B-Tree

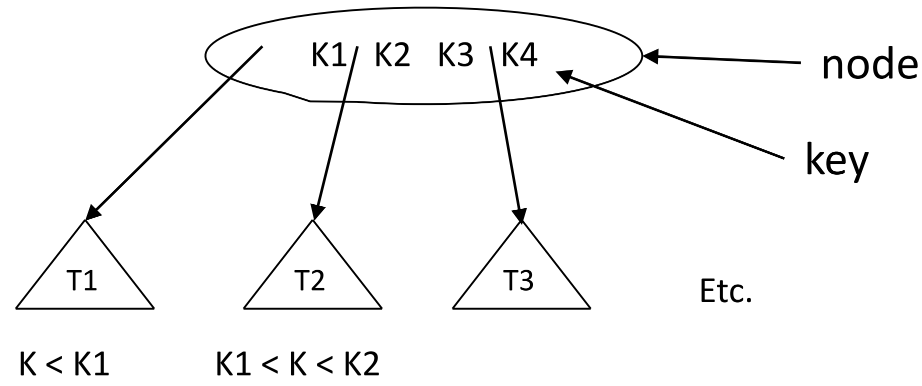
Motivation

- So far, we have assumed that we can store an entire data structure in the main memory (**RAM**) of a computer.
- When **data is too large to fit in main memory**, the data structure must resides on disk and the number of disk accesses becomes important.
- A **disk access is expensive** compared to a typical computer instruction.
- Although disks are cheaper and have higher capacity than main memory, they are much, much **slower** because they have moving **mechanical** parts compared to the purely electronic media.
- **One** disk access is worth **thousands** of instructions.
- The number of disk accesses will dominate the running time.

Motivation Cont..

- Secondary memory (disk) is divided into equal-sized **blocks** (typical sizes are 512, 2048, 4096 or 8192 bytes)
- The basic **I/O** operation transfers the contents of one disk block to/from main memory.
- Our goal is to devise a multiway search tree that will **minimize** file accesses.

m-ary Trees



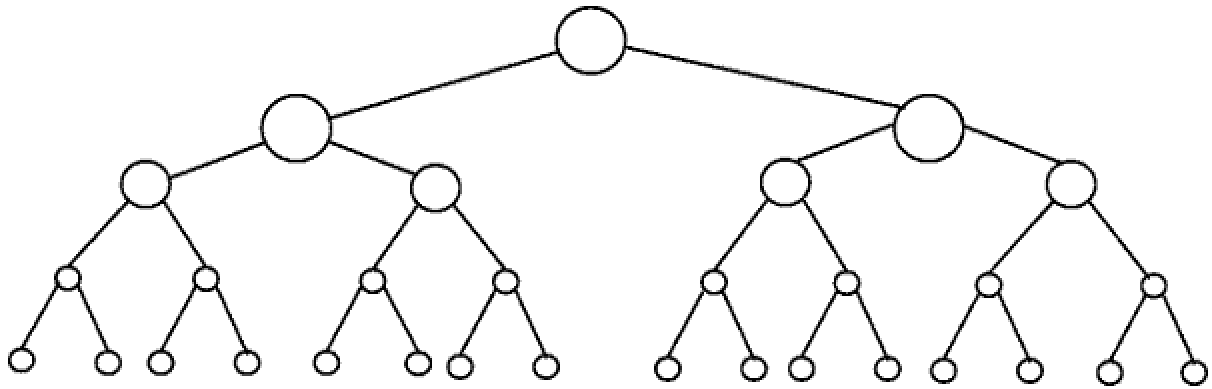
- m-ary search tree allows **m-way branching (m children)**.
- A **node** in a m-tree contains **multiple keys** (K1,K2,K3 etc)**.**
- Each piece of data stored is called a "**key**", because each key is unique and can occur in only one location.
- The **keys** in a node serve as **dividing points** separating the range of keys.
- In applications there would be a record of data associated with each key (e.g, student record, phone number).
- Order of subtrees is based on parent node keys.
- If each node has **m children** and there are **n keys** then the average time taken to search, insert and delete from the tree is $\mathbf{\log_m n}$**.**

