Week-9.

Week 9.1

INDEXING: used to speed up access to desired data.

Search Key: attributes used to look up records in a file.

→ Index file are much smaller than original file.

1) ORDERED INDEX: Search keys are stored in sorted order.

2) HASH INDEX: Search key are distributed uniformally across buckets using a hash funct".

ORDERED Indices:

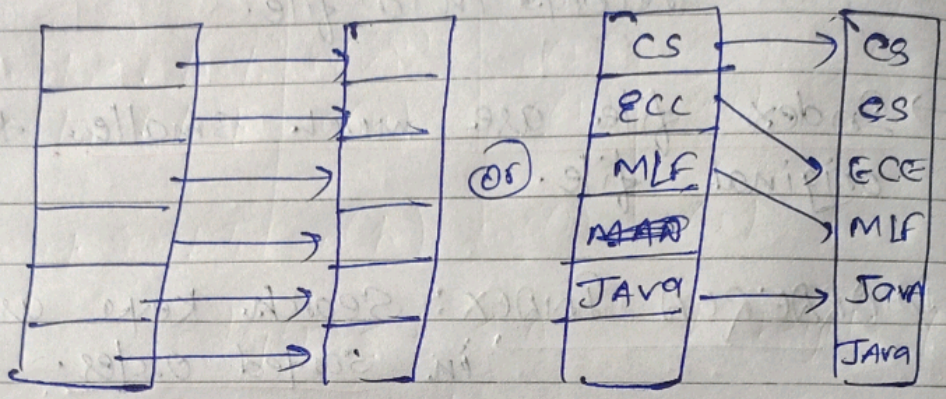
1) PRIMARY INDEX: index structure created on the primary key of database table.
   → also called clustering index.
   → values in primary key index is unique.

2) SECONDARY INDEX: index created on non-primary key in table.
   → values covered in 2ndry index may not be unique.
   → non-clustering index.

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

DENSE INDEX FILE

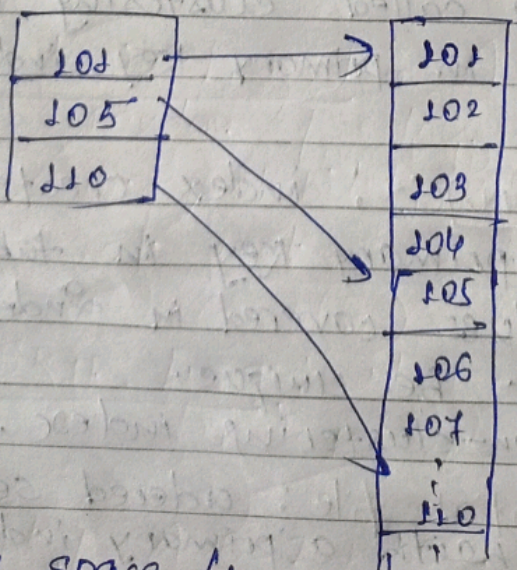
→ index record appears for every search key value in the file.



SPARSE INDER FILE

→ contain index records for only some search key-value.

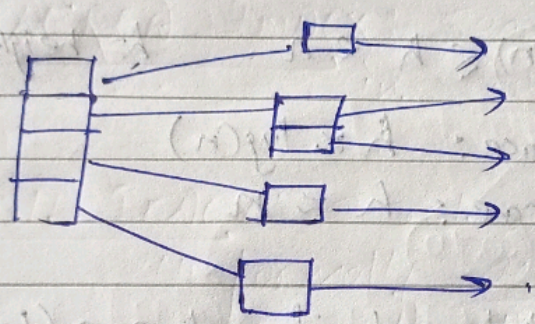
→ applicable when records are sequentially ordered on search-key.



→ less space / less maintainance.

* Sparse index consume less space & less maintainance over insert" delet" than dense index.
Dense index in generally faster than sparse index.

