Module - 3

MULTILEVEL INDEXING AND B-TREES

5.1 Introduction

Limitation of Indexing i.e if index file itself is so voluminous that only rather small parts of it can be kept in main memory at one time, led to utilization of trees concept to file structures.

Evolution of the trees for file structures started with applying Binary search tree (BST) for file structures.

The limitation was unbalanced growth of the BST.

This gave rise to applying of AVL tree concept to File structures. With this the problem of height balance was addressed but to make the tree height balance lot of local rotations was required. This limitation gave rise to development of B- Trees.

Binary tree:	A tree in which each node has at most two children.
Leaf:	A node at the lowest level of a tree.
Height-balanced tree:	A tree in which the difference between the heights of subtrees in limited.

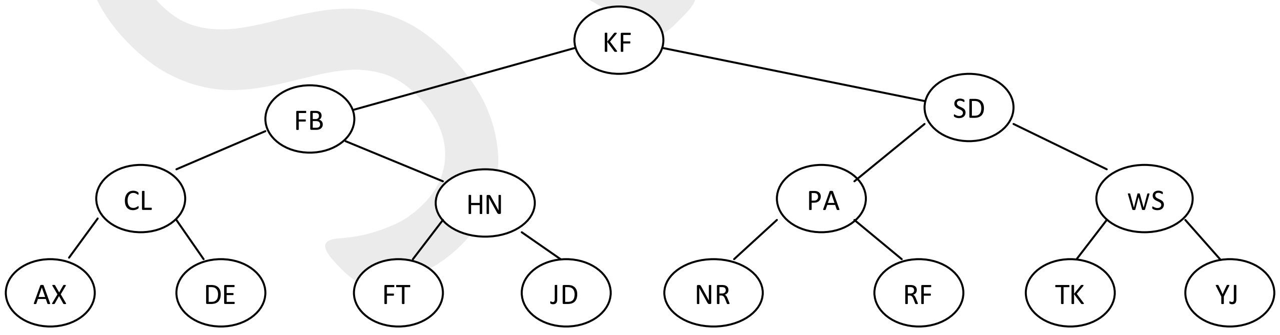
5.2 Statement of the problem

- Searching an index must be faster than binary searching.
- Inserting and deleting must be as fast as searching.

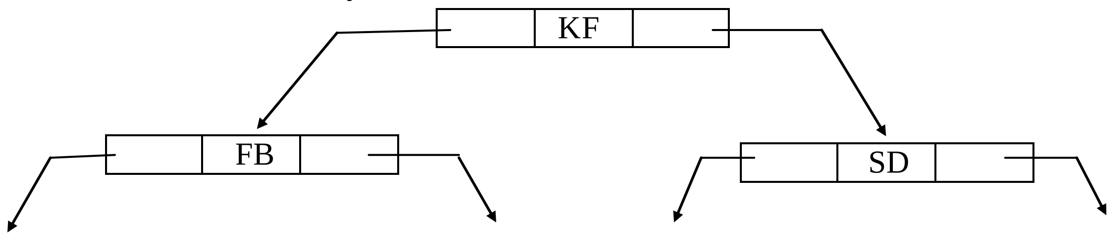
5.3 Indexing with Binary Search Trees

Given the sorted list below we can express a binary search of this list as a binary search tree, as shown in the figure below.

AX, CL, DE, FB, FT, HN, JD, KF, NR, PA, RF, SD, TK, WS, YJ



Using elementary data structure techniques, it is a simple matter to create node that contains right and left link fields so the binary search tree can be constructed as a linked structure.



The figure belov $\boxed{\text{ROOT}} \rightarrow 9$ cord (:nts for a linked representation of the above binary tree.

	Key	Left Child	Right Child
0	FB	10	8
1	JD		
2	RF		
3	SD	6	13
4	AX		
5	YJ		
6	PA	11	2
7	FT		

	Key	Left Child	Right Child
8	HN	7	1
9	KF	0	3
10	CL	4	12
11	**NR**	**15**	**--**
12	DE		
13	WS	14	5
14	TK		
15	***LV***		

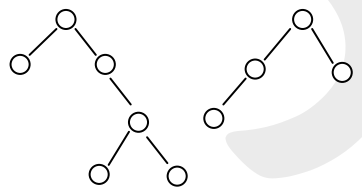
The records in the file illustrated in the figure above appear in random rather than sorted order. The sequence of the records in the file has no necessary relation to the structure of the tree: all the information about the logical structure is carried in the link fields. The very positive consequence that follows from this is that if we add a new key to the file such as LV, we need only link it to the appropriate leaf node to create a tree that provides search performance that is as good as we would get with a binary search on a sorted list. (Showed in Bold)

5.3.1 AVL Trees (G. M Adel'son, Velskii, E. M Landis)

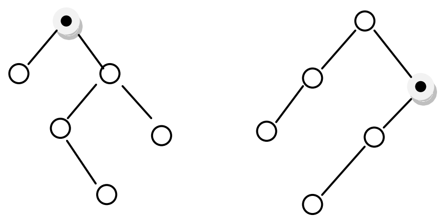
A binary tree which maintains height balance (to HB(1)) by means of localized reorganizations of the nodes.

