Disk Storage and Indexing

Outline

Disk Storage Devices

- Files of Records
- Operations on Files
- Unordered Files
- Ordered Files

Indexing

Disk Storage Devices

Memory Hierarchies

- Primary Storage Level
  - Cache Memory
  - Main Memory
- Secondary Storage Level
  - Magnetic Disks
  - Mass Storage Device
    - CD-ROM
    - DVD
- Flash Memory (Tertiary Storage Level)
  - High Density, High Performance Using EEPROMs

Preferred secondary storage device for high storage capacity and low cost.

Data stored as magnetized areas on magnetic disk surfaces.

A **disk pack** contains several magnetic disks connected to a rotating spindle.

Disks are divided into concentric circular **tracks** on each disk **surface**.

- Track capacities vary typically from 4 to 50 Kbytes or more

Disk Storage Devices (contd.)

A track is divided into smaller **blocks** or **sectors**

- because it usually contains a large amount of information

The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed.

- One type of sector organization calls a portion of a track that subtends a fixed angle at the center as a sector.

A track is divided into **blocks**.

- The block size B is fixed for each system.
  - Typical block sizes range from B=512 bytes to B=4096 bytes.
- Whole blocks are transferred between disk and main memory for processing.

Disk Storage Devices (contd.)

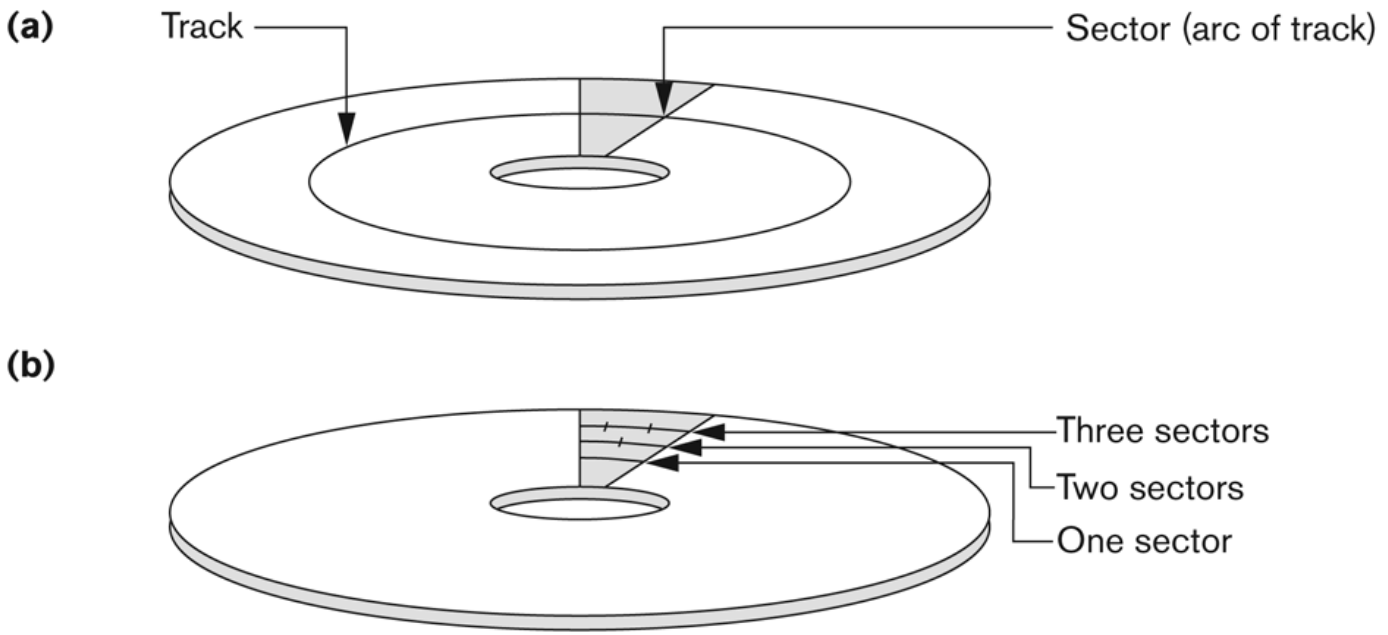
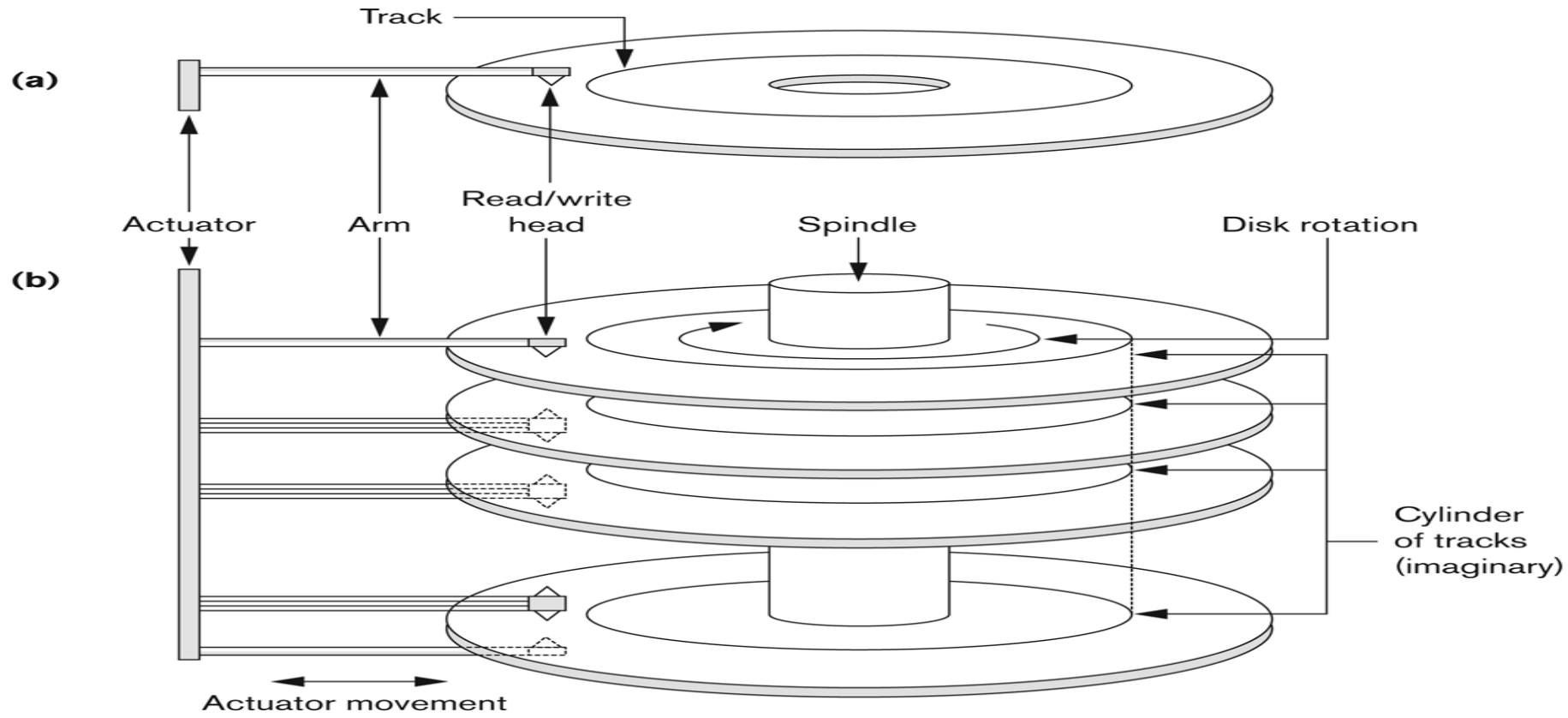