B-Tree

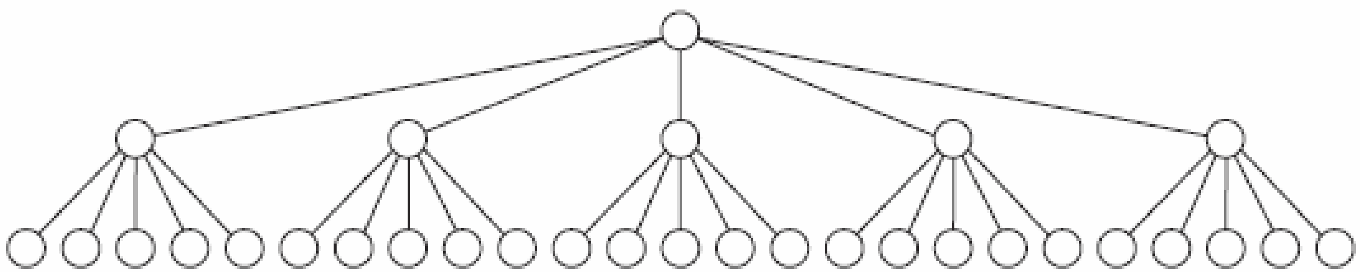
- Ideally, a tree will be balanced and the height will be log n where n is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a **balanced tree** structure like a AVL tree, b-tree, or red-black tree must be used.
- **B-tree** is a self-balancing tree data structure that keeps data **sorted** and allows **searches**, sequential access, insertions, and deletions in logarithmic time.
- The B-tree is a **generalization of a binary search tree** in that a node can have more than two children.
- The idea is that we leave some **key spaces open.** So an insert of a new key is done using available space (most cases).
- Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write **large blocks of data.**

- The B-tree algorithm **minimizes** the number of times a medium must be accessed to locate a desired record, thereby **speeding** up the process.
- The B-tree algorithms **copy selected pages from disk** into main memory as needed and write back onto disk the pages that have changed. B-tree algorithms keep only a constant number of pages in main memory at any time; thus, the size of main memory does not limit the size of B-trees that can be handled.
- In a practical B-tree, there can be thousands, millions, or **billions** of records.
- Real-world B-trees are of higher order (32, 64, 128, or more). For a large B-tree stored on a disk, we often see **branching factors** between 50 and 2000.

- A B-tree **node** is usually as large as a whole **disk page**, and this size limits the number of children a B-tree node can have.
- B-trees are a good example of a data structure for external memory. It is commonly used in **databases** and **filesystems**. (e.g., Used in Mac, NTFS, OS2 for file structure. SQL Server https://technet.microsoft.com/en-us/library/ms177443(v=sql.105).aspx).
- Real world database indexes with millions of records have a **tree depth of four or five**.