- No two subtree's height of any root differs by more than one level.
- AVL trees maintain *HB(1)* with local rotations of the nodes.
- AVL trees are not perfectly balanced.
- Worst case search with an AVL tree is 1.44 $\log_2 (N + 2)$ compares.

Examples of AVL trees:



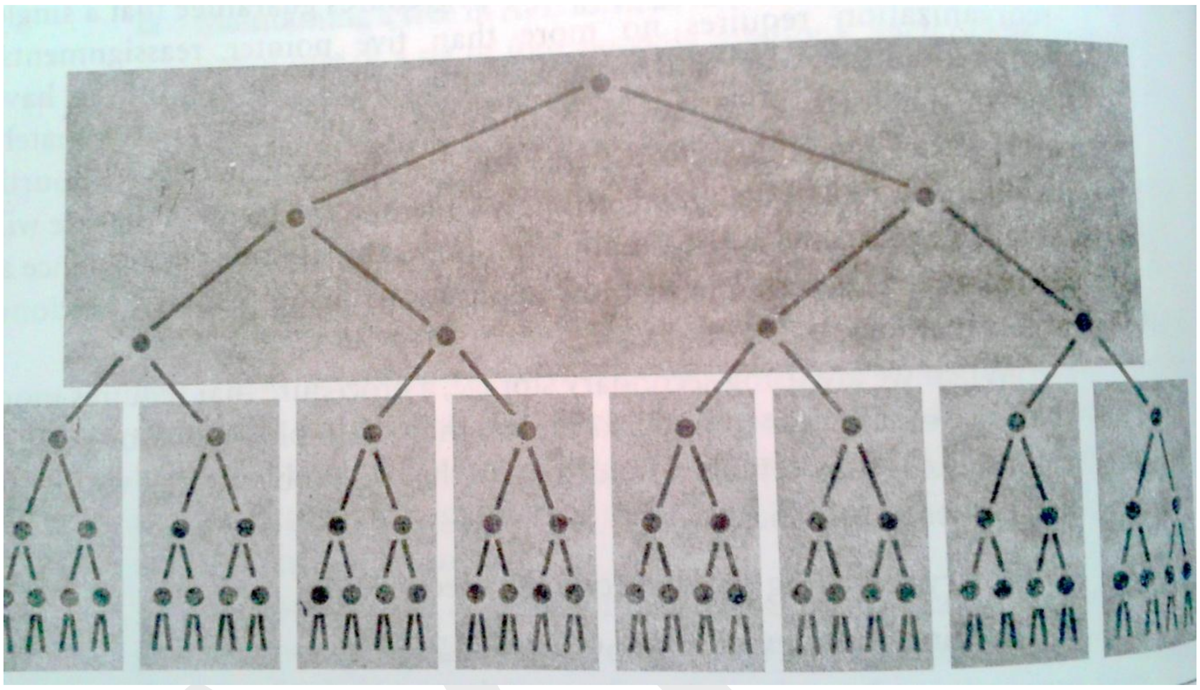
Examples of Non AVL Tress:

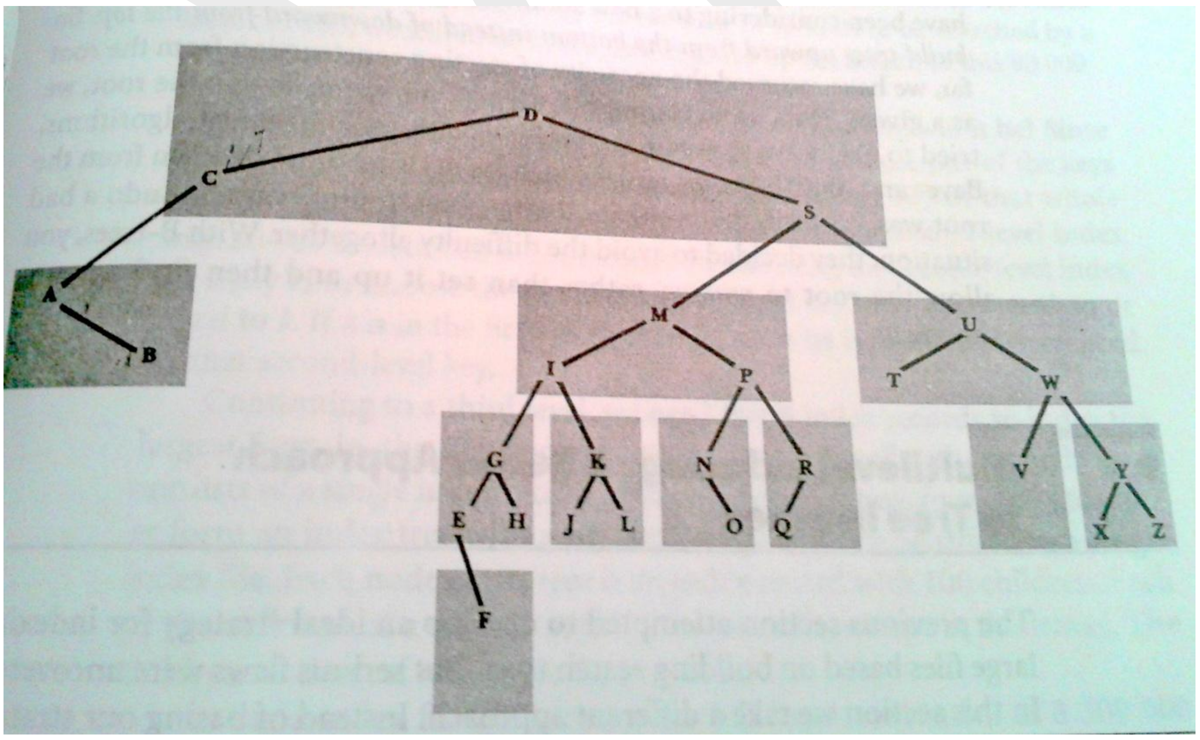


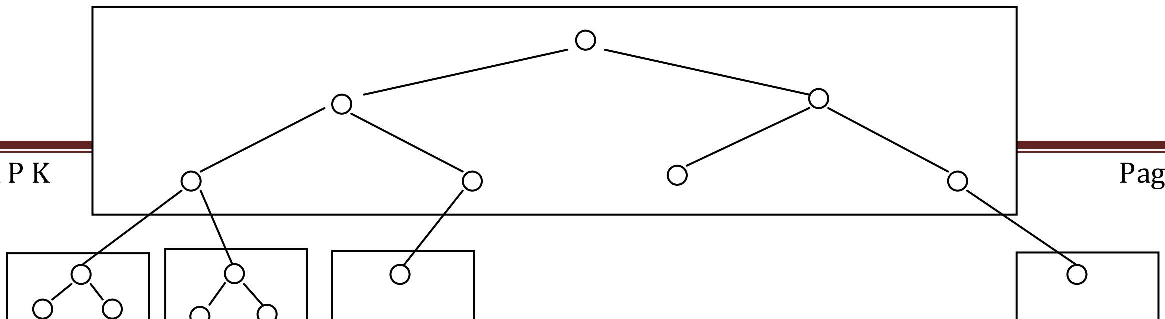
5.3.2 Paged Binary Trees

Disk utilization of a binary search tree is extremely inefficient. That is when a node of binary search tree is read, there are only three useful information. They are the key value, the address of the left and right sub trees. Each disk read produces a minimum of single page. The paged binary tree attempts to address this problem by locating multiple binary nodes on the same disk page. Paging divides a binary tree in to pages and then storing each page in a block of contiguous locations on disk, so that reduces the number of seeks associated with search.

- Worst case search with a balanced paged binary tree with page size M is $\log_{M+1}(N+1)$ compares.
- Balancing a paged binary tree can involve rotations across pages, involving physical movement of nodes.