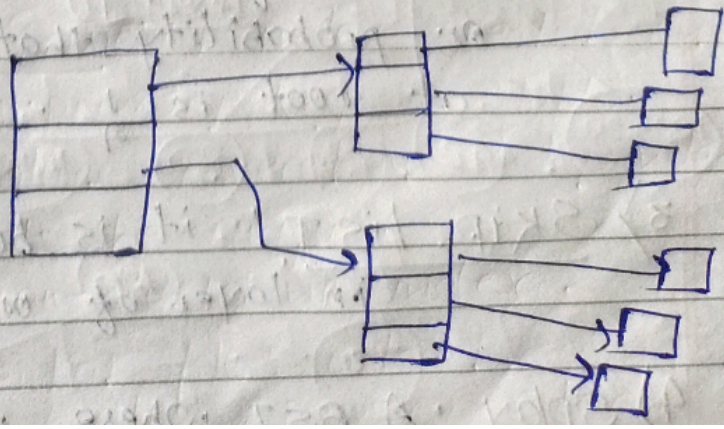
Secondary Index: index on index.



Multilevel indexing: if primary key doesn't fit in memory (large size) then access becomes expensive.

→ sol": create another index for for original index.

- outer index — sparse index of primary index.
- Inner index — primary index file.



Lec-9.2

Data structure	Search	Insert	Delete
Unordered Array	$O(n)$	$O(1)$	$O(1)$
Ordered Array	$O(\log n)$	$O(n)$	$O(n)$
Unordered List	$O(n)$	$O(1)$	$O(1)$
Ordered List	$O(n)$	$O(1)$	$O(1)$
Binary Search Tree	$O(h)$	$O(1)$	$O(1)$

$\log(n) \leq h < n$ $\quad$ h = height of tree

Best case: $h \approx \log(n)$

Worst case: $h \approx n$

→ A BST is balanced if $h \sim O(\log n)$

- Balancing Guarantees may be of various type:

1) ~~Roots~~ AVL TREE : A BST where height of two child sub-tree of any node differ by at most 1.

2) Randomized BST : A BST on n keys is random if either it is empty ($n=0$) or probability that a given key is at root is $\frac{1}{n}$.

3) Skip List : it is built (probabilistically) in layer of ordered linked list.

4) Splay : A BST where recently accessed element are quick to access again.

Lec-9:2

2-3-4 Trees

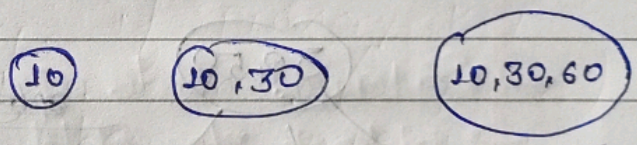
→ all leaves are at same depth (bottom level)
→ $h \sim O(\log n)$
→ complexity of search, insert and delete : $O(h) \sim O(\log n)$

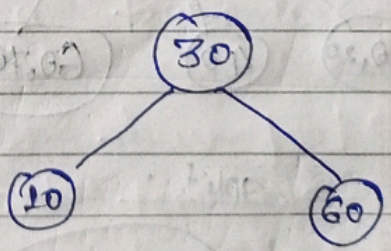
→ all data kept in sorted order.

order = 2
- 2-nodes have one key & 2 child
- 3-nodes (order = 3) have two key & 3-child
- 4-nodes (order = 4) have three key & 4-child

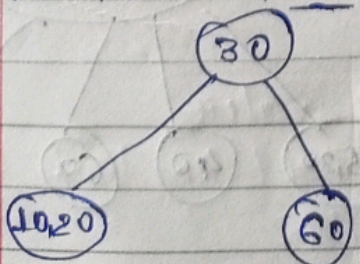
Ex Insert 10, 30, 60, 20, 50, 40, 70, 80, 15, 90, 100

