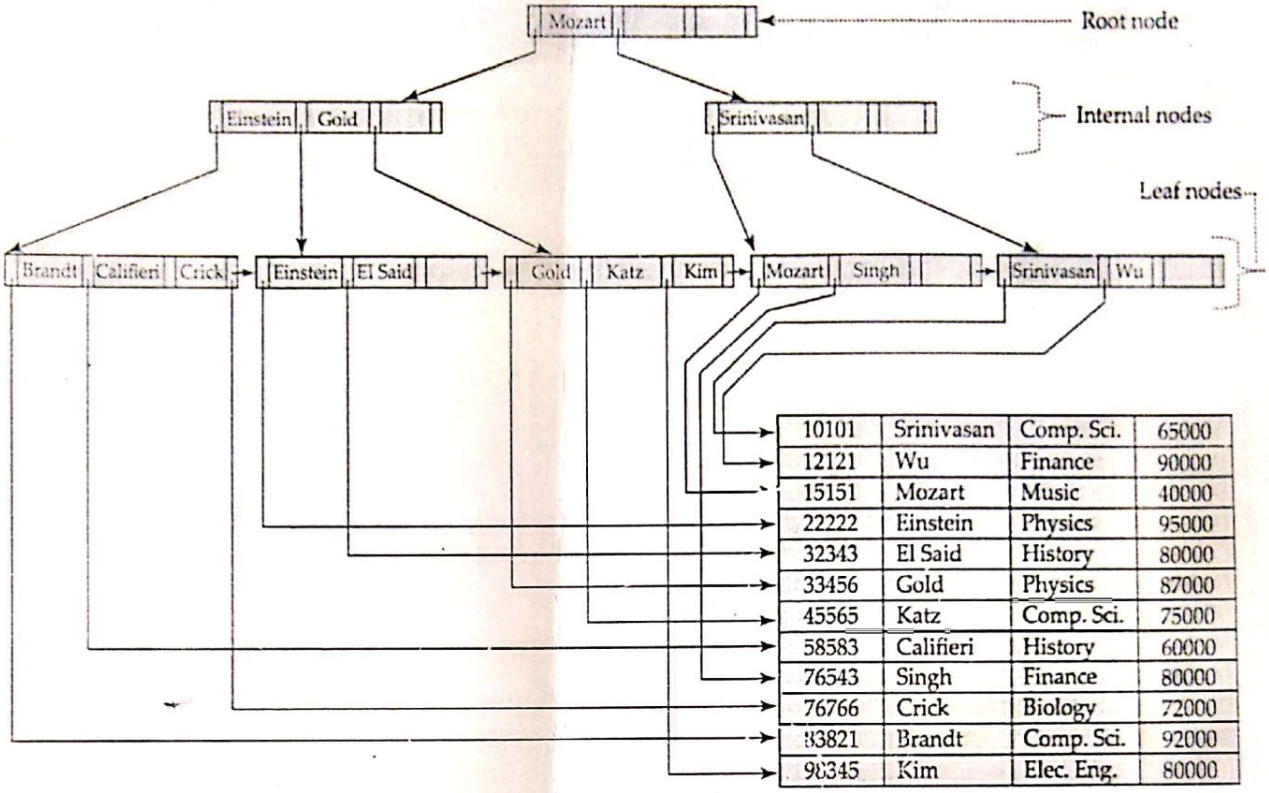
Step 9

Step 10 !



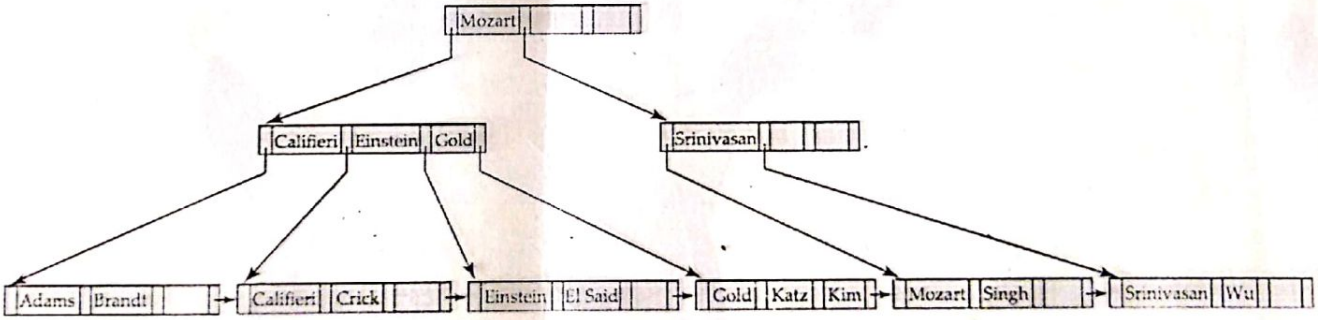
(1,4,5,9,10,11,12,13,18,20,29,30,38.40,41,45,50,60,70)