○

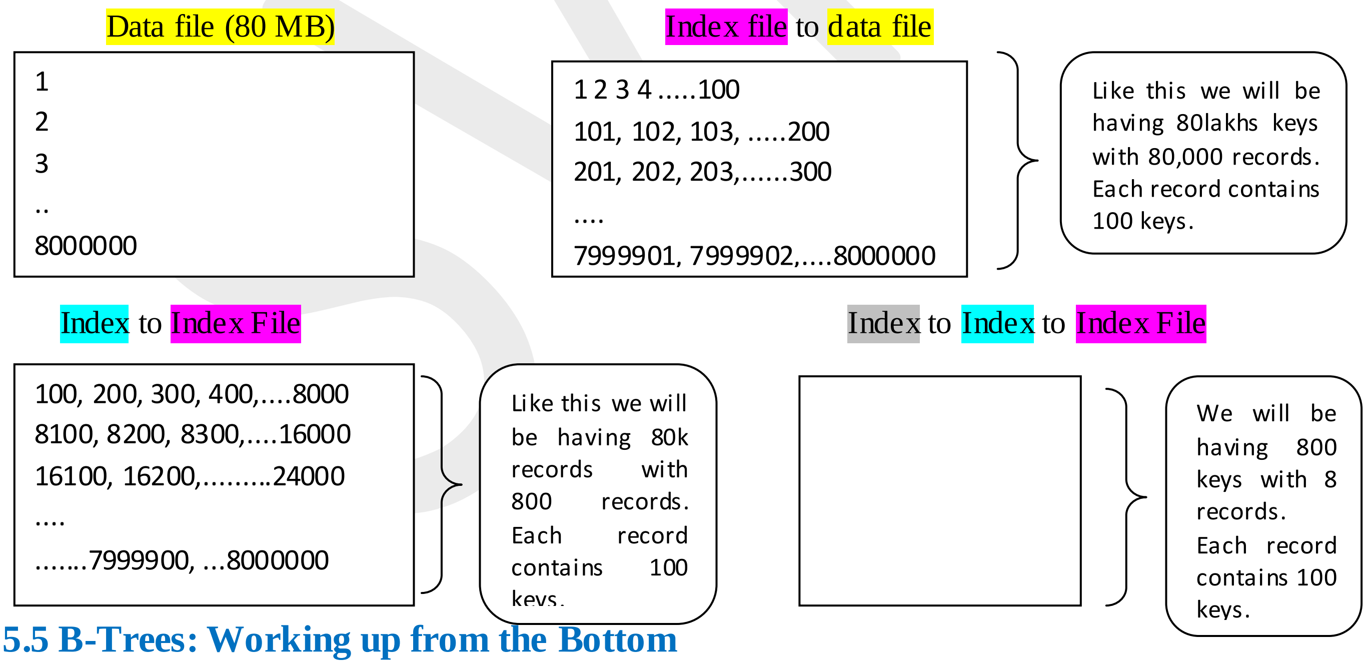
The Problem with Paged Binary Trees

- ➢ Only valid when we have the entire set of keys in hand before the tree is built.
- ➢ Problems due to out of balance.

5.4 Multilevel Indexing: A Better Approach to Tree Indexes

Consider a 80 MB file which has 80,00,000 (80 lakhs) records. Each record is 100 bytes & each key is 10 – byte keys. An index of this file has 80,00,000 key – reference pairs divided among a sequence of index records. Let's suppose that we can put 100 key – reference pairs in a single index record. Hence there are 80,000 records in the index.

The 100 largest keys are inserted into an indexed record, and that record is written to the index file. The next largest 100 keys go into the next record of the file, and so on. This continues until we have 80,000 index records in the index file. We must find a way to speed up the search of this 80000 record file. So we construct an index to index file. Then an index to index to index file is constructed. This is as shown in the below figures.



Data file (80 MB)

1
2
3
..
8000000

Index file to data file

1 2 3 4 100
101, 102, 103, ...200
201, 202, 203,......300
....
7999901, 7999902,....8000000

Like this we will be having 80lakhs keys with 80,000 records. Each record contains 100 keys.

Index to Index File

100, 200, 300, 400,....8000
8100, 8200, 8300,....16000
16100, 16200,.........24000
....
.......7999900, ...8000000

Like this we will be having 80k records with 800 records. Each record contains 100 keys.

Index to Index to Index File

We will be having 800 keys with 8 records. Each record contains 100 keys.

5.5 B-Trees: Working up from the Bottom

B – Trees are multileveled indexes that solve the problem of linear cost of insertion and deletion.

In B – trees the overflow record is split into two records, each half full. Deletion takes a similar strategy of merging two records into a single record when necessary. Each node of a B – tree is index record.

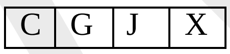
Each record has same maximum number of key reference pairs, called the order of the B – tree. The records also have a minimum number of key-reference pairs, typically half of the order.

- An attempt to insert a new key into an index record that is not full is simply done by updating the index record.
- If the new key is the new largest key in the index record, it is the new higher level key of that record, and the next higher level of the index must be updated.
- When insertion into an index record causes it to be overflow, it is split into two records, each with half of the keys. Since a new index node has been created at this level, the largest key in this new node must be inserted into the next higher level node. This is called promotion of the key.
- Sometimes promotion of key may cause an overflow at that level. This in turn causes that node to be split, and a key promoted to the next level. This continues as far as necessary. This causes another level to be added to the multilevel index.

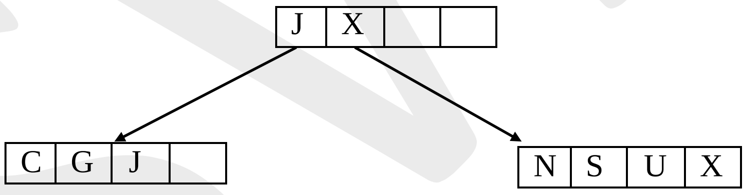
5.6 Example of Creating a B – Tree. (Also refer to the example solved in the class).

Construct a B – Tree of order 4, for the following set of keys
C G J X N S U O A E B H I F K L Q R T V

