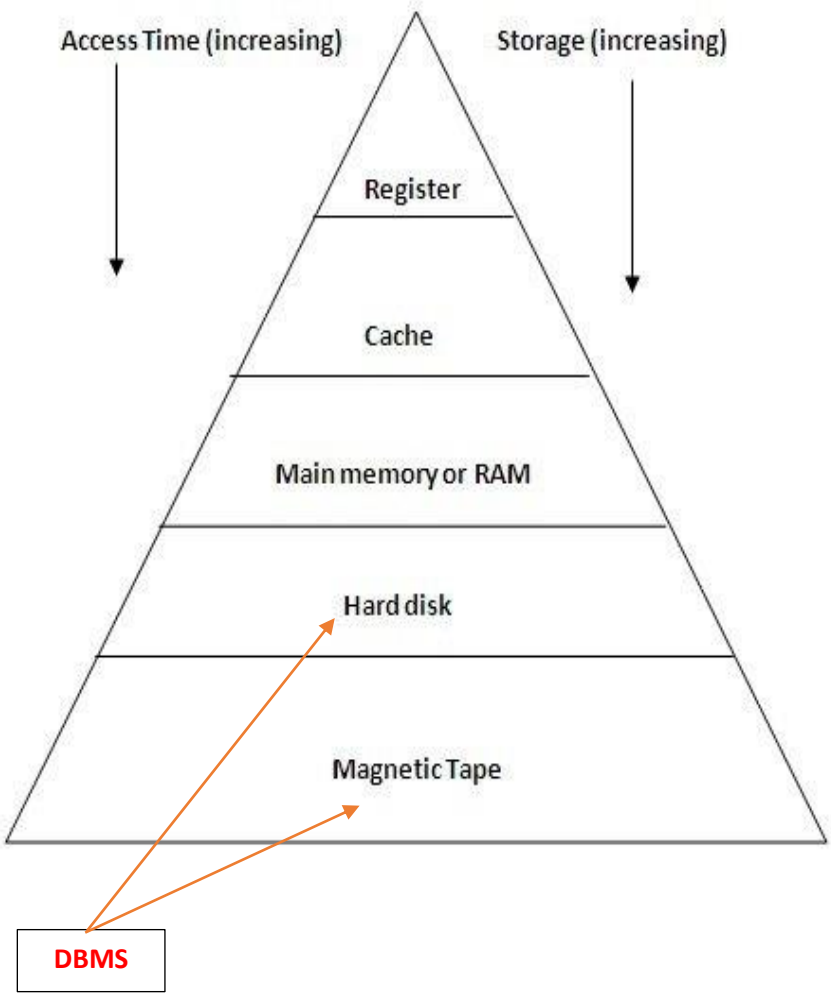
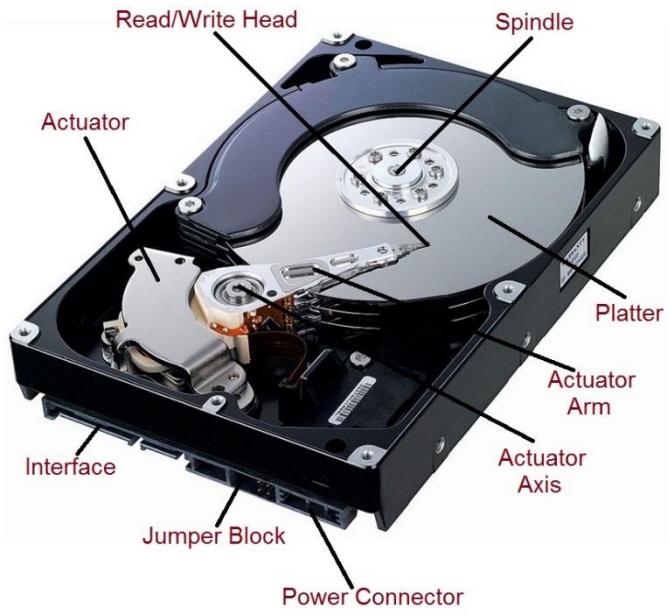
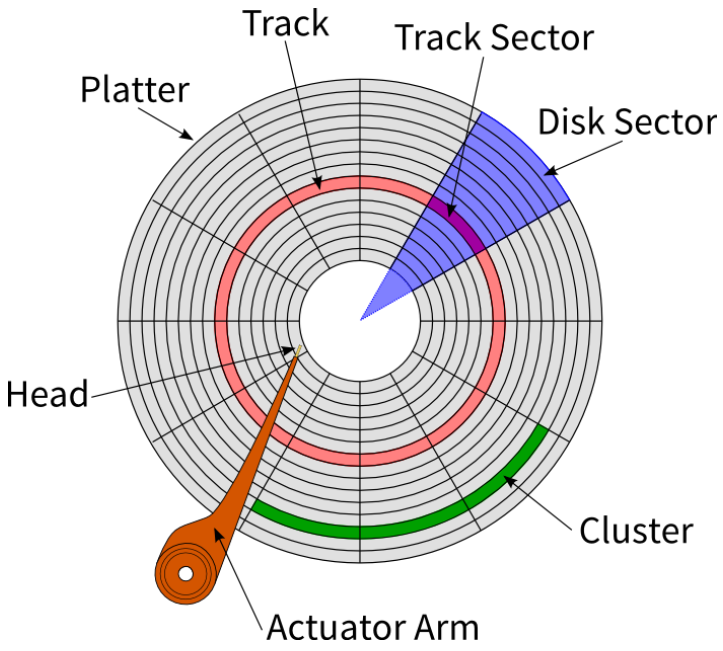
The Memory Hierarchy