if n=4 i) Insert 27 ii) Delete 5,9,12,4,11

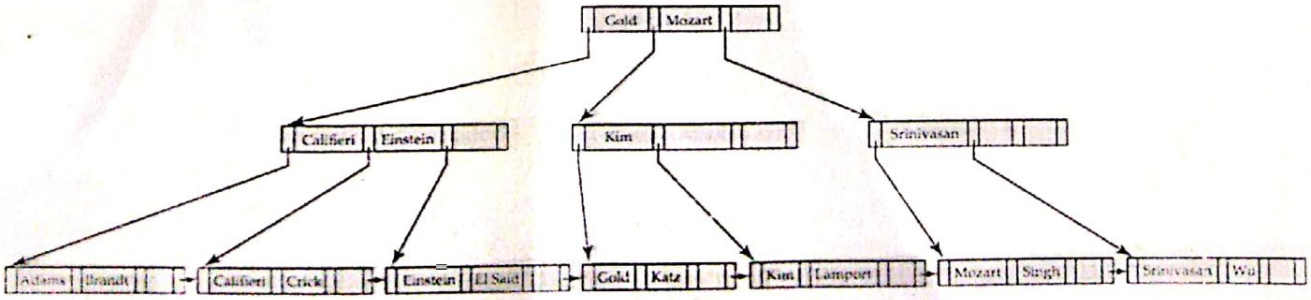


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

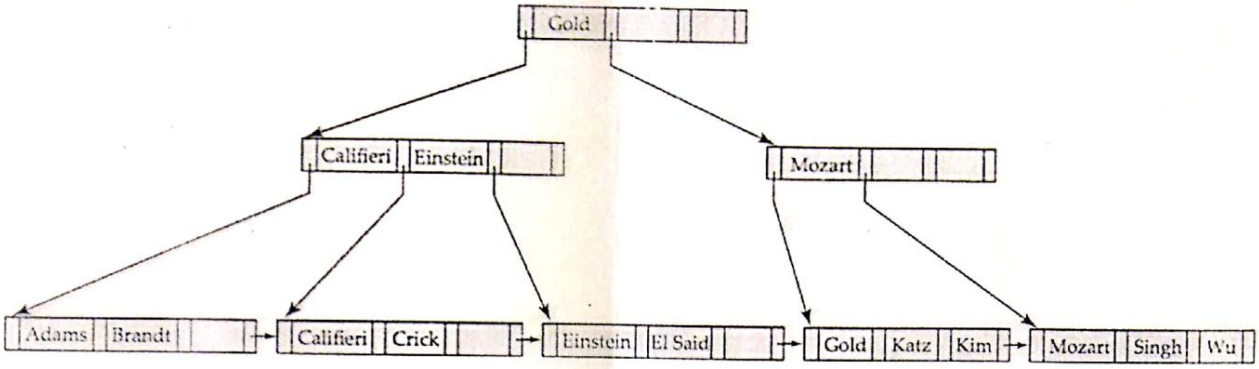
After insertion of "Adams"



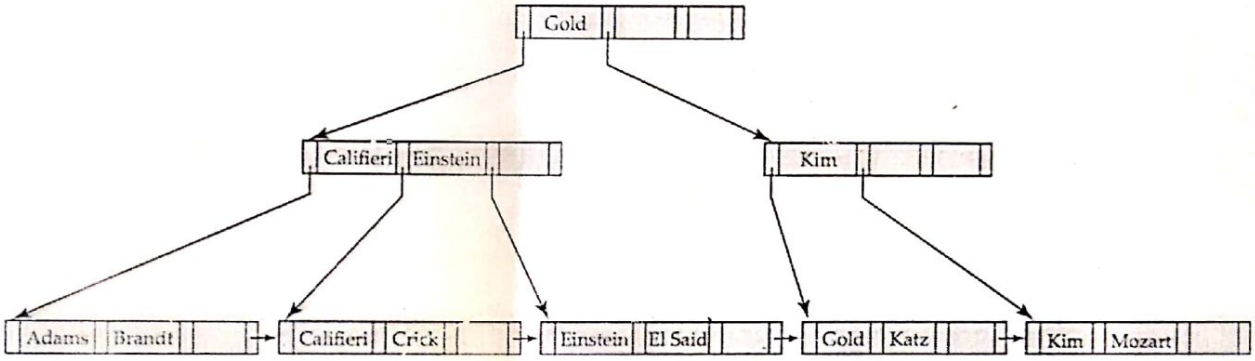
After insertion of "Lamport



Deleting "Srinivasan



Deletion of "Singh" and "Wu



Construct a B+-tree for the following set of key values:(2, 3, 5, 7, 11, 17, 19, 23, 29, 31).Assume that the tree is initially empty and values are added in ascending order. Construct B+-trees for the cases where the number of pointers that will fit in one node is as follows: **a.** Four **b.** Six **c.** Eight Show the form of the tree after each of the following series of operations:

a. Insert 9. **b.** Insert 10. **c.** Insert 8. **d.** Delete 23. **e.** Delete 19.

Transactions

A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

E.g. transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

Two main issues to deal with:

i) Failures of various kinds, such as hardware failures and system crashes ii) Concurrent execution of multiple transactions

Atomicity requirement

- If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state.
- Failure could be due to software or hardware.
- The system should ensure that updates of a partially executed transaction are not reflected in the database.

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist (exist) even if there are software or hardware failures.

Consistency requirement: The sum of A and B is unchanged by the execution of the transaction.

In general, consistency requirements include

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints (eg)sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

✓ A transaction must see a consistent database.

- ✓ During transaction execution the database may be temporarily inconsistent.
- ✓ When the transaction completes successfully the database must be consistent
- ✓ Erroneous transaction logic can lead to inconsistency

Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

	T1	T2
1.	**read**(A)	
2.	$A := A - 50$	
3.	**write**(A)	
		read(A), read(B), print(A+B)
4.	**read**(B)	
5.	$B := B + 50$	
6.	**write**(B)	

Isolation can be ensured trivially by running transactions **serially** that is, one after the other.