- 10
- 10, 30
- 10, 30, 60
- ~~10, 30~~ split for 20

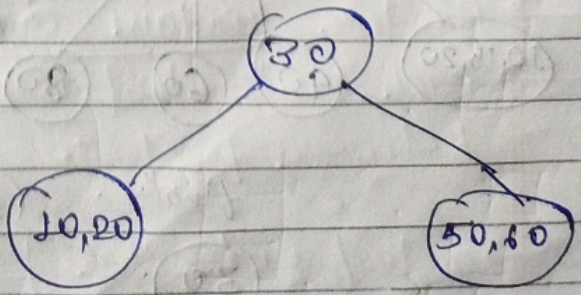




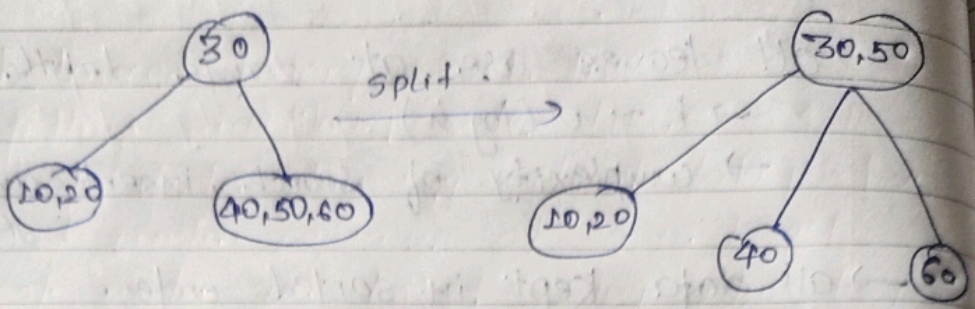
→ 10, 30, 60, 20



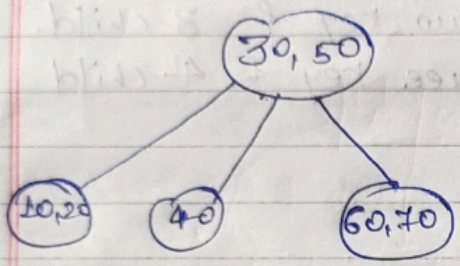
⇒ 10, 30, 60, 20, 50



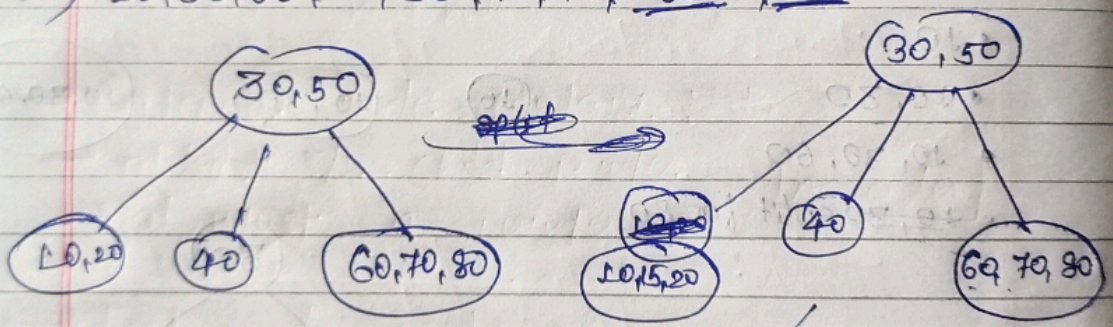
⇒ 10, 30, 60, 20, 50, 40



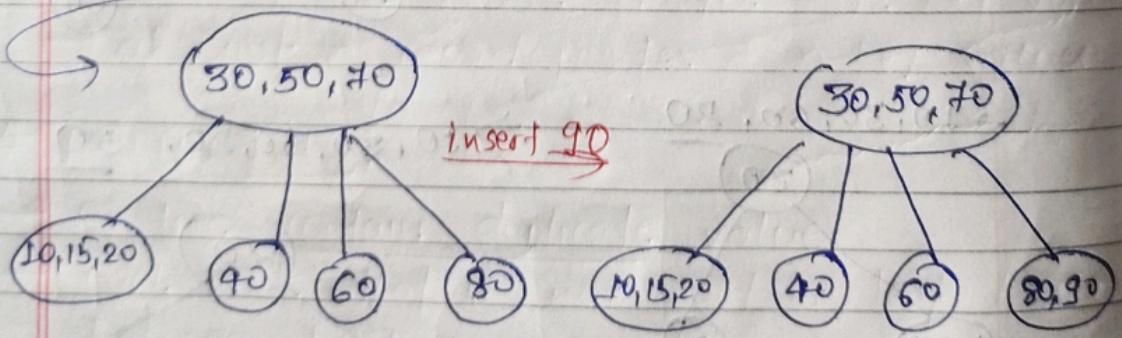
⇒ 10, 30, 60, 20, 50, 40, 70



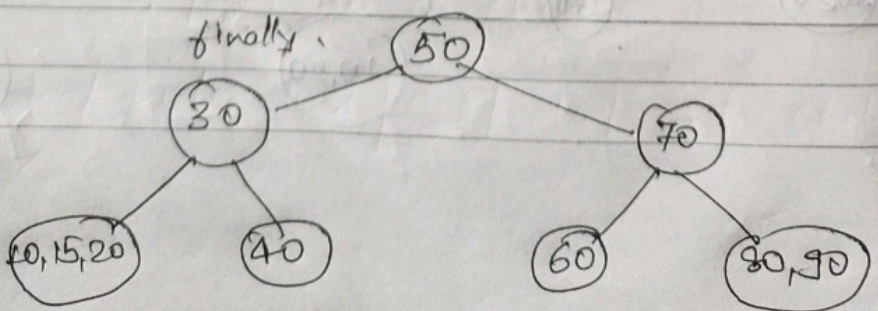
⇒ 10, 30, 60, 20, 50, 40, 70, 80, 15



split.



finally.



2-3-4 Tree

Advantage :-

→ all leaves are at same level. $h \sim o(\log n)$

→ complexity of search, insert and delete : $o(h) \sim o(\log n)$

→ all data kept in sorted order.

→ Generalization easily to large node

→ extend to external data structure.

Disadvantage :

→ complexity :

→ slower insert & deletion.

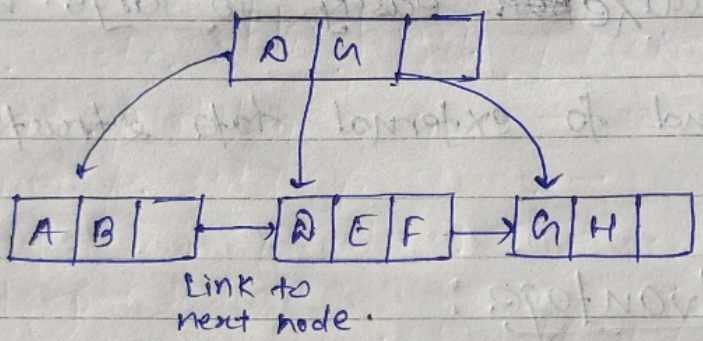
→ uses variety of node-type → need to destruct and construct multiple nodes for converting a 2-node to-3 and so on.

→ all path from root to leave are of same length

lec 9.3

B^+ Tree

→ balanced binary search Tree.
→ follow multilevel index like 2-3-4 tree
→ leaf node denoting actual data pointers.
→ leaf node remains at same height.
→ leaf node are linked using a link list.
→ support random access as well as sequential access.



→ Internal node contains:
→ at least $n/2$ child pointer
→ at most n pointer.

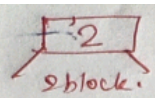
[n = order]

→ Leaf node contain:
→ at least $n/2$ record pointer & $n/2$ key values