Internal Register >> Information inside the CPU is stored in registers. **Hardware>>** D flip-flops
Cache >> It is used to improve latency of fetching information from Main Memory to CPU registers. *Types:* L1, L2 & L3 cache. **Hardware >>** SRAM (6 transistors)
Main Memory (RAM) >> Program Instructions and Data are normally loaded into RAM memory. **Hardware >>** DRAM (capacitor, transistor)
Secondary Storage (HDD) >> Permanent storage of programs and data. **Hardware >>** Magnetic Disk, SSD(microchip)
Tertiary Storage >> updates less frequently than secondary and is not constantly online at all. **Hardware >>** Magnetic tapes, Optical disks/tapes

HDD >>





Mohammad Imam Hossain, Lecturer, Dept. of CSE, UIU. Email: imambuet11@gmail.com

Transfer of Data>>

Disk Blocks is a group of sectors that the operating system can address. Entire blocks are moved to or from a continuous section of main memory called buffer.

For example, NTFS Block Size is 4096 bytes(4KB). Block size can vary from 4-64 KB.

- A key technique for speeding up database operations is to arrange data so that when one piece of a disk block is needed, it is likely that other data on the same block will also be needed at about the same time.
- It is not sufficient simply to scatter the records that represent tuples of a relation among various blocks.

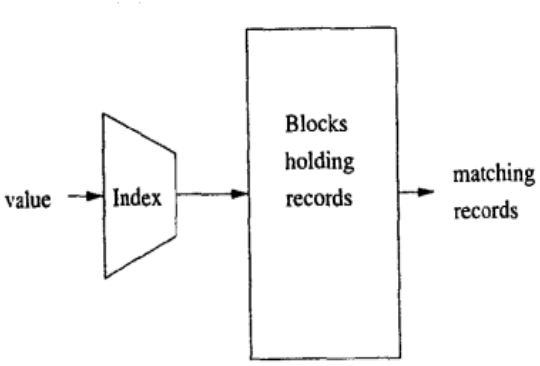
Indexing

Queries like,

"Find all accounts at the Perryridge branch"

references only a fraction of the account records.

It is inefficient for the system to read every record and to check the *branch-name* field for the name "Perryridge". That is why we use **index structure** to gain fast random access to records in a file.



For example, to retrieve an *account* record given the account number

- The database system would look up an index to find on which disk block the corresponding record resides and then fetch the disk block, to get the *account* record.