ACID Properties

A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

Atomicity: Either all operations of the transaction are properly reflected in the database or none are.

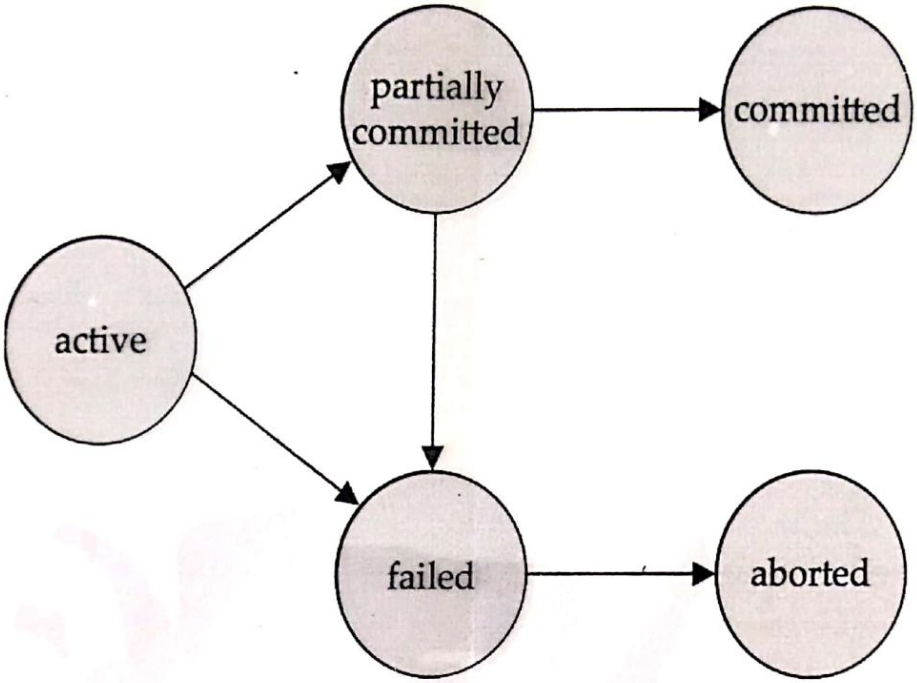
Consistency: Execution of a transaction in isolation preserves the consistency of the database. [(ie) the sum of A & B is unchanged]

Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction States



Active - the initial state; the transaction stays in this state while it is executing

Partially committed - after the final statement has been executed.

Failed - after the discovery that normal execution can no longer proceed.

Aborted – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:

i) restart the transaction - can be done only if no internal logical error

ii)kill the transaction

Committed – after successful completion.

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are:

Increased processor and disk utilization, leading to better transaction *throughput*(E.g) one transaction can be using the CPU while another is reading from or writing to the disk

Reduced average response time for transactions: short transactions need not wait behind long ones.

Concurrency control schemes – mechanisms to achieve isolation that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

Schedule – A sequences of instructions that specify the chronological (sequential) order in which instructions of concurrent transactions are executed

- a schedule for a set of transactions must consist of all instructions of those transactions
- must preserve the order in which the instructions appear in each individual transaction.

A transaction that successfully completes its execution will have a commit instruction as the last statement. By default transaction assumed to execute commit instruction as its last step

A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1:

Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.

A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

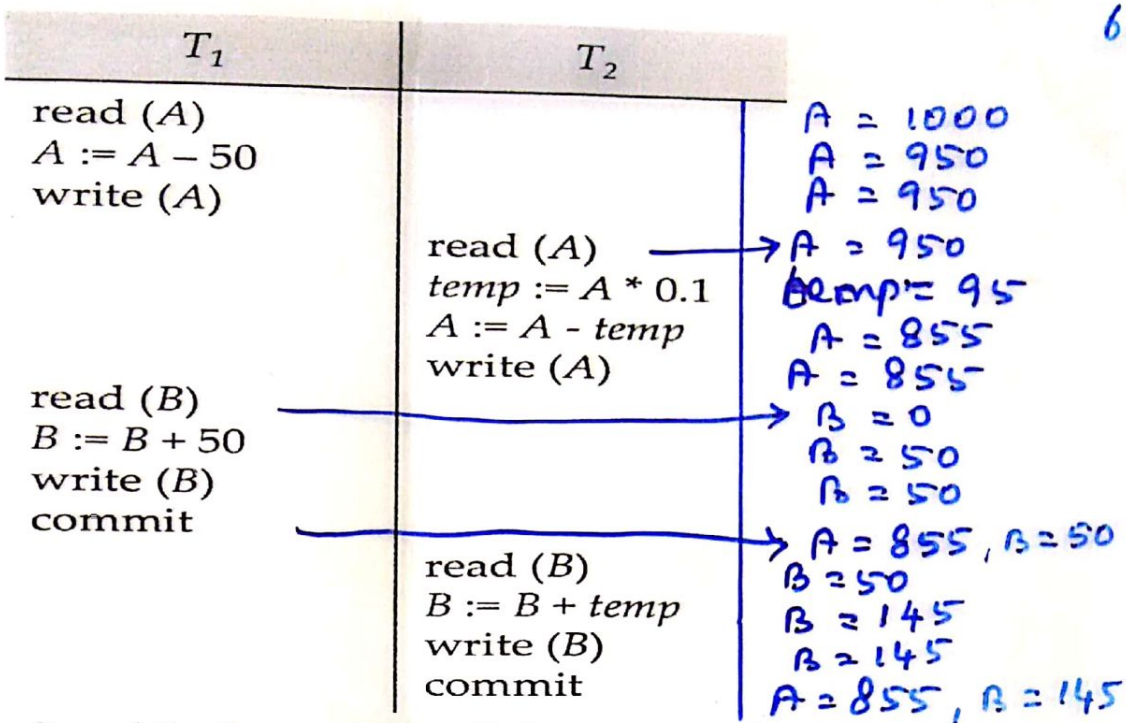
A = 1000
A = 950
A = 950
B = 0
B = 50
B = 50
A = 950, B = 50
A = 950
temp = 95
A = 855
A = 855
B = 50
B = 145
B = 145
A = 855 B = 145