→ at most n record pointer & n key values

PROPERTIES:

→ B^+ tree contain duplicate value
→ each node that is not root or leaf has between $\lceil \frac{n}{2} \rceil$ to n child.

→ a leaf node has $\lceil \frac{n-1}{2} \rceil$ to $\lceil n-1 \rceil$ child



Search key

block pointer point to child of tree
Record pointer point to actual data.

→ if root is not leaf it has atleast 2 child
→ if root is leaf, it can have [0 to (n-1)] child

→ B^+ tree contain a relatively small no. of
  → level below root has at least $2 \cdot \lceil \frac{n}{2} \rceil$ levels.
  → next level has $2 \cdot \lceil \frac{n}{2} \rceil \cdot \lceil \frac{n}{2} \rceil$ value
  ⋮
→ if these are 'K' search value in file, the tree height is NOT more than $\lceil \log_{n/2} K \rceil$

→ insert" & delet" can be handled efficiently

B - Tree	B^+ Tree
→ no linked list	→ linked list
→ no duplicate	→ duplicate (only in leaf)
→ record pointer present	→ No record pointer.
→ slow searching	→ fast searching
→ insert" & delet" complex	→ easy
→ imple mentat" hard	→ easy
→ use less node than B^+	→ use more node than B
→ B have greater depth than B^+	

Q) Consider B^+ tree with 4 million search key with order ($n = 80$). max no. of nodes to be accessed in lookup operation:

$$\log_{80/2} 4\text{million} = \log_{40} 4000{,}000 = 5$$

lec 9.4

Hashing

→ Static Hashing
→ Dynamic Hashing
→ comparison of ordered Indexing & Hashing
→ Bitmap Indices

→ A hash function maps data of arbitary size to fix size value.