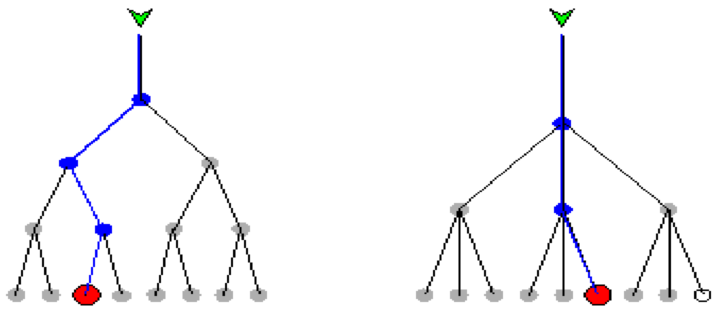


Binary tree of 31 nodes has 5 levels



A 5-ary tree of 31 nodes has only 3 levels

Binary Tree vs B-Tree

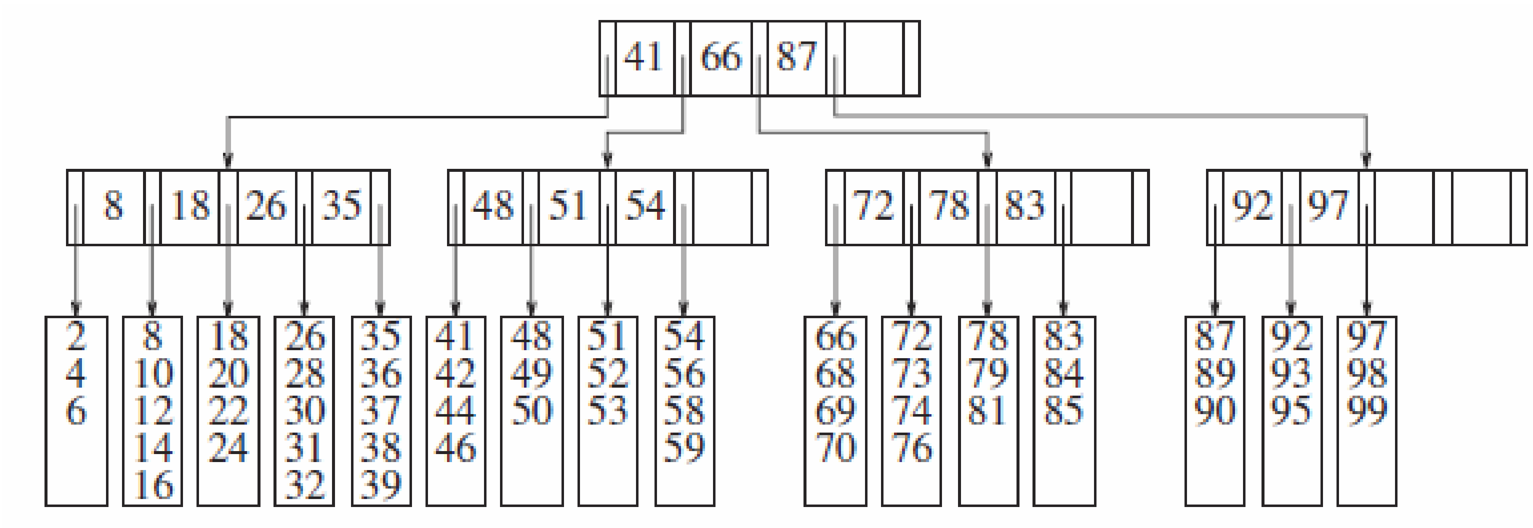


- **B-trees save time** by using nodes with **many branches**, compared with **binary trees**, in which each node has only two children. When there are many children per node, a record can be found by passing through fewer nodes than if there are two children per node.
- The tradeoff is that the **decision process** at each node is more complicated in a B-tree as compared with a binary tree. A program is required to execute the operations in a B-tree. But this program is stored in **RAM**, so it runs fast.

Properties of a B-Tree

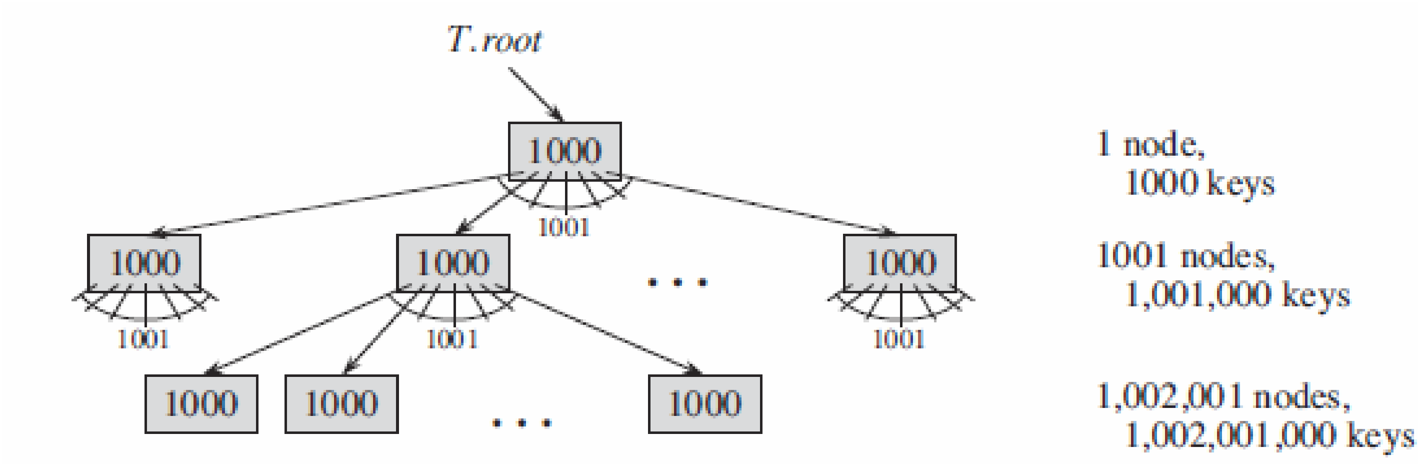
- **Order** of a tree = maximum number of **children** per node.
- B-tree of **order m** has the following properties:
  1. The root is either a leaf or has between 2 and m children.
  2. Every node has at most **m children (pointers)**.
  3. Every node except for the root and the leaves has **at least $\lceil m/2 \rceil$ children**.
  4. A non-leaf node stores up to **m-1 keys**.
  5. All leaves appear at the same level.