Types of Indices>>

i. **Ordered indices:** Based on a sorted ordering of the indexed key values.
ii. **Hash indices** **:** Based on a uniform distribution of indexed key values (determined by hash function) across a range of buckets.

What type you will consider for your system depends on several factors, such as

- **Access types**: The types of access (Point queries, Range queries) that are supported efficiently.
- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to *find the correct place* to insert the new data item, as well as the time it takes to *update the index structure*.
- **Deletion time**: The time it takes to delete a data item. This value includes the time it takes to *find the item* to be deleted, as well as the time it takes to *update the index structure*.
- **Space overhead**: The additional space occupied by an index structure. Provided that, the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

Ordered Indices >>

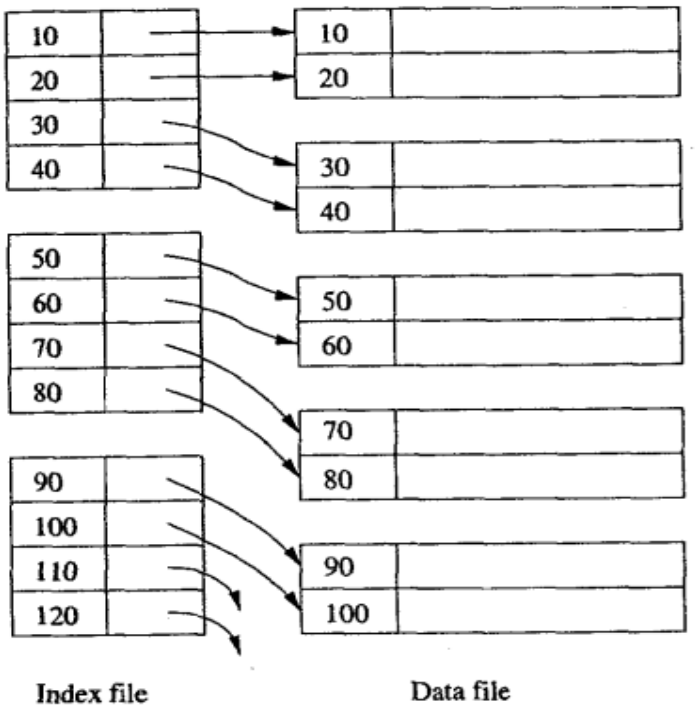
- Each index structure is associated with a particular search key. An attribute or set of attributes used to look up records in a file/disk block/page is called a **search key.**
- An **index record** consists of a *search-key value*, and *pointers* to one or more **data records** with that value as their search-key value.
  The pointer to a data record consists of the *identifier of a disk block* and *an offset* within the disk block to identify the record within the block.
- An *ordered index* stores the values of the search keys in sorted order, and associates with each search key the data records that contain it.

Different Types of Ordered Indices>>

1. **Primary Index:** If the file containing the records is sequentially ordered, a **primary index** is an ordered index whose search key also defines the sequential order of the file. Primary indices are also called **clustering indices**.

   The search key of a primary index is usually the primary key, although that is not necessarily so.

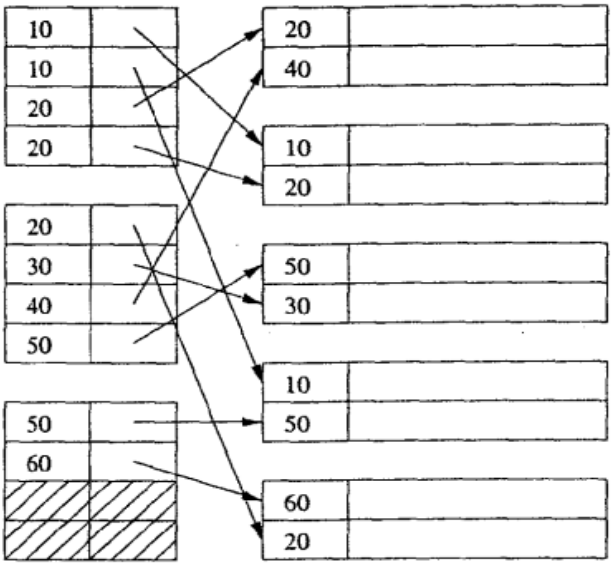
   If all files are ordered sequentially on some search key, then such files with a primary index on the search key, are called **index-sequential files**.