→ key = K $h(K)$ = hash value / hash

→ if two keys $K_1 \neq K_2$, $h(K_1) = h(K_2)$ then collison occurs.

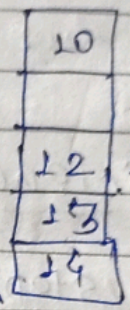
→ A hash function should be collision free and Fast.

Static Hashing.

→ BUCKETS :- stores records.

→ Hash funct" used to locate records for access, insertion, delet".

ex: locate: 10, 12, 14, 13
h(x) = x mod 10



→ Worst hash funct" maps all key to some bucket. this makes access time O(n)

→ ideal hash funct" is uniform. i.e each key goes to each bucket.

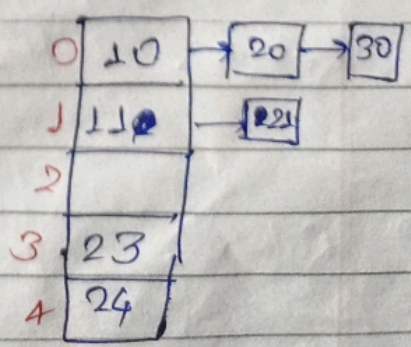
Bucket overflow occurs due to:

→ insufficient buckets.
→ multiple records have same bucket value.

Overflow of bucket can be reduce:

i) overflow chaining: (close hashing)

ex: locate: 10, 20, 30, 11, 21, 23, 24.



Hash Indices

→ Hashing ~~co~~ used not only for file organisation, but also for index-structure creation.

→ hash indices are always secondary indices.

Deficiencies of Static Hashing

→ good for fix size of dataset

→ In static hashing, no. of buckets is fix.
  :→ if database grow → overflow occurs.
  :→ if database shrink → wastage of storage

One-solution: periodic re-organize of file with new hash fun" (expensive)

Bettersolution: allow to modify bucket size dynamically.

advantage: → fast performance not degrade with growth of file
→ minimal space overhead

disadvantage: → bucket adress become big
→ change bucket size is costly.

Dynamic / Extendable Hashing.

↳ variable no. of buckets.

→ go through ppt 9.4 example

Bitmap Indices

→ special type of index designed for efficient quering on multiple keys.

→ applicable on small table.

→ bitmap has as many bits as records.

	ID	gender	Level
0	1	m	L1
1	2	f	L2
2	3	f	L1
3	4	m	L4
4	5	f	L3

bitmap on gender

m: 10010

f: 01101

पहली row
उतनी no. of bits

bitmap for income

L1: 10100

L2: 01000

L3: 00001

L4: 00010

L5: 00000

Q find male with ~~income~~ level L1

m: 10010

and

L1: 10100

1 0 0 0 0 → intersect of m & L1

↳ 1st row is answer

lec 9.5

Index definat" is SQL.

① Create Index:

Create index <index_name> on <table_nae> (attribute list)

② Drop a index:

drop index <index name>

Guideline for indexing

→ Rule 0: Indexes lead to Access more index search is easy but update become slow

→ Rule 1: Index the correct table.

→ Rule 2: Index the correct column.