Schedule 2 : A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

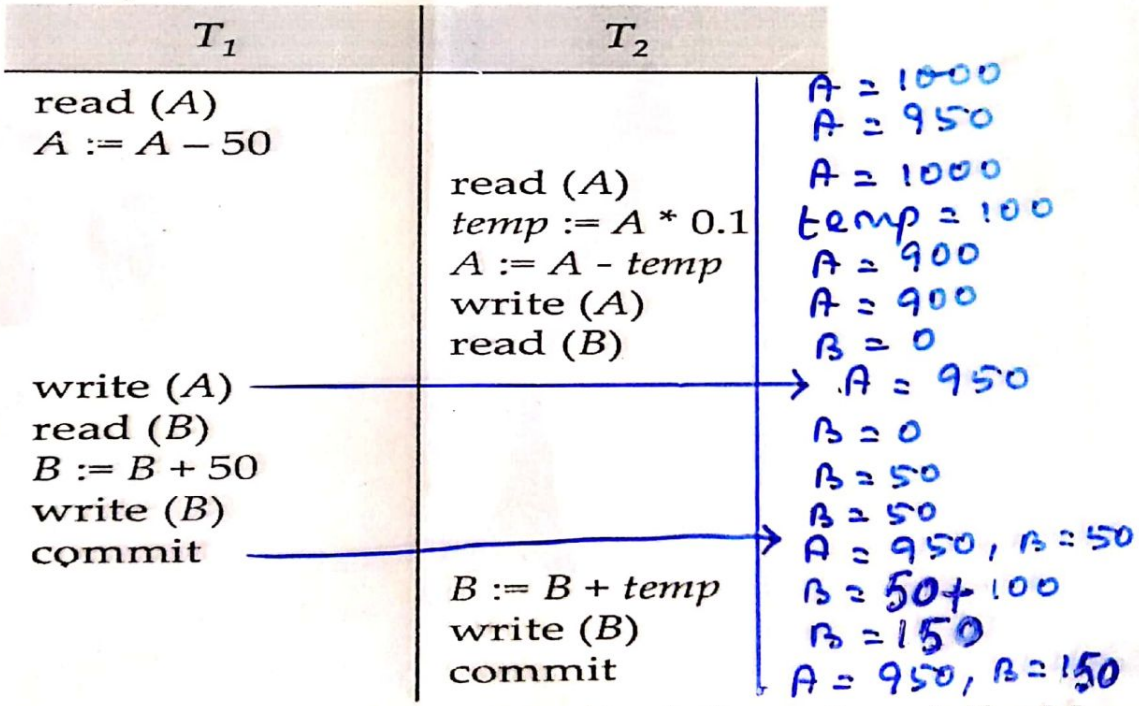
A = 1000
temp = 100
A = 900
A = 900
B = 0
B = 100
B = 100
A = 900, B = 100
A = 900
A = 850
A = 850
B = 100
B = 150
B = 150
A = 850, B = 150

Schedule 3: Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.



In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4: The following concurrent schedule does not preserve the value of $(A + B)$.



Distinction between the terms serial schedule and serializable schedule: A schedule in which all the instructions belonging to one single transaction appear together is called a ***serial schedule***. A ***serializable schedule*** has a weaker restriction that it should be *equivalent* to some serial schedule. There are two definitions of schedule equivalence – conflict equivalence and view equivalence.

* The given interleaved execution of these instructions is said to be serializable if it produces the same result as some serial execution of the transaction.

Serializability

• A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. Conflict serializability
2. View serializability

Conflicting Instructions

• Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

• Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.

• If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability: If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent. We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule

• Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Schedule 5 – swapping a pair of instructions from Schedule 3

T1	T2
read (A)	
write (A)	
	read (A)
read (B)	
	write (A)
write (B)	read (B), write (B)

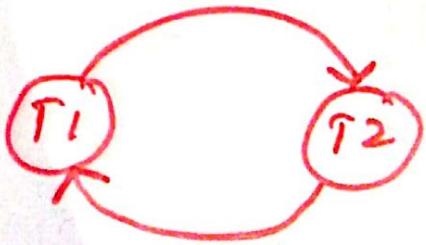
Problem1: Is the following schedule is conflict serializable?

T1	T2	T3
Read (A)		
$A = f_1(A)$		
Write (A)		
	Read (A)	
	$A = f_2(A)$	
	Write (A)	
	Read (B)	
	$B = f_3(B)$	
	Write (B)	
		Read (B)
		$B = f_4(B)$
		Write (B)

As the graph is acyclic, the schedule is conflict serializable

Problem2: Is the following schedule is conflict seri?