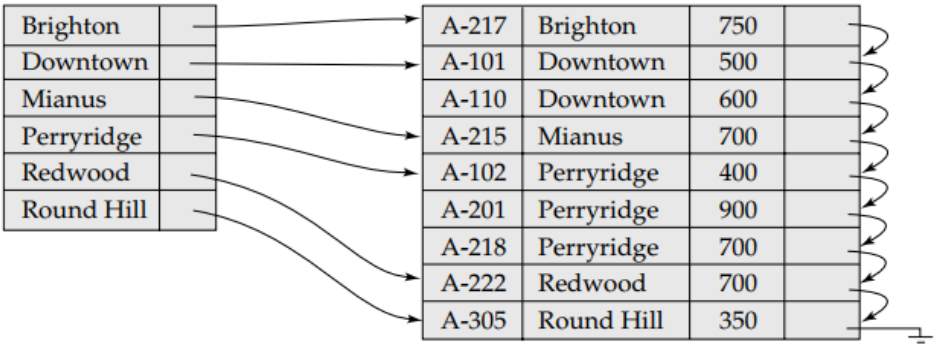
2. **Secondary Index:** Ordered Indices whose search key specifies an order different from the sequential order of the file are called **secondary indices**, or **non-clustering** indices.

   Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file.

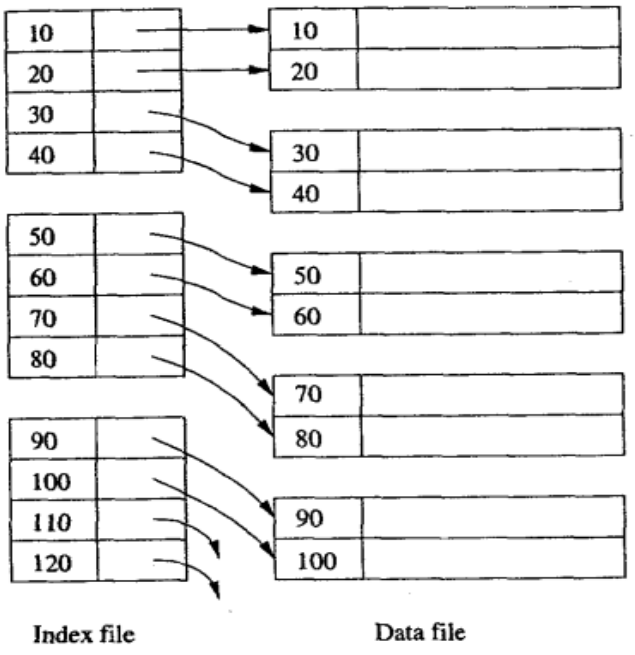
If a secondary index stores only some of the search-key values(sparse), records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.



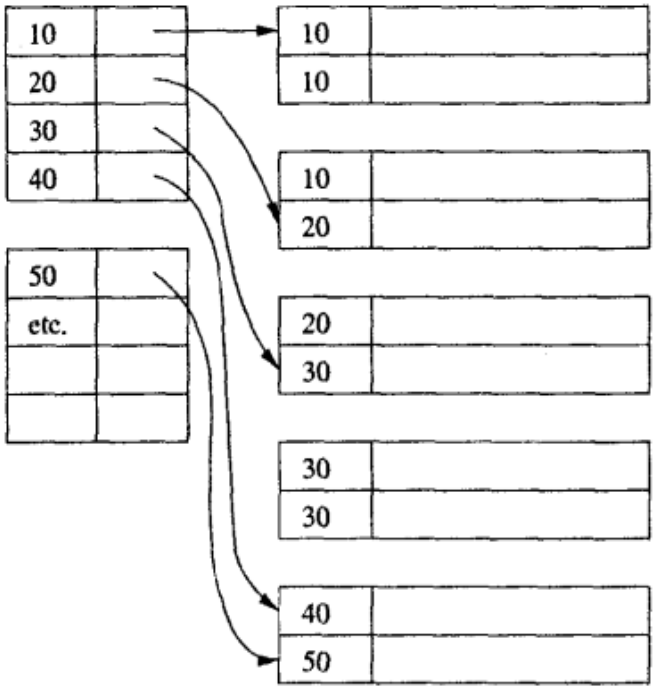
3. **Dense Index:** An index record appears for every search-key value in the file.