Figure 13.2
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

Disk Storage Devices (contd.)

Figure 13.1
(a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.



Files of Records

A **file** is a *sequence* of records, where each record is a collection of data values (or data items).

A **file descriptor** (or **file header**) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.

Records are stored on disk blocks.

The **blocking factor bfr** for a file is the (average) number of file records stored in a disk block.

A file can have **fixed-length** records or **variable-length** records.

Records

Fixed Length Records

- All records in a file are of the same record type. If every record in the file has exactly the same size.

Variable length records

- If different records in the file have different sizes.

Mixed filed

- Records contain fields which have values of a particular type

E.g., amount, date, time, age

Fields themselves may be fixed length or variable length

Variable length fields can be mixed into one record:

- Separator characters or length fields are needed so that the record can be "parsed."

Blocking

Blocking:

- Refers to storing a number of records in one block on the disk.

Blocking factor (**bfr**) refers to the number of records per block.

There may be empty space in a block if an integral number of records do not fit in one block.

Spanned Records:

- Refers to records that exceed the size of one or more blocks and hence span a number of blocks.

Files of Records (contd.)

File records can be **unspanned** or **spanned**

- **Unspanned**: no record can span two blocks
- **Spanned**: a record can be stored in more than one block

The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.

In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.

Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**.

- Usually spanned blocking is used with such files.

Operation on Files

Typical file operations include:

- **OPEN**: Readies the file for access, and associates a pointer that will refer to a *current* file record at each point in time.
- **FIND**: Searches for the first file record that satisfies a certain condition, and makes it the current file record.
- **FINDNEXT**: Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
- **READ**: Reads the current file record into a program variable.
- **INSERT**: Inserts a new record into the file & makes it the current file record.
- **DELETE**: Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
- **MODIFY**: Changes the values of some fields of the current file record.
- **CLOSE**: Terminates access to the file.
- **REORGANIZE**: Reorganizes the file records.
  - For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- **READ_ORDERED**: Read the file blocks in order of a specific field of the file.

Unordered Files

Also called a **heap** or a **pile** file.

New records are inserted at the end of the file.

A **linear search** through the file records is necessary to search for a record.

- This requires reading and searching half the file blocks on the average, and is hence quite expensive.

Record insertion is quite efficient.

Reading the records in order of a particular field requires sorting the file records.

Ordered Files

Also called a **sequential** file.

File records are kept sorted by the values of an *ordering field*.

Insertion is expensive: records must be inserted in the correct order.

- It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.

A **binary search** can be used to search for a record on its *ordering field* value.

- This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.

Reading the records in order of the ordering field is quite efficient.

Indexing

SINGLE LEVEL AND MULTI LEVEL INDEXING, DYNAMIC MULTI LEVEL INDEXING USING B TREES AND B+ TREES

Indexing

Classification of Indexing
- Dense and Sparse Indexing

Types of Indexing
- Single and Multi level

Single Level
-Primary Index
-Secondary Index
- Cluster Index

Multi Level
-B Tree
-B+ Tree

Indexing -> Reduce the number of disk access required.
Optimize the Performance of Database

Index -> It is a data strictures which is used to easy to relate the access the data in database table.