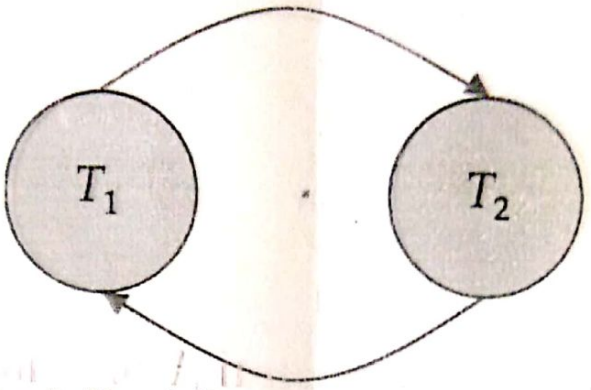
T1	T2
read (A)	
A = A - 50;	
	read (A)
	temp = A * 0.1
	A = A - temp;
	write (A)
	Read (B)
write (A)	
read (B)	
B = B + 50;	
write (B);	
	B = B + temp;
	write (B);



cycle exists, so the schedule is not conflict serializable.

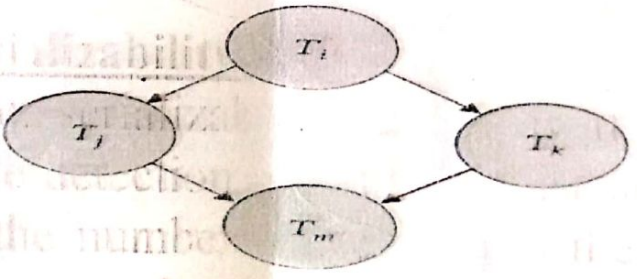
We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
We may label the arc by the item that was accessed.

Example 1

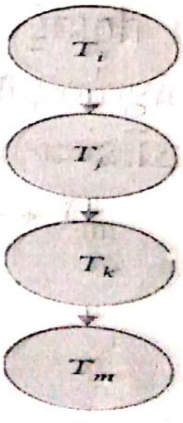


Test for Conflict Serializability:

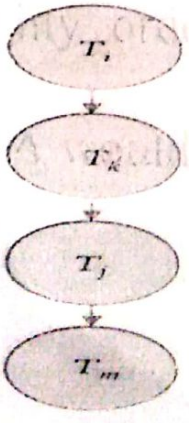
A schedule is conflict serializable if and only if its precedence graph is acyclic. Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)
If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
For example, a serializability order for Schedule A would be
$T_i \rightarrow T_j \rightarrow T_k \rightarrow T_m$



(a)



(b)



(c)

The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds.

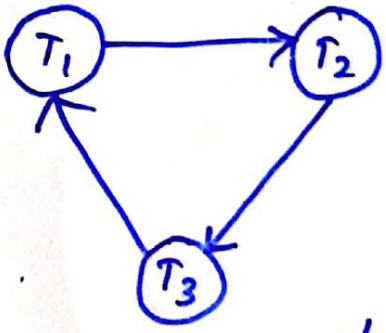
1. T_i executes write(Q) before T_j executes read(Q)
2. T_i executes read(Q) before T_j executes write(Q)
3. T_i executes write(Q) before T_j executes write(Q)

* If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles, then the schedule S is conflict serializable.

* A serializability order of the transactions can be obtained by topological sorting.

Problem: Consider following schedule. Is the given schedule conflict serializable?

T1	T2	T3
Read(A)		
	Read(B)	
$A = f_1(A)$		
		Read(C)
	$B = f_2(B)$	
	write(B)	
		$C = f_3(C)$
		write(C)
write(A)		
		Read(B)
	Read(A)	
	$A = f_4(A)$	
Read(C)		
	write(A)	
$C = f_4(C)$		
write(C)		
		$B = f_5(B)$, write(B)

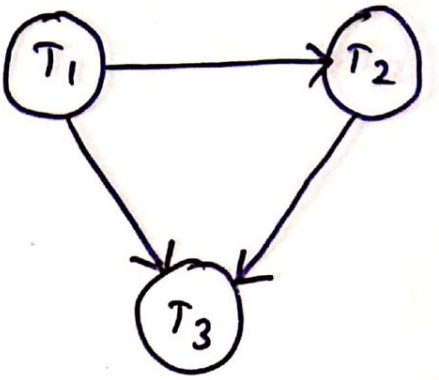


graph has cycle, so the schedule is not conflict serializable.

Problem: Is the following schedule conflict serializable?

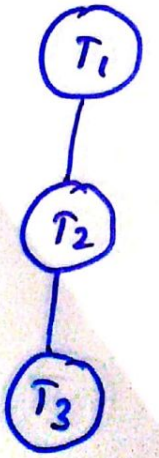
T_1	T_2	T_3
Read (A)		
$A = f_1(A)$		
Read (C)		
Write (A)		
	$T1 \rightarrow T2$	
$C = f_2(C)$		
	Read (B)	
Write (C)		
	Read (A)	
		Read (C)
	$B = f_3(B)$	
	Write (B)	
		$C = f_4(C)$
	$T2 \rightarrow T3$	Read (B)
		Write (C)
$T_1 \rightarrow T_3$	$A = f_5(A)$	
	Write (A)	
		$B = f_6(B)$
		Write (B)

The precedence graph is



The graph has no cycle, so the given schedule is conflict serializable.

To find the conflict equivalent serial schedule, apply topological sorting to the precedence graph.