→ Rule 3: Limit no. of index for each table

→ Rule 4: Choose order of column in composite index

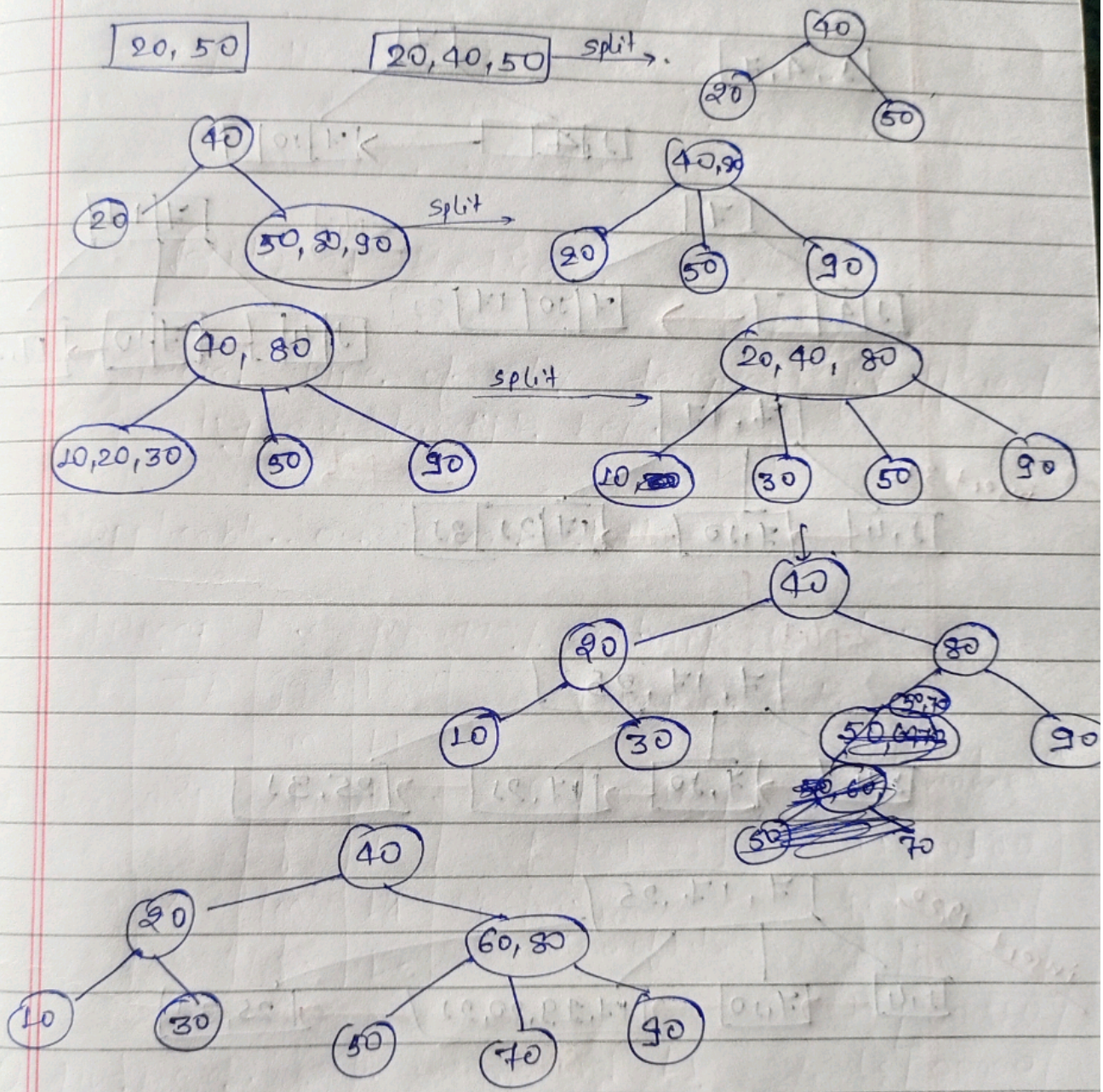
→ Rule 5: Gather statistic to make Index usage more accurate.

→ Rule 6: Drop Indexes that are no longer Require.