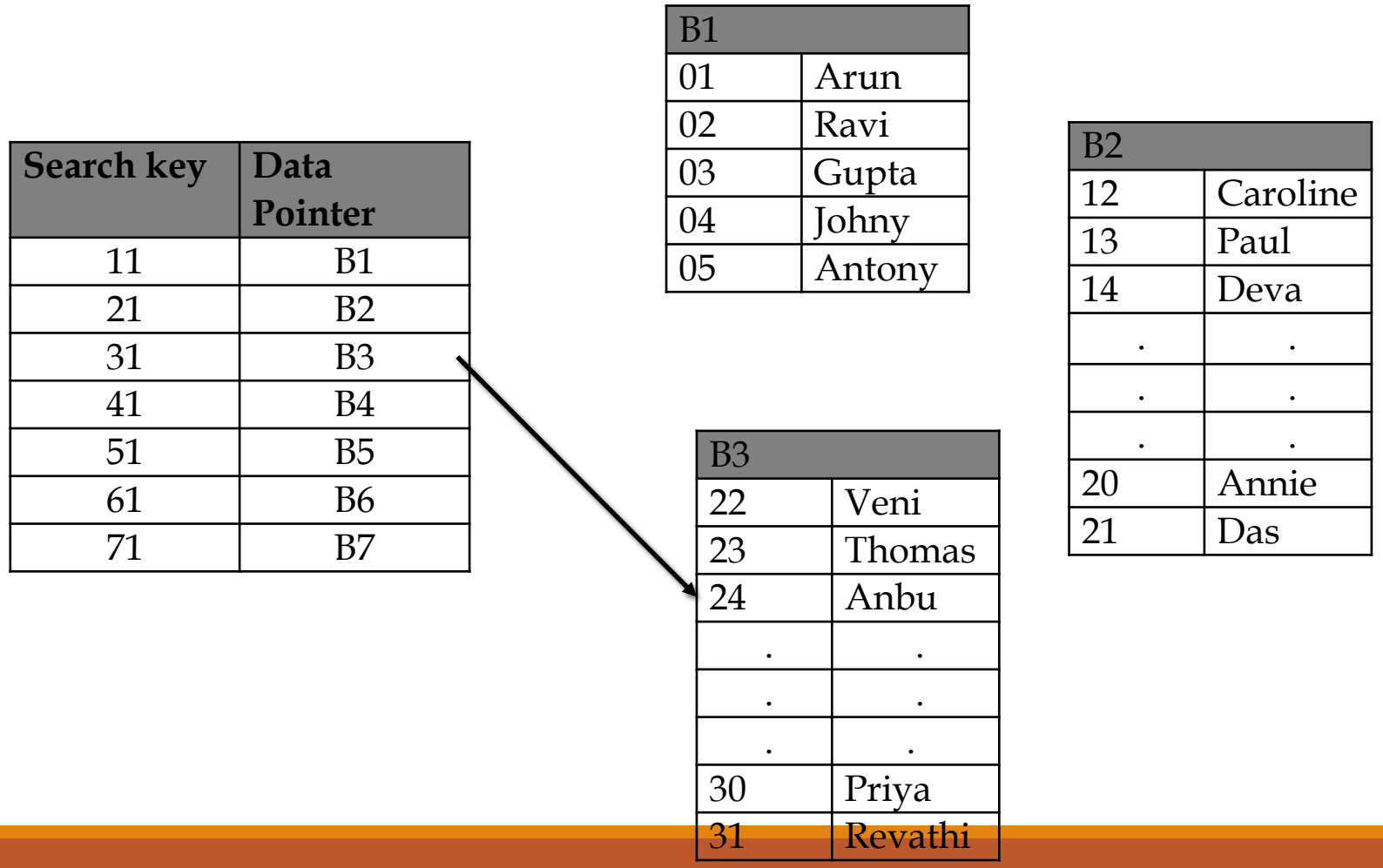
Example -> Deadlock 20, 37-38, 128,

Structures of Index

Search Key	Data Pointer

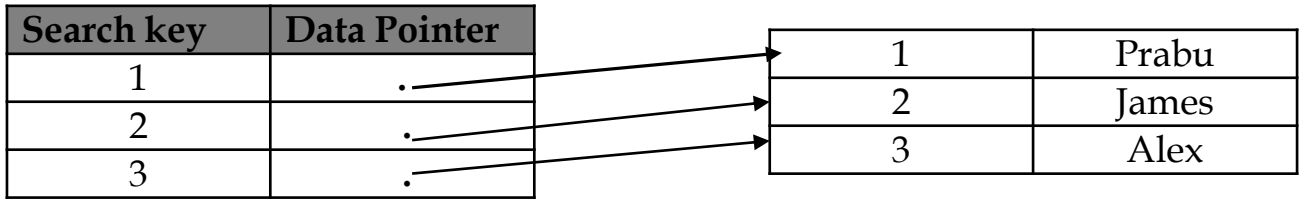
Search Key contains: Primary Key and Candidate Key

Data Pointer (Data Reference): Contains set of pointer holding the address of block.

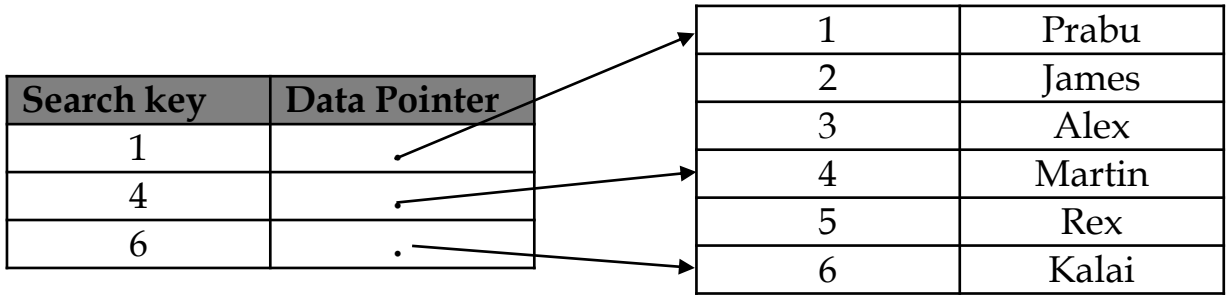

Search key
Data Pointer
11
B1
21
B2
31
B3
41
B4
51
B5
61
B6
71
B7
B1
01
Arun
02
Ravi
03
Gupta
04
Johny
05
Antony
B2
12
Caroline
13
Paul
14
Deva
20
Annie
21
Das
B3
22
Veni
23
Thomas
24
Anbu
30
Priya
31
Revathi

Dense Vs Sparse Indexing

Dense Indexing: An index entry is created for every search key value

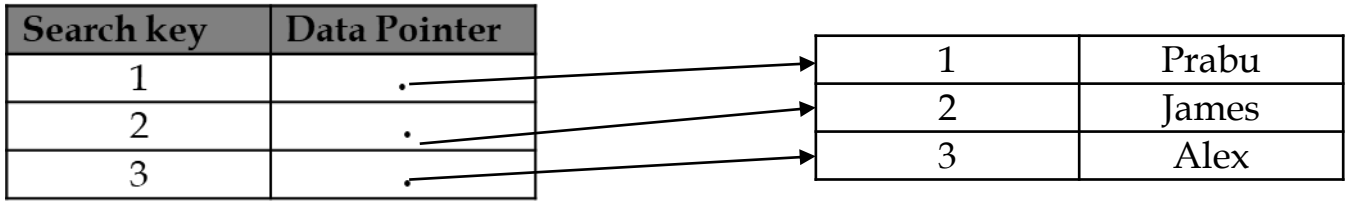


Sparse Indexing: An index entry is created for some records.