Schedule 3

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 6

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

View Serializability : Let S and S' be two schedules with the same set of transactions. S and S' are view equivalent if the following three conditions are met, for each data item Q,

1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q. (Initial Read)
2. If in schedule S transaction T_i executes read(Q), and that value was produced by transaction T_j (if any), then in

1) Initial Read : If a transaction T_1 reading data item A from initial database S_1, then in S_2 also T_1 should read A from initial database.

S1

T_1	T_2	T_3
	R(A)	
W(A)		
		R(A)
	R(B)	

S2

T_1	T_2	T_3
		R(A)
	R(A)	
W(A)		
	R(B)	

2) Updated Read [Write Read] - If T_i is reading A which is updated by T_j in S_1 then in S_2 also T_i should read A which is updated by T_j

S1

T_1	T_2	T_3
W(A)		
	W(A)	
		R(A)

S2

T_1	T_2	T_3
W(A)		
		R(A)
	W(A)	

3) final Write : If a transaction T_1 updated A at last in S_1, then in S_2 also T_1 should perform final write operations.

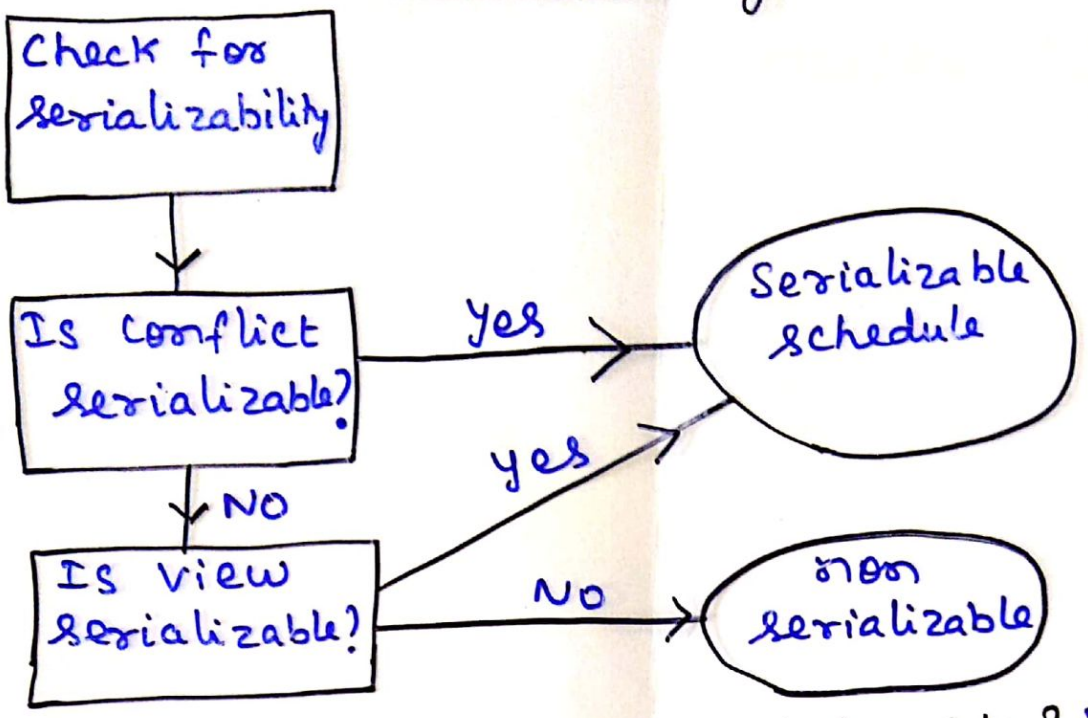
S_1

T_1	T_2
R(A)	
	W(A)
W(A)	

S2

T_1	T_2
R(A)	
W(A)	
	W(A)

Serializability



How to check view serializable? The schedule should satisfy three conditions.

1. Initial Read 2. Write Read (Update) 3. final write

I. Check the scdule is conflict (or) view serializable?

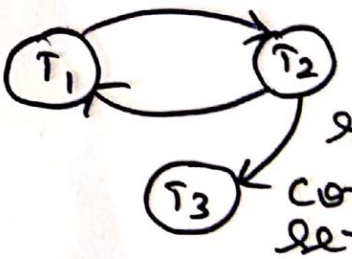
S : $R_2(B)$ $W_2(A)$ $R_1(A)$ $R_3(A)$ $W_1(B)$ $W_2(B)$ $W_3(B)$

$W_2(A)$ $R_1(A)$: $T_2 \rightarrow T_1$

$W_1(B)$ $W_2(B)$: $T_1 \rightarrow T_2$

$W_2(B)$ $W_3(B)$: $T_2 \rightarrow T_3$

conflict serializable!



Cycle exist so, it is not conflict serializable.