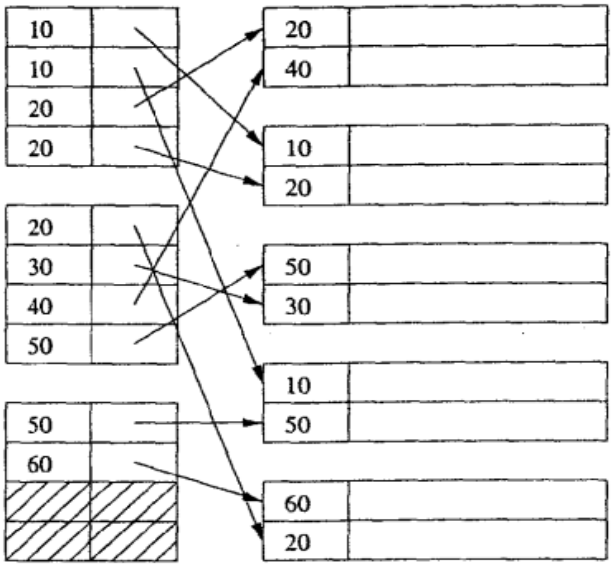
Type 1) **Dense + Primary Index + No Duplicate Search Keys**



Type 2) **Dense + Primary Index + Duplicate Search Keys**



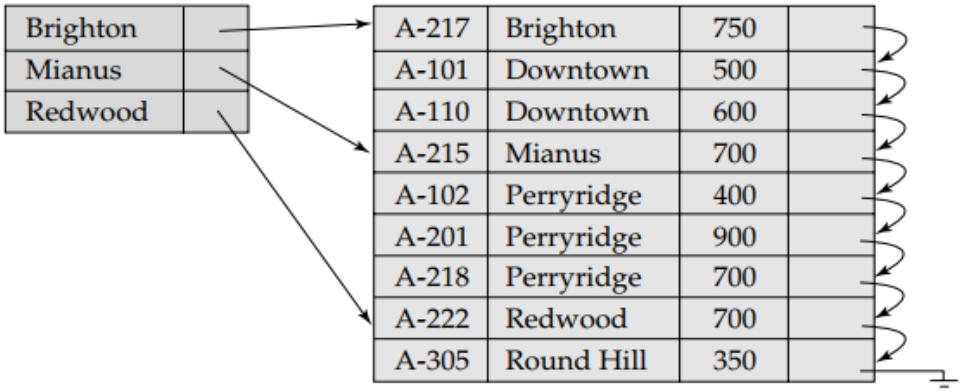
Type 3) **Dense + Secondary Index (with/without Duplicate Search Keys)**



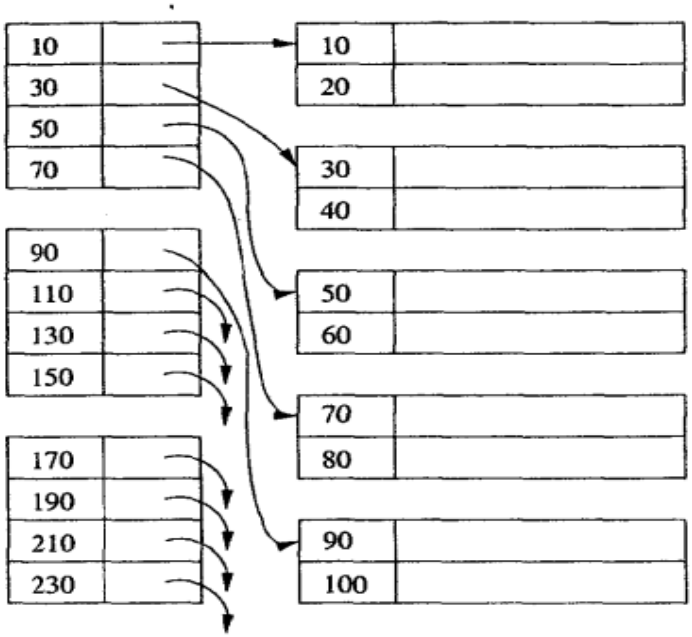
4. **Sparse Index:** An index record appears for only some of the search-key values.