Single Level and Multi level

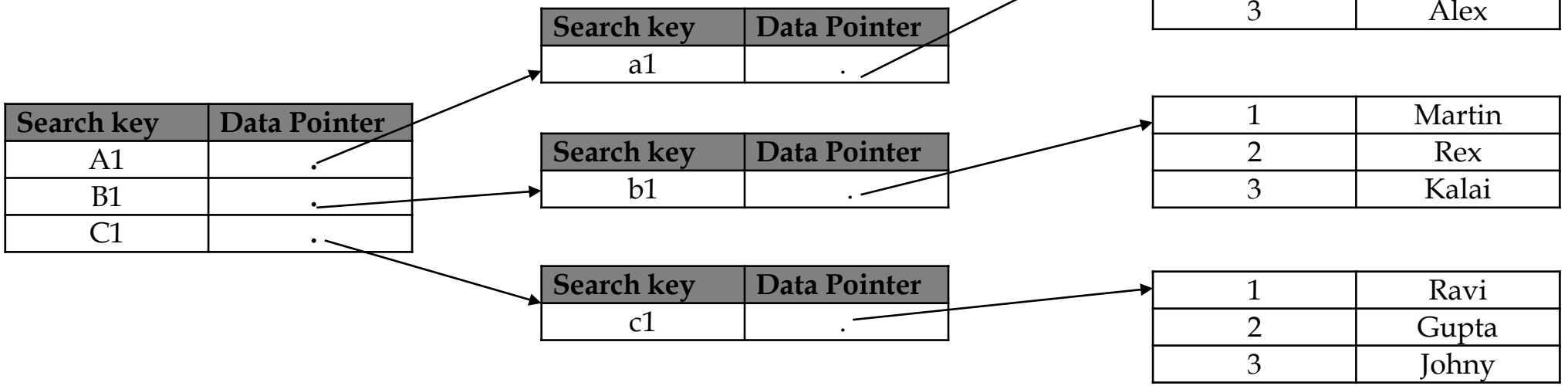
Single Level



Search key	Data Pointer
1	.
2	.
3	.

1	Prabu
2	James
3	Alex

Multi Level



Search key	Data Pointer
A1	.
B1	.
C1	.

Search key	Data Pointer
a1	.

Search key	Data Pointer
b1	.

Search key	Data Pointer
c1	.

1	Prabu
2	James
3	Alex

1	Martin
2	Rex
3	Kalai

1	Ravi
2	Gupta
3	Johny

Primary Indexing(Primary Key + Ordered Data)

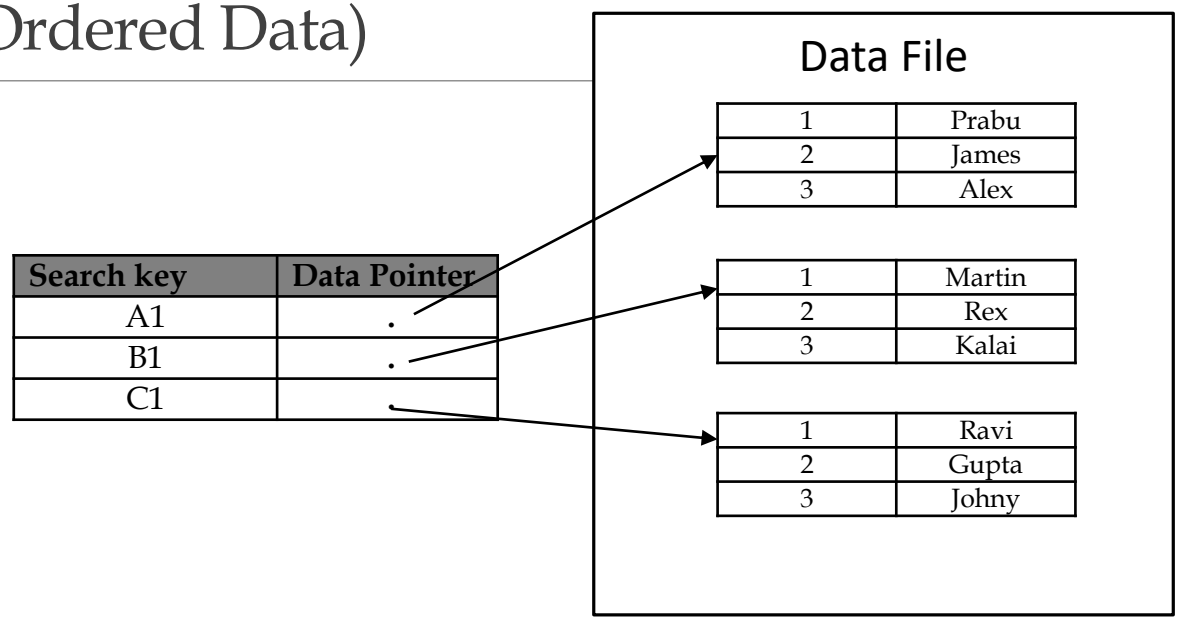
1. Fixed length size with two fields.

Search Key	Data Pointer

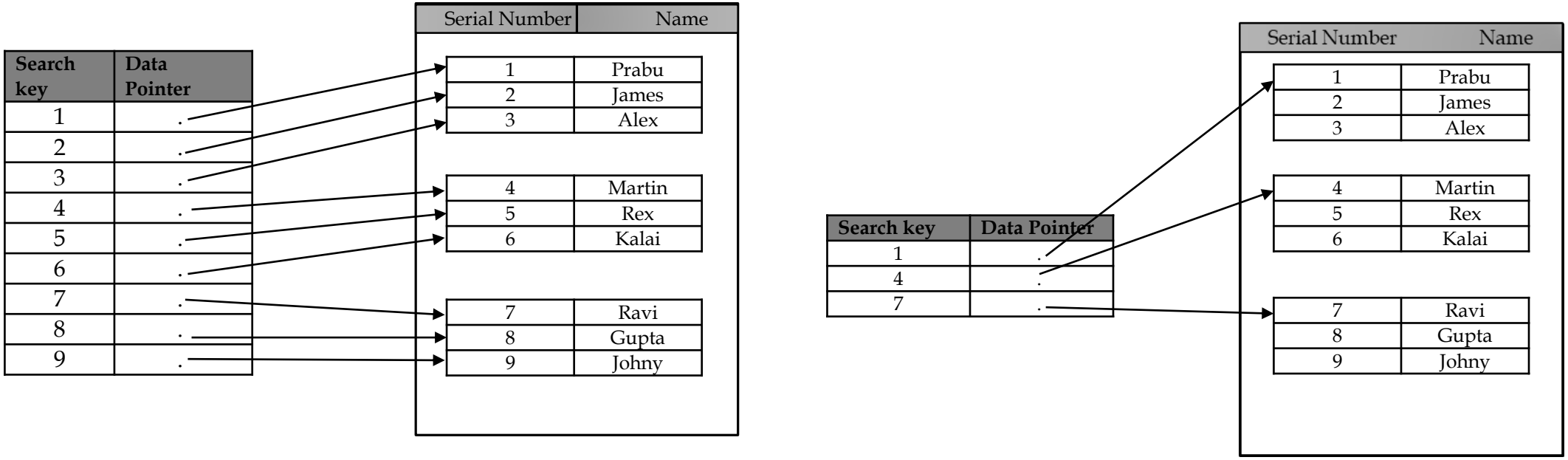
2. Index entry is created for first record of Each block is called “Block Anchor”.