View serializable

	A	B
Initial Read	×	T_2
Write Read	T_2, T_1, T_3	×
final write	T_2	T_3

∴ the dependency is T_2 execute before T_1, T_3

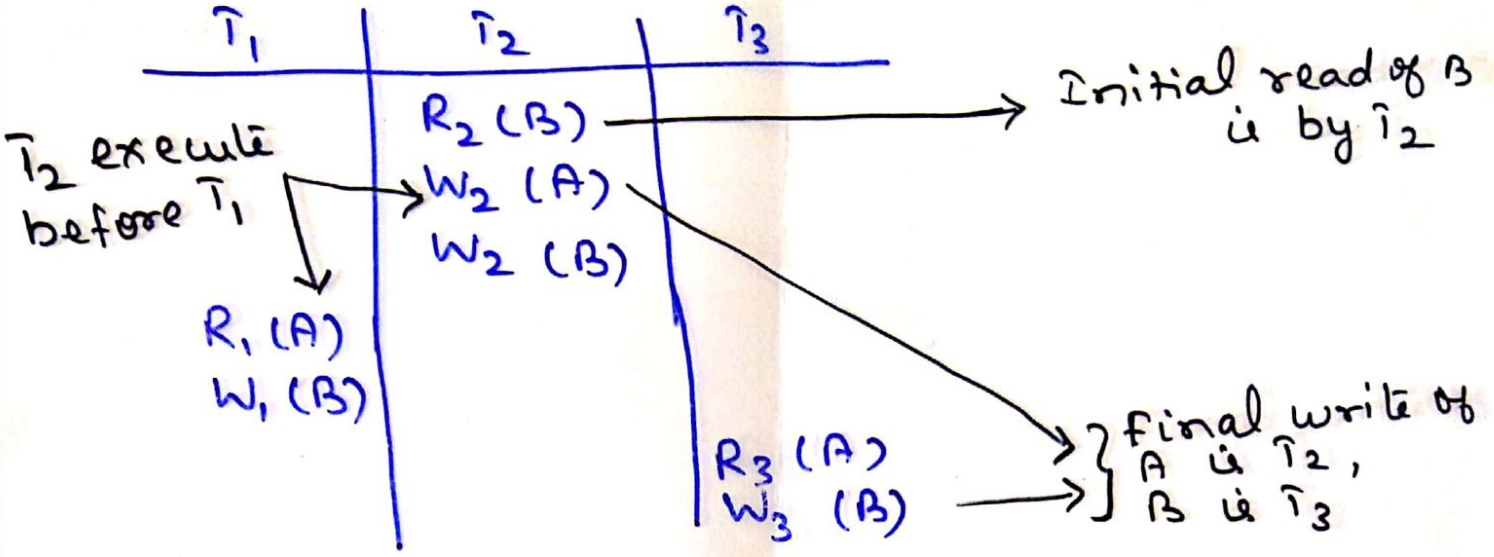
$T_2 \rightarrow T_1, T_3$

Based on A, $T_2 \rightarrow T_1, T_3$

Based on B, $T_2 \rightarrow T_3$

∴ The serial schedule is $T_2 \rightarrow T_1 \rightarrow T_3$

Write the Transactions [ie instructions] in a serial schedule,



II Check the following schedule is view serializable (or) not ?

S: $R_2(B)$ $R_2(A)$ $R_1(A)$ $R_3(A)$ $W_1(B)$ $W_2(B)$ $W_3(B)$

$T_2 \rightarrow T_1$, $T_1 \rightarrow T_2$, $T_2 \rightarrow T_3$

∴ It is not conflict serializable.

Check for view serializable

	A	B
Initial Read	T_2 T_1 T_3	T_2
Write Read	×	×
final Update	×	T_3

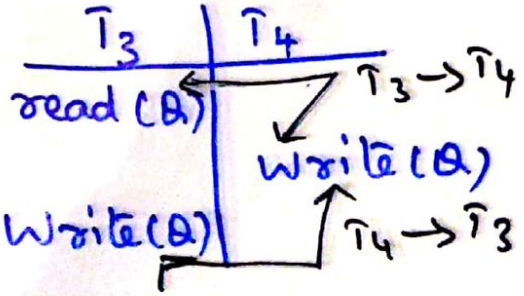
Based on A, T_2 read first.

Based on B, the dependency is T_2 execute before T_3.

So, it is view serializable. $T_2 \rightarrow T_3$

∴ the serial schedule is $T_2 \rightarrow T_1 \rightarrow T_3$

III Check the schedule is serializable (or) not?



∴ It is not conflict serializable.

check for view

	A
Initial Read	T_3
write Read	—
final write	T_3

Based on A, the dependency is $T_3 \rightarrow T_4$

∴ It is view serializable.

Shortcut to find view serializable:

Step 1. The schedule is not conflict serializable.
Step 2. The schedule contain blind writes.
[without performing read operations and directly perform write operations]

IV. Check the schedule is view serializable?

S: $R_1(A)$ $R_2(A)$ $R_3(A)$ $R_4(A)$ $R_4(B)$ $W_2(B)$ $W_2(B)$ $W_3(B)$ $W_4(B)$

$T_1 \to T_2$ $T_2 \to T_1$ $T_1 \to T_3$ $T_3 \to T_4$

Step1 : The schedule is not conflict serializable.
Step2 : $W_2(B)$, $W_3(B)$ and $W_4(B)$ are blind writes.
∴ The schedule is view serializable.

Initial Read →

	A	B
	T_1 T_2 T_3 T_4	T_1
Write Read	✗	✗
final write	✗	T_4

Based on A, there is no dependency.
Based on B, the dependency is T_1 execute before T_4
∴ The serial schedule is $< T_1\ T_2\ T_3\ T_4 >$ (or) $< T_1\ T_3\ T_2\ T_4 >$.

V. Check the schedule is serializable (or) not?

S: $R_2(B)$ $W_2(A)$ $R_1(A)$ $R_3(A)$ $W_1(B)$ $W_2(B)$ $W_3(B)$

$T_2 \to T_1$ $T_1 \to T_2$

The schedule is not conflict serializable and also contain blind writes [$W_1(B)$, $W_3(B)$] so it is view serializable.