   To locate a record,
   we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

   A good design is to have a sparse index with *one index entry per block* as the time to scan the entire block is negligible than the time to bring a block from disk into main memory.



Type 1) **Sparse + Primary Index**



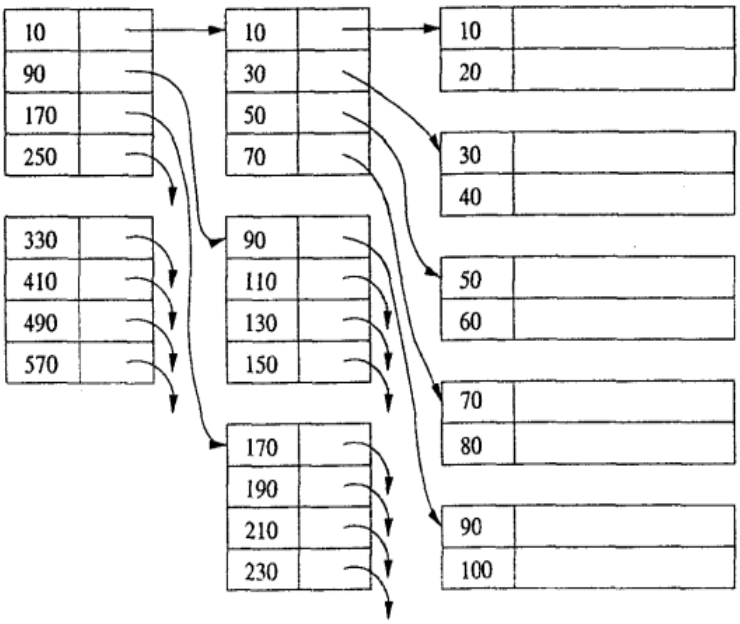
Type 2) **Sparse + Secondary Index** IS IT POSSIBLE!!!!!!!

Dense vs Sparse Index:

- It is generally faster to locate a record if we have a dense index rather than a sparse index.
- However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.
- In practice,
  to have a file with 100,000 records, with 10 records stored in each data block. If we have one index record per block, the index has 10,000 records. Index records are smaller than data records, so let us assume that 100 index records fit on a block. Thus, our index occupies 100 blocks.

  Such large indices are stored as sequential files on disk. A search for an entry in the index block requires as many as $\lceil \log_2(b) \rceil$ blocks to be read(binary search).

5. **Multilevel Indices:** The process of searching a large index structure may be costly. The solution is Multilevel Index (Indices with two or more levels).