max 2 key
max child = 3

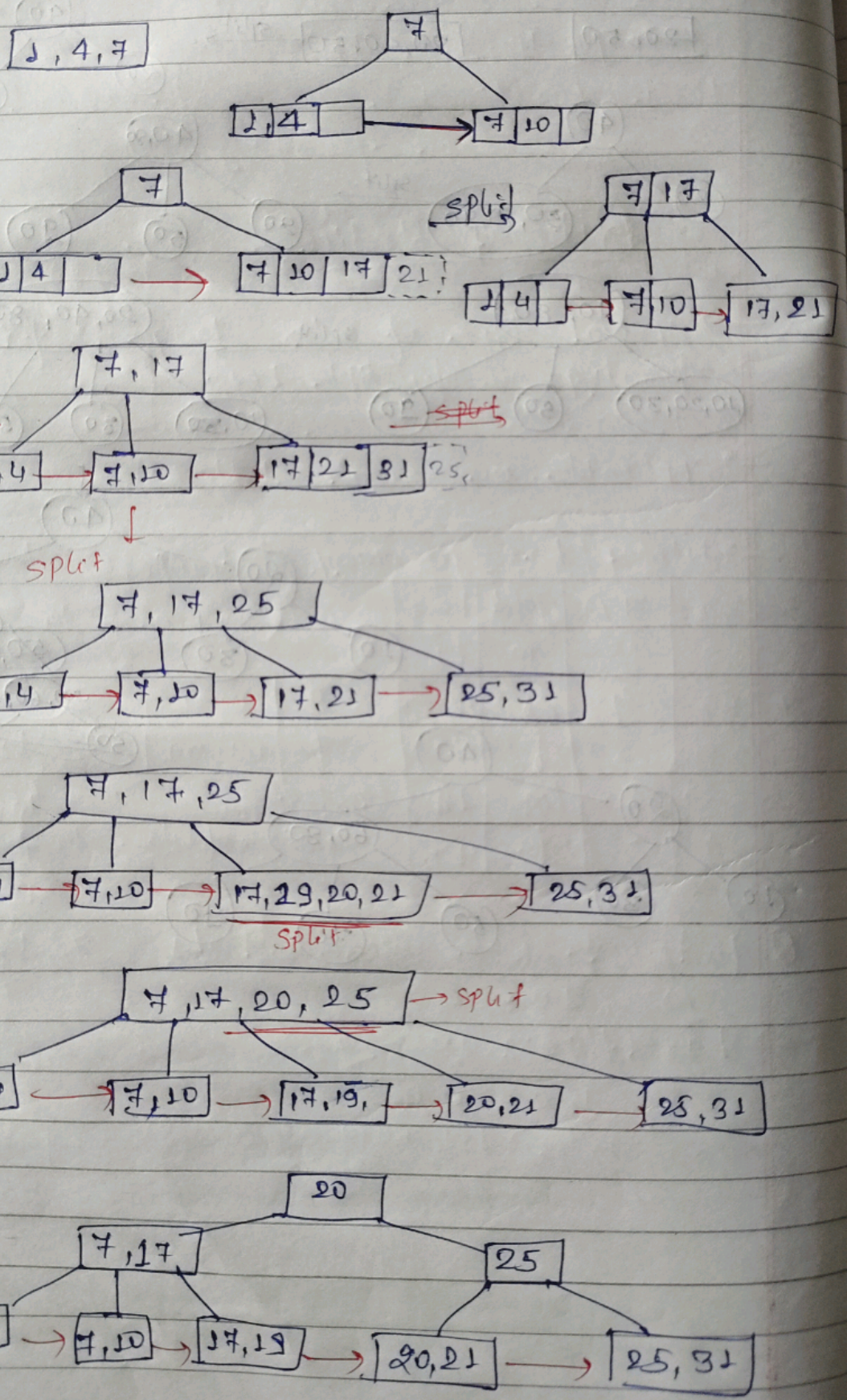
Construct 3-order B+ tree

20, 50, 40, 80, 90, 10, 30, 70, 60

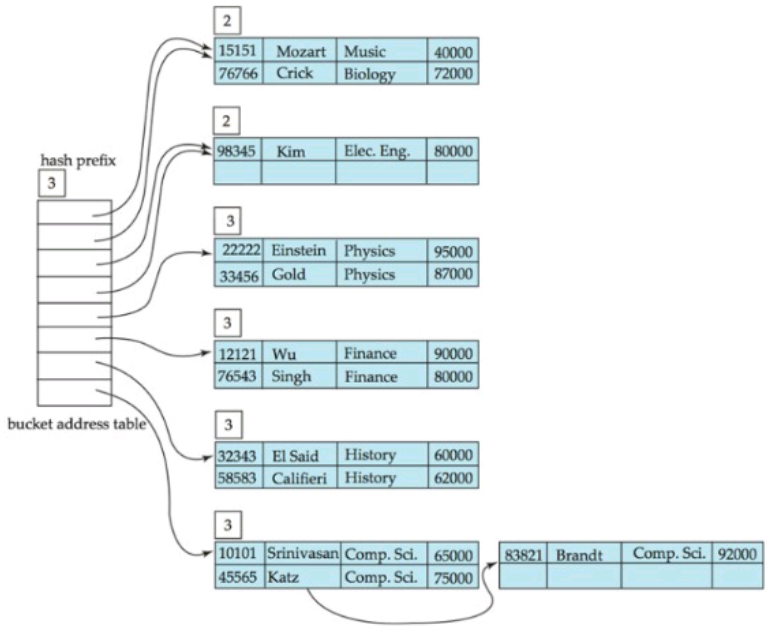


Q Construct 4-order B^+ tree (max key = 3, max child = 4)