	A	B
Initial Read	—	T_2
Write Read	$T_2 \to T_1$	—
Final Write	T_2	T_3

Based on A, the dependency is T_2 execute before T_1: $T_2 \to T_1$

Based on B, the dependency is T_2 execute before T_3

∴ The serial schedule is $< T_2, T_1, T_3 >$

VI Prove the following schedule is not view serializable. S: $R_1(A)\ R_2(A)\ R_3(B)\ W_1(A)\ R_2(C)\ R_2(B)\ W_2(B)\ W_3(C)$

	A	B	C
Initial Read	T_1	T_3	T_2
Write Read	—	—	—
final write	T_1	T_2	T_3

Based on data item A there is no dependency.

Based on B, the dependency is $T_3 \to T_2$

Based on C, the dependency is $T_2 \to T_3$

There is a conflict between B and C, so it is not view serializable.

Test for View Serializability: The precedence graph test for conflict serializability cannot be used directly to test for view serializability. Extension to test for view serializability has cost exponential in the size of the precedence graph. The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.

Recoverable Schedules: if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j. The following schedule is not recoverable if T_9 commits immediately after the read

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

Can lead to the undoing of a significant amount of work

Cascadeless schedules — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j. Every cascadeless schedule is also recoverable. It is desirable to restrict the schedules to those that are cascadeless.

Exercise problem 1: During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass.

1. *active→partially committed→committed*
2. *active → partially committed → aborted*
3. *active → failed → aborted*

Exercise problem 2: Justify the following statement: Concurrent execution of transactions is more important when data must be fetched from (slow) disk or when transactions are long, and is less important when data is in memory and transactions are very short.

Answer: If a transaction is very long or when it fetches data from a slow disk, it takes a long time to complete. In absence of concurrency, other transactions will have to wait for longer period of time. Average response time will increase. Also when the transaction is reading data from disk, CPU is idle. So resources are not properly utilized. Hence concurrent execution becomes important in this case. However, when the transactions are short or the data is available in memory, these problems do not occur.

Exercise problem 3: Consider the following two transactions:

$T1$: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).
$T2$: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$; write(A).

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

b. Show a concurrent execution of $T1$ and $T2$ that produces a nonserializable schedule.

a. There are two possible executions: $T1, T2$ and $T2, T1$.

Case 1:

	A	B
initially	0	0
after $T1$	0	1
after $T2$	0	1

Consistency met: $A = 0 \vee B = 0 \equiv T \vee F = T$

Case 2:

	A	B
initially	0	0
after $T2$	1	0
after $T1$	1	0

Consistency met: $A = 0 \vee B = 0 \equiv F \vee T = T$

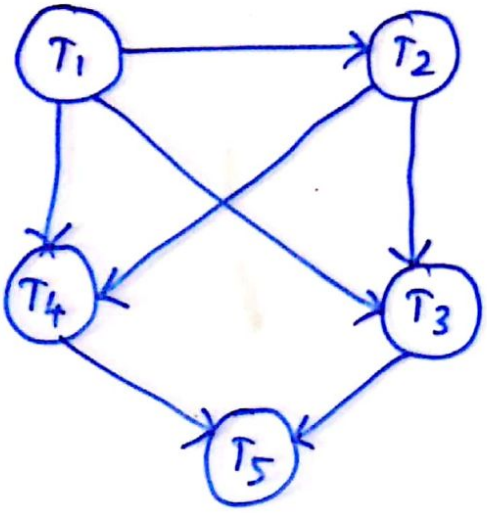
b. Any interleaving of $T1$ and $T2$ results in a non-serializable schedule.

T1	T2
read (A)	
	read (B)
	read (A)
read (B)	
if A = 0 then B = B + 1	
	if B = 0 then A = A + 1
Write (B)	
	Write (A)

Exercise problem 4: Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

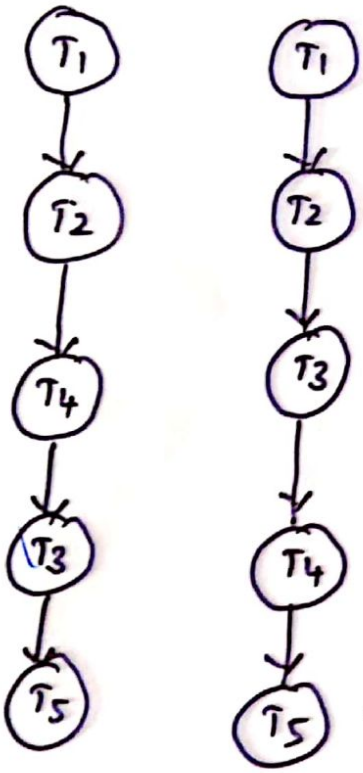
Answer: Most of the concurrency control protocols used in practise are based on conflict serializability—they actually permit only a subset of conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

Exercise problem 6! Is the corresponding schedule conflict serializable?



The graph has no cycles, so the schedule is conflict serializable.

The serializability order is obtained by applying topological sorting.



Topological sorting steps:

Step1: Initialize the serial schedule as empty

Step2: find a transaction T_i, such that no arcs entering T_i. T_j is the next transaction in the schedule

Step3: Remove T_i, and all edges emitting from T_i. If the remaining set is non empty, return to step2, else the serial schedule is complete.