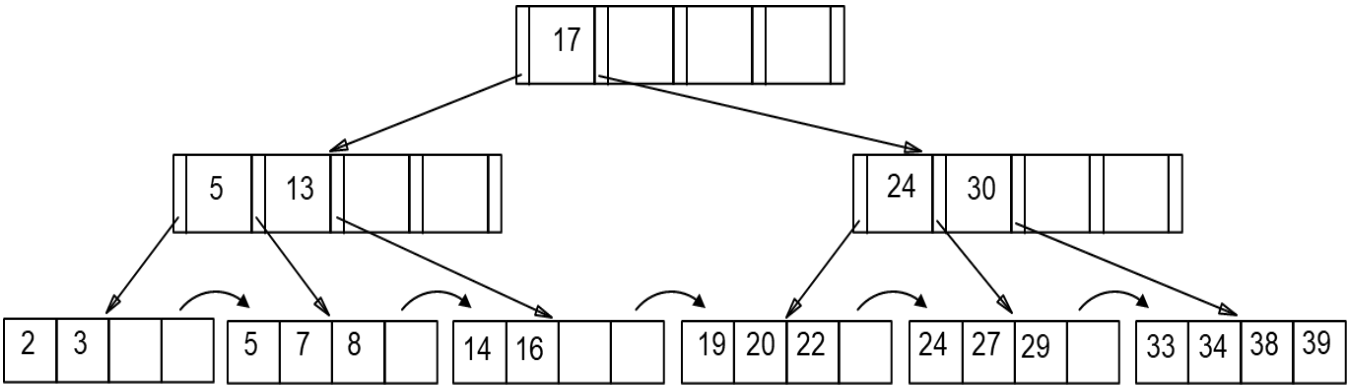


Problems of Multilevel Indices?

6. **B^+ Tree Index Structure:** This 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 a **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 B^+ tree >>
   - Degree/Order/Maximum no of Pointers = n = 5 and Maximum no of keys = n-1 = 4
   - The search key values within a node are kept in ascending sorted order.



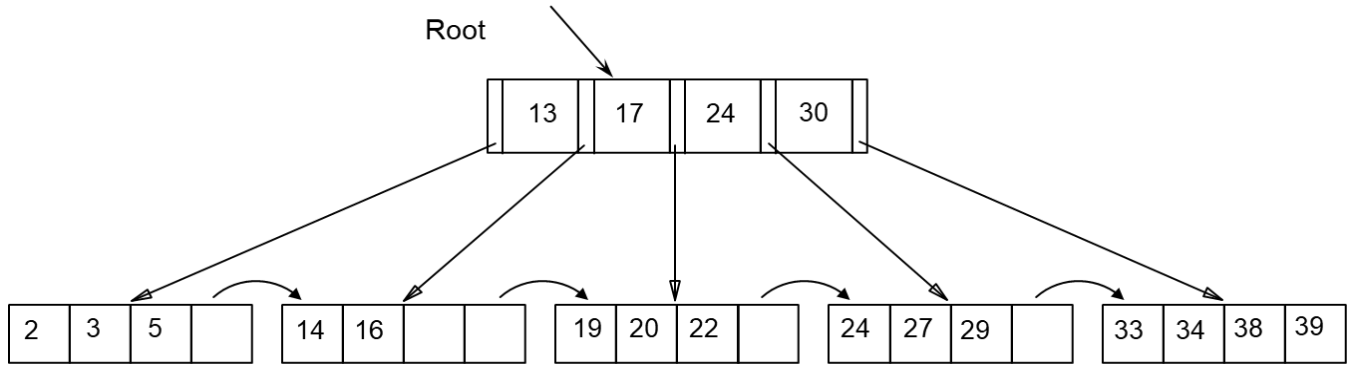
   - Root node >> Minimum no of keys =1 and minimum no of pointers = 2
   - Non-leaf node >> Minimum no of keys = $\left\lceil \frac{n}{2} \right\rceil - 1$ and minimum no of pointers = $\left\lceil \frac{n}{2} \right\rceil$
   - Leaf node >> Minimum no of keys = $\left\lceil \frac{n-1}{2} \right\rceil$ and minimum no of pointers = $\left\lceil \frac{n-1}{2} \right\rceil + 1$
   - Left child node search key values are less then the parent key value and Right child values are greater than or equal to the parent key value.