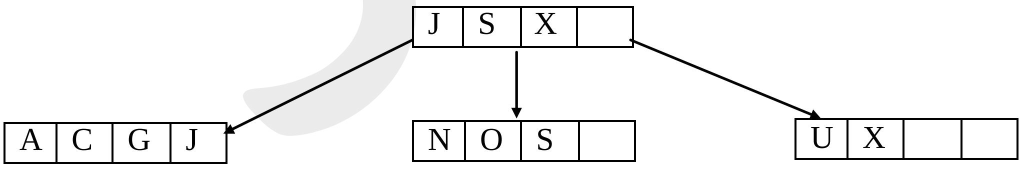
Step 1: After Insertion of C G J X



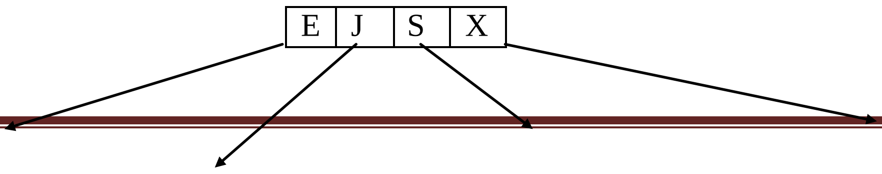
Step 2: After Insertion of N S U

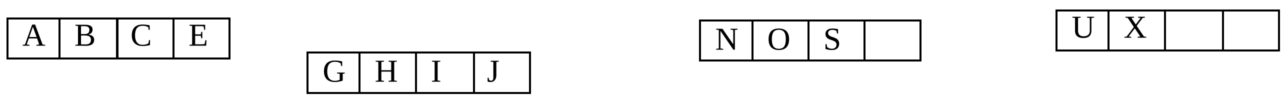


Step 3: After insertion of O A

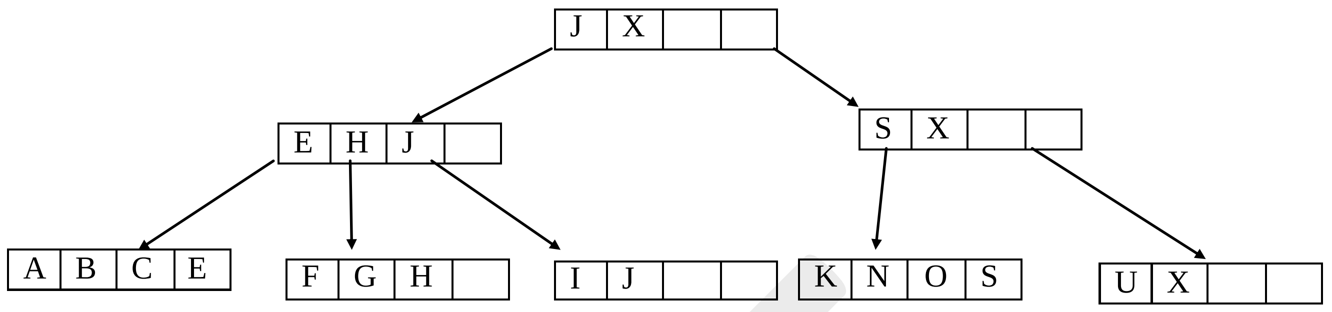


Step 4: After insertion of E B H I

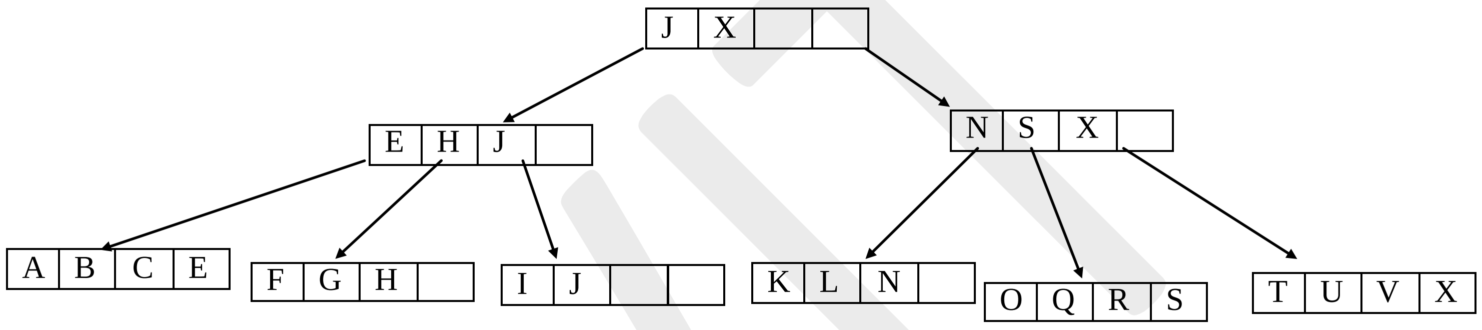




Step 5: After insertion of F K



Step 6: After insertion of L Q R T V



5.7 B – Tree Nomenclature.

- Order of a B – tree:
    - ➢ According to **Bayer, Mc Creight & Comer** order of the B – tree is **minimum** number of **keys** that can be in a page of the tree.
    - ➢ **Knuth** definition of order of the B- tree is **maximum** number of **descendants** that a page can have.
- LEAF
    - ➢ **Bayer** and **Mc Creight** refer to the **lowest** level of keys in a B – tree as the leaf level.
    - ➢ **Knuth** considers the leaf of a B – tree to be **one level below the lowest level** of keys. In other words, he considers the leaves to be the actual data records that might be pointed to by the lowest level of the keys in the tree.