3. No. of index entries = No. of blocks.

4. No. of access required = $\log_2 n$ +1



Primary Dense Index Vs Primary Sparse Index



Primary Indexing: * {ordering + key}

↳ P.K {unique + Null}

Example: Index: (structure)

Primary Dense Indexing

P.K	D.P

↳ Data Pointer

↓ Search key

S.K	D.P
1	•
2	•
3	•
4	•
5	•

Emp. No.	Emp. No.	Age
1	Rahul	27
2	Ravi	31
3	Rekha	34
4	Richa	37
5	Raghav	38

No - Replicates

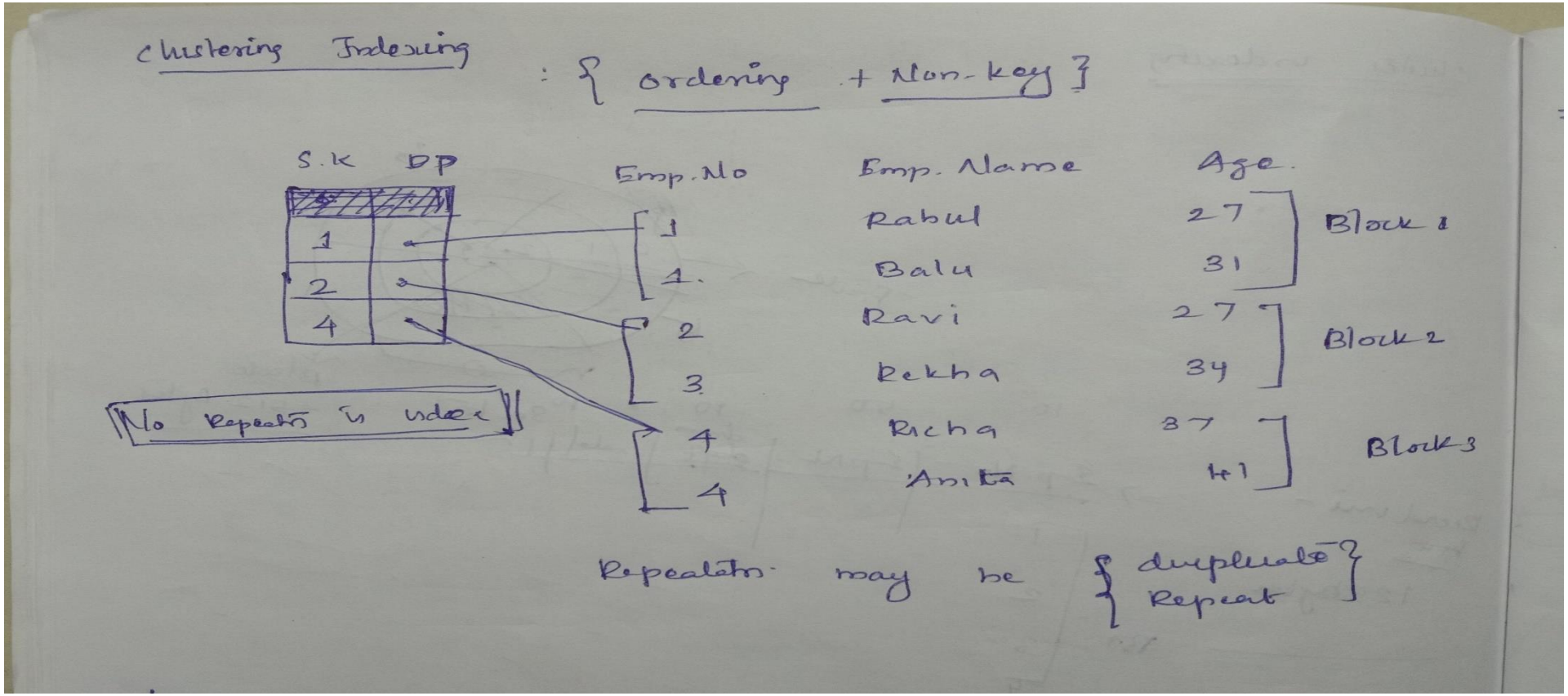
Database: ordering + key

Primary Sparse Indexing:

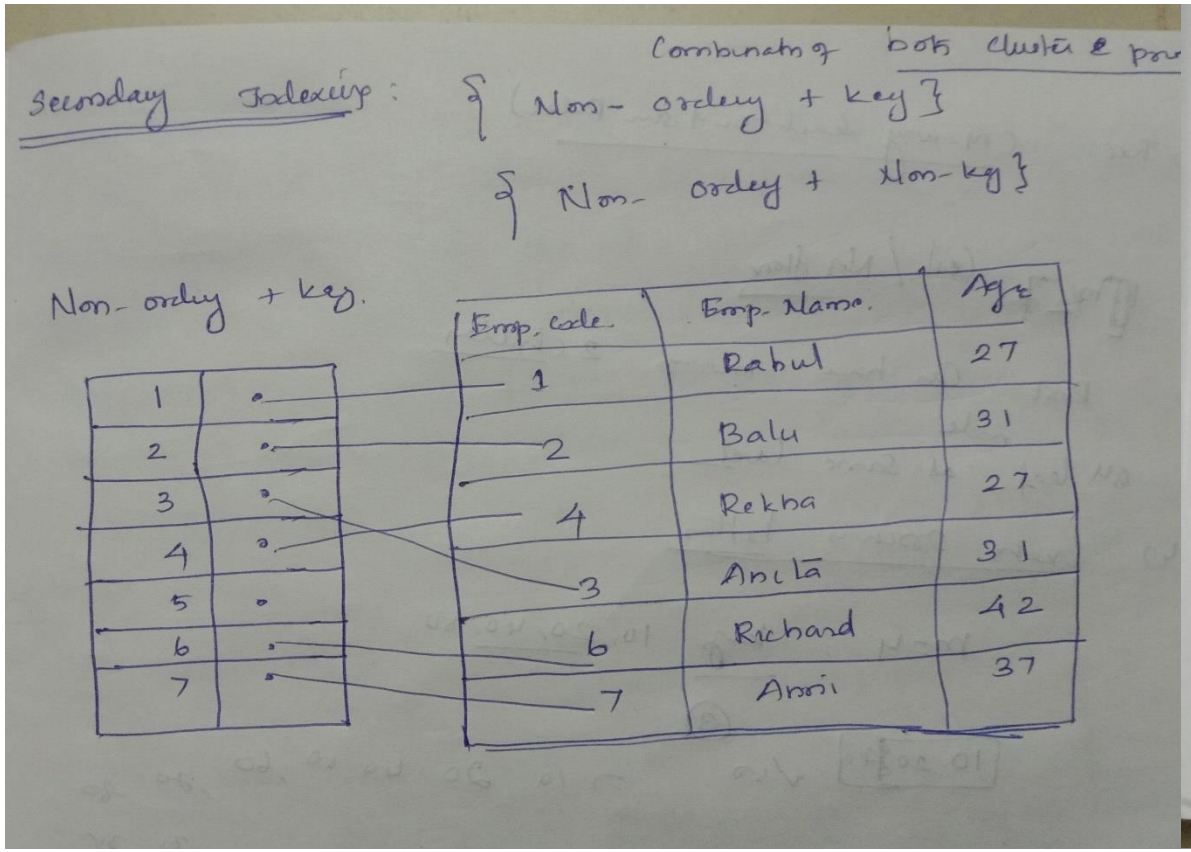
S.K	B.P
1	•
3	•
5	•

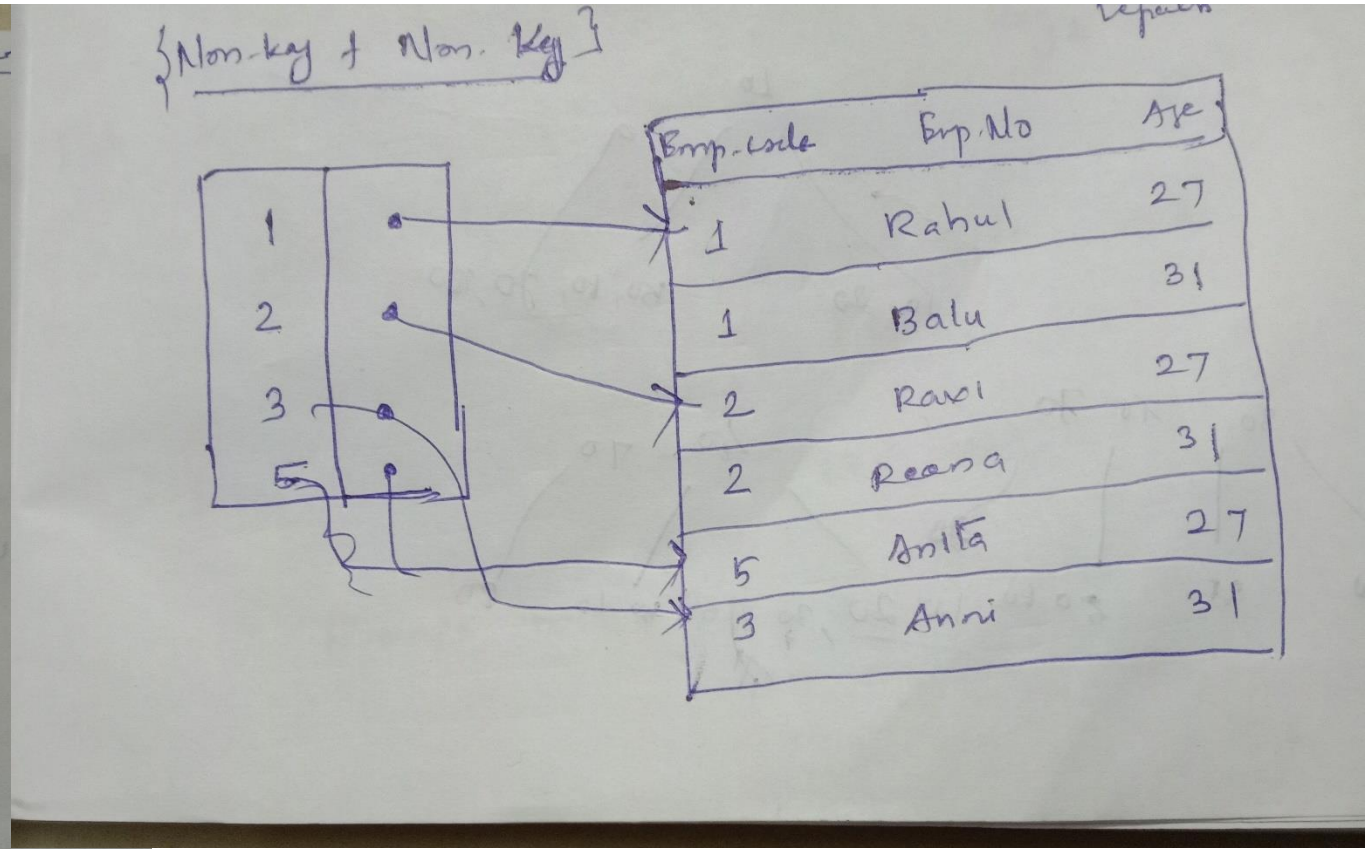
Emp. No	Emp. No	Age	
1	Rahul	27	B1
2	Ravi	31	
3	Rekha	34	B2
4	Richa	37	
5	Raghav	38	B3
6	Roona	40	

Cluster Indexing



Secondary Indexing





Number of Records=30000, block size=1024 bytes, Strategy =unspanned, Record size=100bytes, key size=6 bytes, pointer size=9 bytes, Find the average number of block access with (or) without index?