1, 4, 7, 10, 17, 21, 31, 25, 19, 20



- Hash structure after insertion of "Kim" record

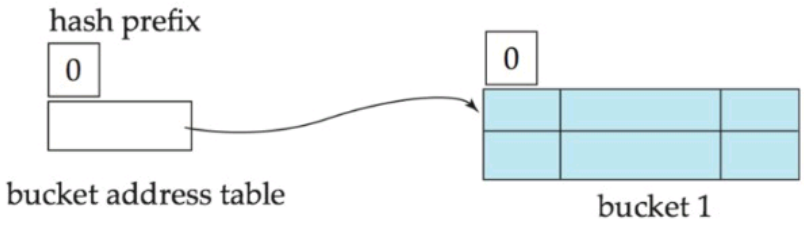


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

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

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

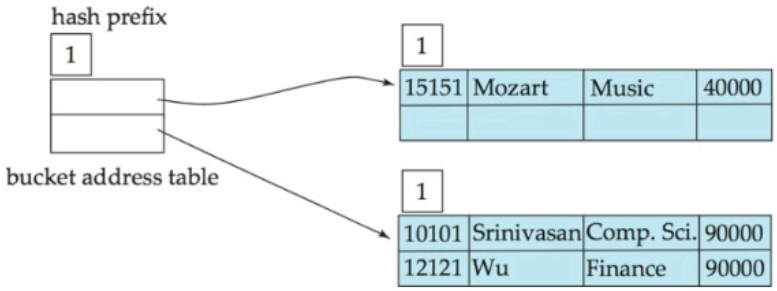
- Initial Hash structure; bucket size = 2



- Insert "Mozart", "Srinivasan", and "Wu" records

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

- Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records

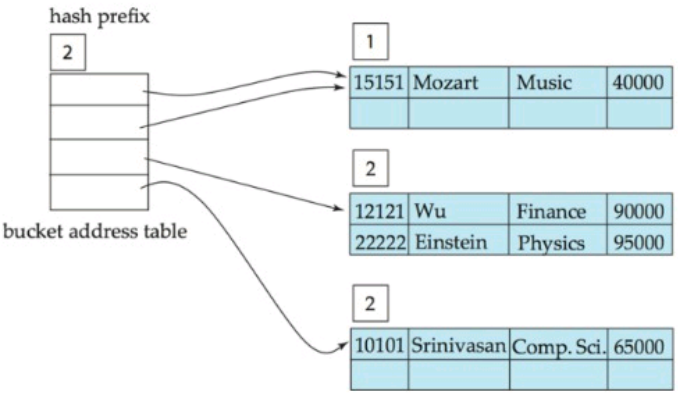


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

- Insert Einstein record

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

- Hash structure after insertion of "Einstein" record

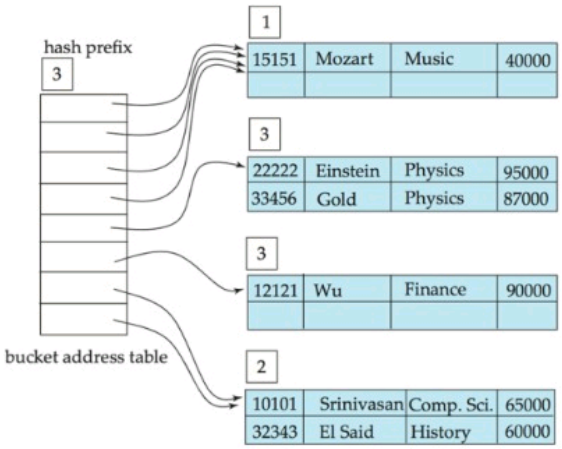


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

- Insert "Gold" and "El Said" records

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

- Hash structure after insertion of "Gold" and "El Said" records



dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

- Insert Katz record

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