5.8 Formal Definition of B-Tree Properties

For a B-Tree of Order m,

Every page has a maximum of m descendants.

- Every page, except for the root and the leaves, has a minimum of $\lceil m/2 \rceil$ descendants.
- The root has a minimum of 2 descendants (unless it is a leaf.)
- All of the leaves are on the same level.
- The leaf level forms a complete, ordered index of the associated data file.

5.9 Worst Case Search Depth

The maximum number of disk accesses required to locate a key in the tree is nothing but worst case search. Every key appears in the leaf level. So finding the depth of the tree is nothing but worst case search. The worst case occurs when every page of the tree has only the minimum number of descendants. In such case the keys are spread over a maximal height for the tree and a minimal breadth.

For a B – tree of order ‘m’ the minimum number of descendants from the root page is two, so the second level of the tree contains only two pages. Each of these pages in turn has at least ceiling function of m/2 given by “ $\lceil m/2 \rceil$ ” descendants. The general pattern of the relation between depth and the minimum number of descendants takes the following form:

Level	Minimum Number of Descendants
1 (root)	2
2	$2^* \lceil m/2 \rceil$
3	$2^*(\lceil m/2 \rceil)^2$
4	$2^*(\lceil m/2 \rceil)^3$
d	$2^*(\lceil m/2 \rceil)^{d-1}$

So in general for any level d of B – tree, the minimum number of descendants extending from that level is

$$2^*(\lceil m/2 \rceil)^{d-1}$$

For a tree with ‘N’ keys in its leaves, we express the relationship between keys and the minimum height d as

$$N \geq 2^* \lceil m/2 \rceil^{d-1}$$

Solving for d we arrive at the expression:

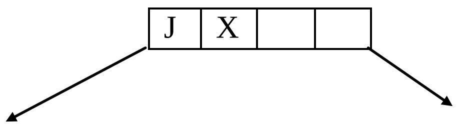
$$d \leq 1 + \log_{\lceil m/2 \rceil} (N/2)$$

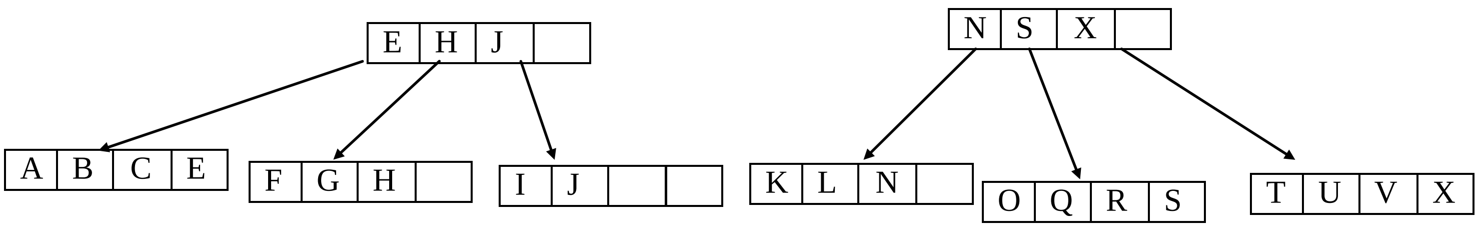
Problem based on the depth calculation refer class notes.

5.10 Deletion merging and redistribution

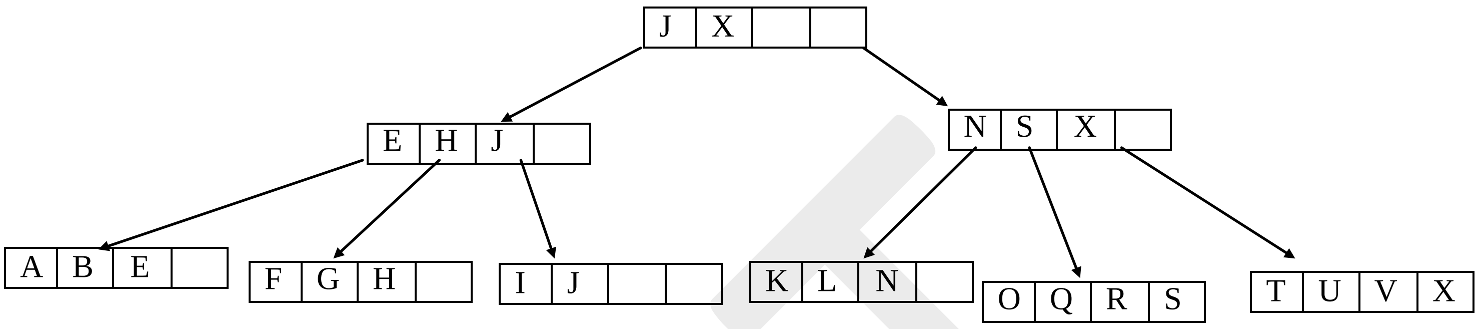
Deletion of a key can result in several different situations. Consider the B – tree below. What happens when we delete some of its keys.