Data record that can for in single block=B/R= 1024/100=10.24 (round of/ whole no)

Total Record =30000, Total no of block= 30000/10=3000 Blocks.

Without index, we need Log n= log 3000= 12 block (11.55)

With index. Size of index = key size+ pointer size= 9+6=15

Index= block size/size of index= 1024/15= 68 (68.2)

Index record=No of data Blocks =3000

No. of index Blocks(n)=Index record/index=3000/68=45.

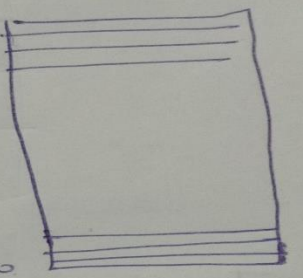
So average no of blocks access= $\log_2 n$ +1= $\log_2 45$ +1= 6+1=7

1	Prabu
2	James
3	Alex
.	.
.	.
.	.
100	Martin
101	Rex
.	.
.	.
30000	Johny

Primary Index:

An ordered file with r = 30,000 records stored on a disk with block size B = 1024 Bytes. File records are a fixed size and are unspanned, with record length R = 100 byts. key size = 6 bytes, pointer size = 9 bytes, Find the average number of block access with index (or) without index?

Without Index:



Given data.

(i) Total Records (r) = 30,000

(ii) Block size (B) = 1024,

(iii) Record size (R) = 100

Single block = $[B/R] = 1024/100 = 10.24 = 10$

Total No. of Block = $30,000/10 = 3000$

Average Search (Index) = $\log_2 n = \log_2 3000 = 11.55$

$= \frac{12}{9}$ blocks $\frac{}{b}$

S.K	D.P
V	R

Klub Index:

Index = Block size (B) / size of index (V+R)

$= 1024/15 = 68$ [68.2]

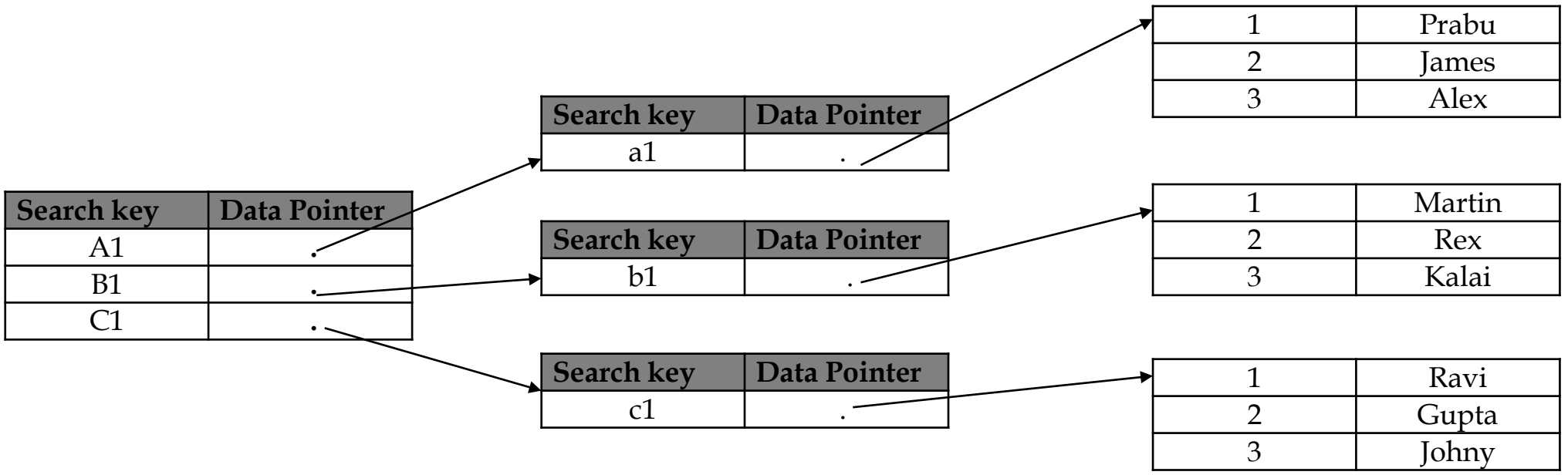
Total No. of Blocks = $3000/68 = 45$

Average no. of blak access = $\log_2 n + 1$

$= \log_2 45 + 1$

$= 6 + 1 = 7$

B Tree & Rules


Search key
Data Pointer
A1
B1
C1
Search key
Data Pointer
a1
Search key
Data Pointer
b1
Search key
Data Pointer
c1
1
Prabu
2
James
3
Alex
1
Martin
2
Rex
3
Kalai
1
Ravi
2
Gupta
3
Johny

Rules for B-Tree:-

*) Balanced m-way tree [m → order]

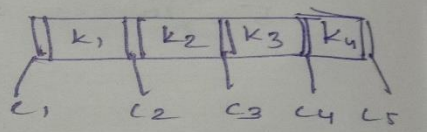
*) Can have more than one key and more than two children.

*) All leaf node must be at same level.

*) Order = m,

→ Every node has m- children

Order = 5
Children = 5



key = 4

min Children

Leaf node : 0

Root node : 2

Intermediate node : $\lceil m/2 \rceil$ = Ceil. $2.5 = 3$

$\lfloor m/2 \rfloor = 2.5 = 2$

Floor

* keys:-

maximum (m-1) keys.

min. keys:- Root node : 01

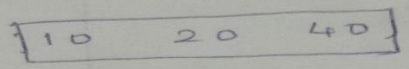
allnode [Except Root node]

$= \lceil m/2 \rceil - 1$

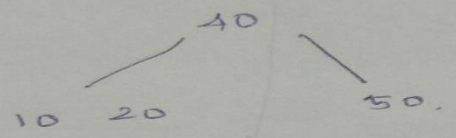
* Insert, should be in Leaf node.