Example, for n=5,

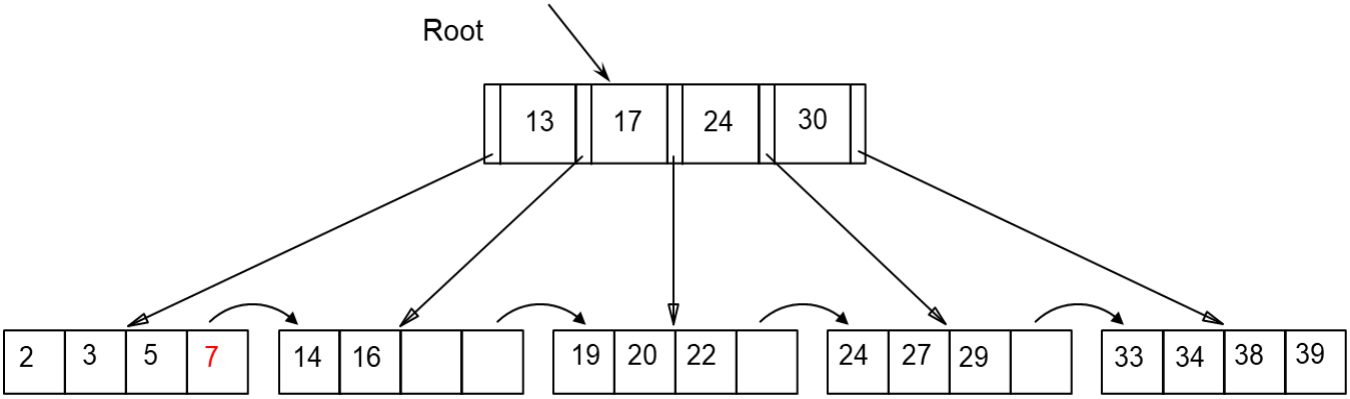
- Each node contains maximum 5 pointers
- Each node contains maximum 4 key values
- Each root node contains at least 2 pointers
- Each non-leaf node contains at least $\lceil 5/2 \rceil$ = 3 pointers that is at least 2 key values
- Each leaf node contains at least $\lceil (5-1)/2 \rceil$ = 2 key values

Insertion into a B$^+$ Tree >>

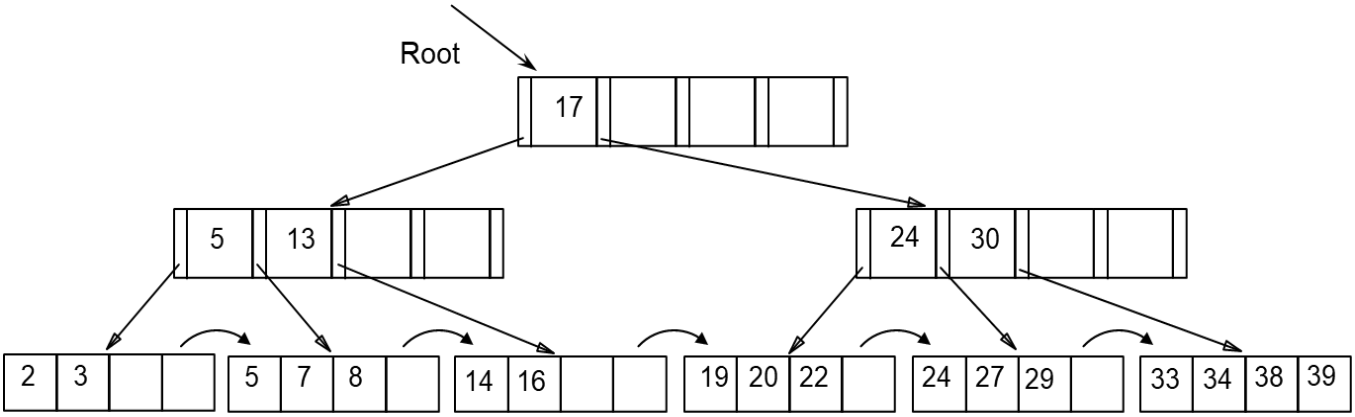
initially,



inserting 7(free slot exists),



inserting 8(no free slot + copy up + push up),



Class Practice Samples:

1. Construct a B^+ tree for the following set of key values, where each internal node can contain at most 4 childrens. Assume that the tree is initially empty and values are added sequentially one by one.

   a) 11, 61, 101, 5, 40, 25, 80, 30, 92, 130, 165, 35, 50, 56
   b) 5, 50, 100, 25, 40, 45, 150, 80, 30, 15, 35