The simplest situation is illustrated by deleting key ‘C’

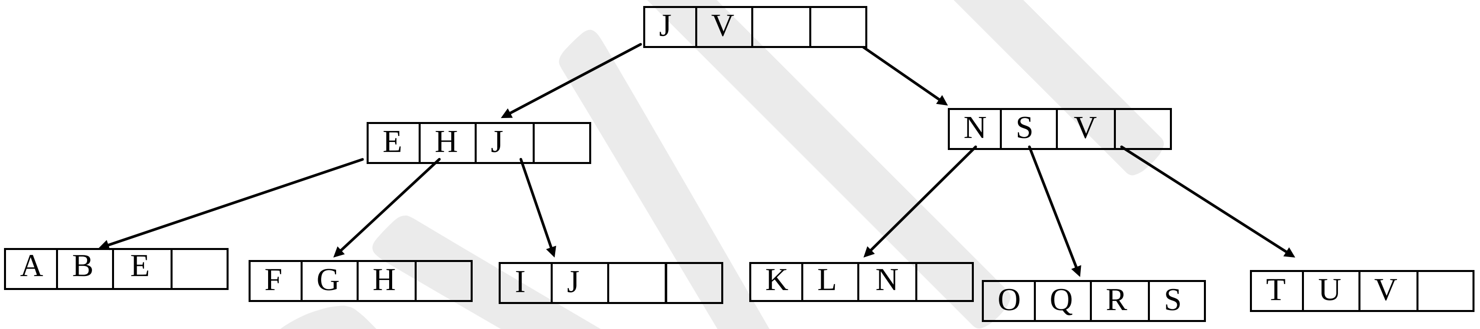




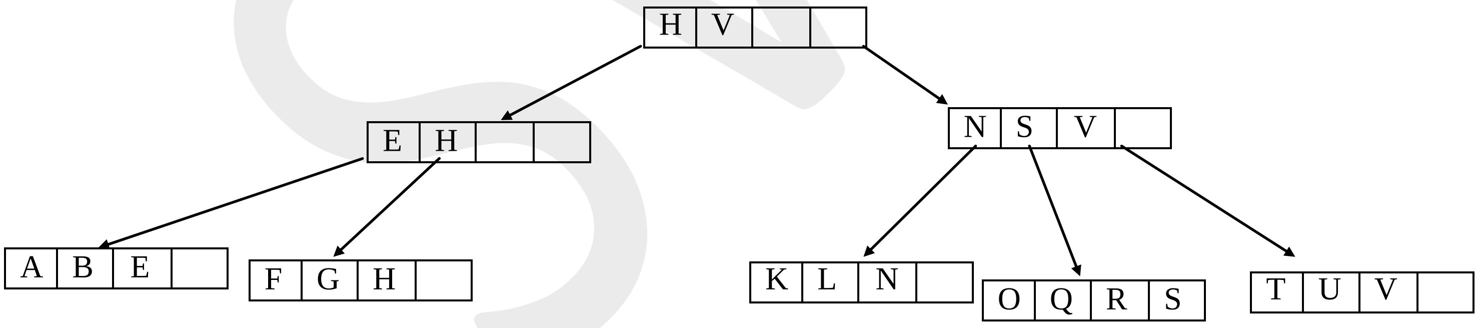
Removal of key ‘C’, change occurs only in leaf node.



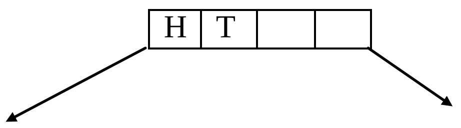
Removal of key ‘X’ change occurs in level 2 which needs to be updated in level 1 also.

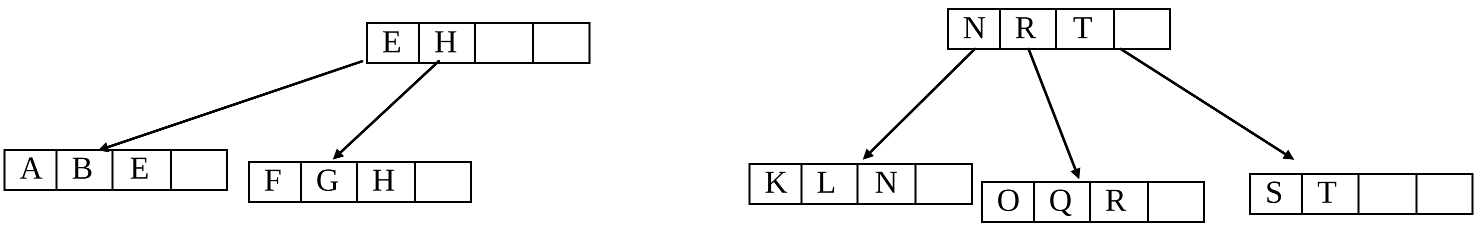


Removal of key ‘J’ causes underflow of that leaf. So it is merged.