Values: 10, 20, 40, 50, 60, 70, 80, 30, 35, 05, 15

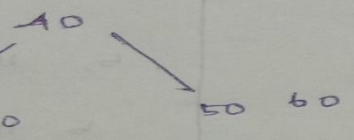
order (m) = 4. key should be : m-1 = 4-1 = 3



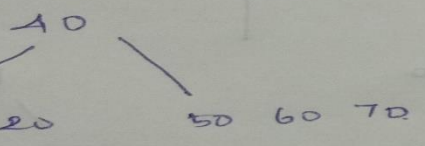
Insert 50:



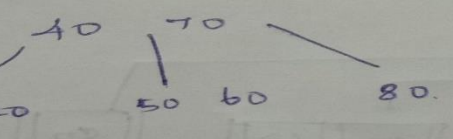
Insert: 60



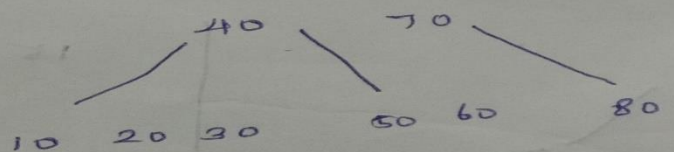
Insert: 70:



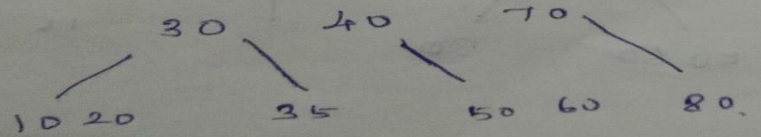
Insert 80:

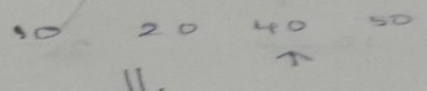


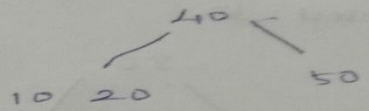
Insert 30

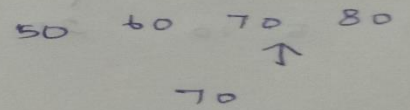


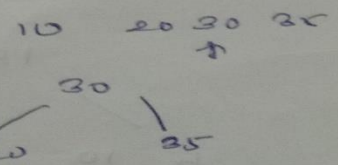
Insert 35:

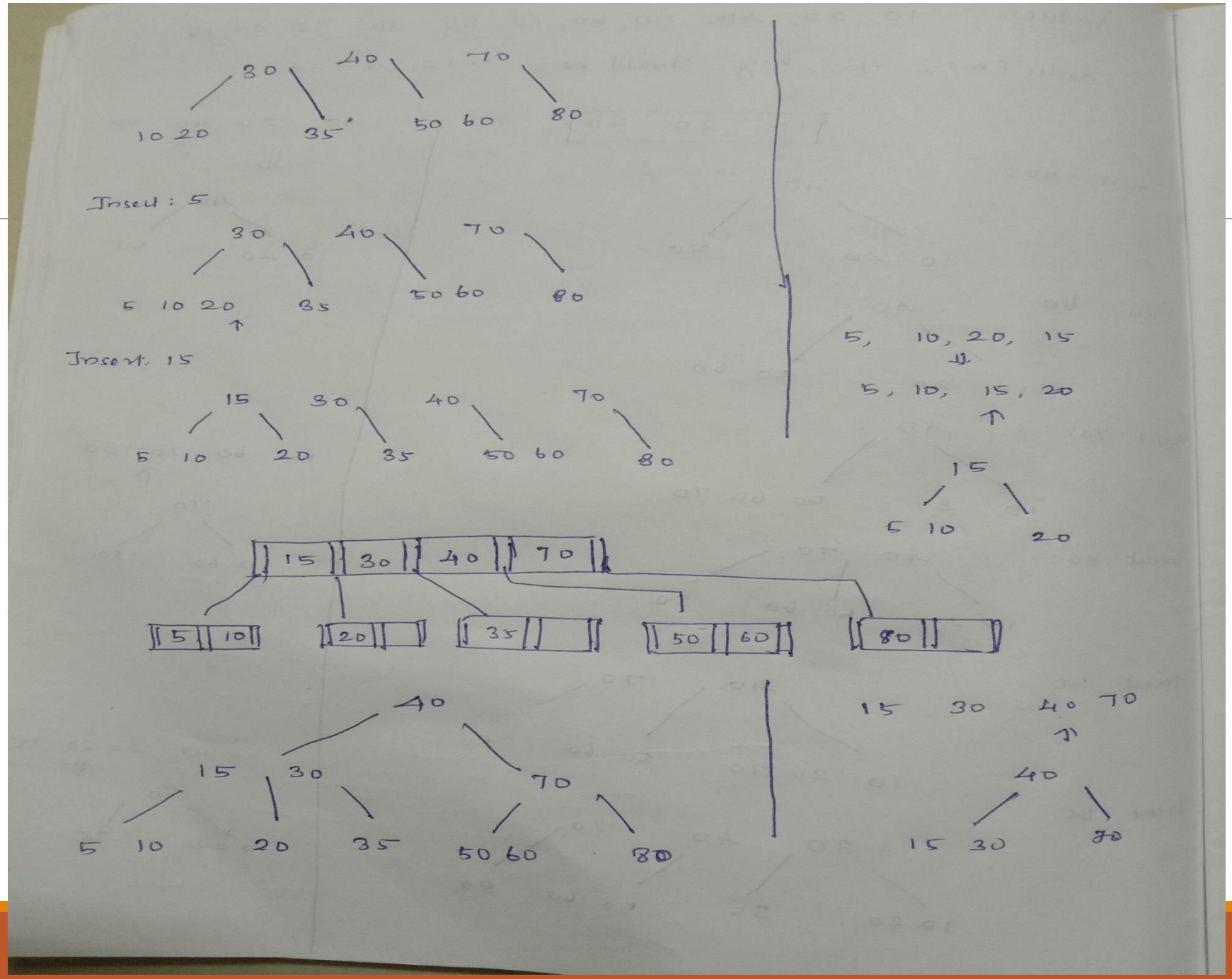











30 40 70
10 20 35 50 60 80
Insert : 5
30 40 70
5 10 20 35 50 60 80
Insert: 15
15 30 40 70
5 10 20 35 50 60 80
15 30 40 70
5 10 20 35 50 60 80
40
15 30 70
5 10 20 35 50 60 80
5, 10, 20, 15
5, 10, 15, 20
15
5 10 20
15 30 40 70