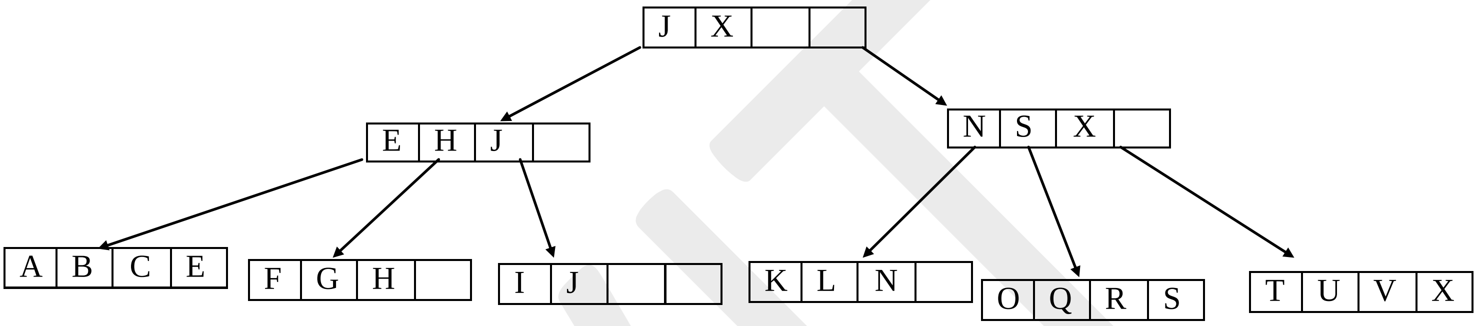
Deletion of keys ‘U’ & ‘V’ causes the right most leaf node to underflow. Here redistribution of keys among the nodes needs to be done. This redistribution leads to be updated in level 2 which in turn needs to be updated in level 1.



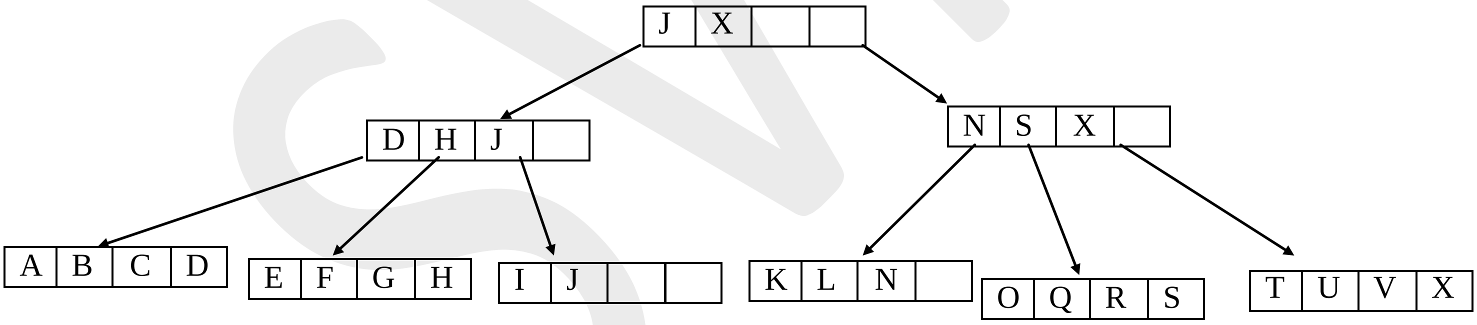


5.11 Redistribution during Insertion: A Way to Improve Storage Utilization

Redistribution during insertion is a way of avoiding, or at least postponing, the creation of new pages. Rather than splitting a full page and creating two approximately half-full pages, redistribution lets us place some of the overflowing keys in to another page. This indeed makes B-Tree more efficient in space utilization. For example consider the B - tree below



Now if key ‘D’ needs to be inserted, the left most page overflow and it needs to be split which results in one more leaf node added to B - Tree. Which results in approximately two half full pages. Instead of splitting the nodes if we can redistribute the keys among the nodes, we can postpone the splitting. The B – Tree below shows after insertion of key ‘D’ and redistribution.



5.12 Buffering of Pages: Virtual B – Trees

Consider an index file which is 1 MB of records and we have memory of 256 KB. Given a page size of 4 KB, holding around 64 keys per page, the B – tree can be contained in three levels. We may need at the max 2 disk seeks to retrieve any key from the B – Tree. To still improve on this we create a page – buffer to hold some number of B – tree pages, perhaps 5, 10 or more. As we read pages in from the disk in response to user request, we fill up the buffer. Then when a page is requested, we access it from the memory if we can, thereby avoiding a disk access. If the page is not in the memory, then we read it in to the buffer from secondary storage, replacing one of the pages that were previously there. A B- tree that uses a memory buffer in this way is sometimes referred to as a Virtual B – Tree. The page replacement algorithms are:

- LRU- Least recently used.
- Replacement based on the page height.—Here we always retain the pages that occur at the highest levels of the tree.