A B-tree of order 5



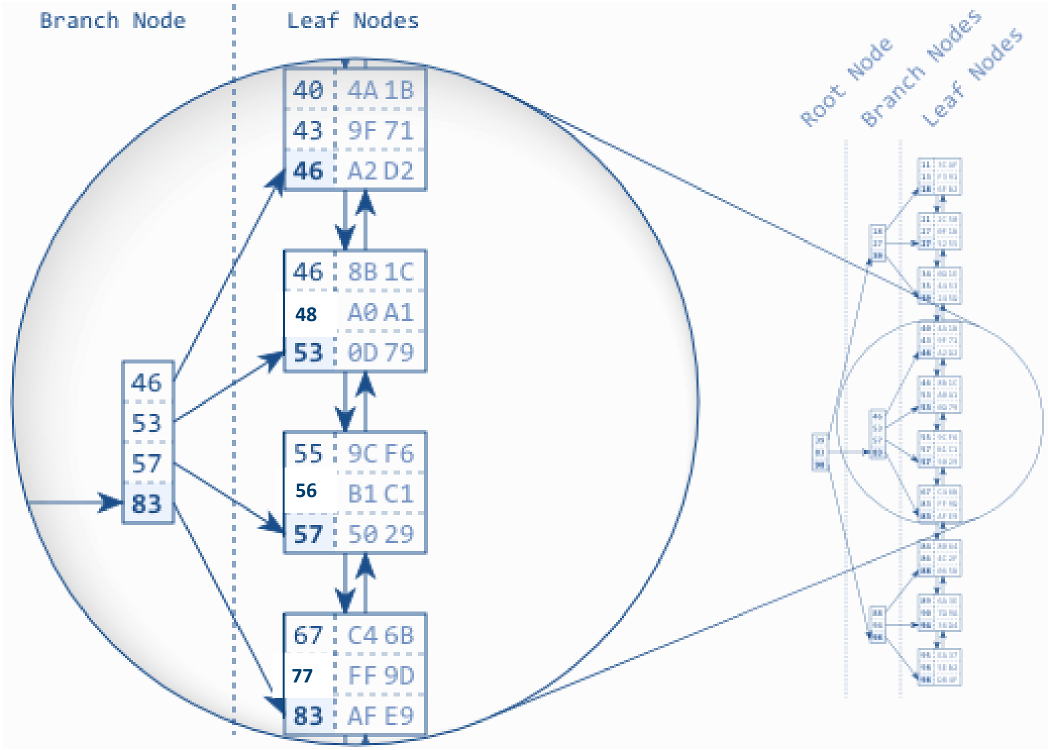
Notice that all non-leaf nodes have between 3 and 5 children (and thus between 2 and 4 keys).

How many keys a B-tree can store with a branching factor of 1001 and height 2?



- Max number of keys at depth d in a B-tree= $m^d * k$
  - m = order of tree
  - k = max number of keys per node
  - d=depth
- Since we can keep the **root node permanently in main memory**, we can find any key in this tree by making at most only two disk accesses.

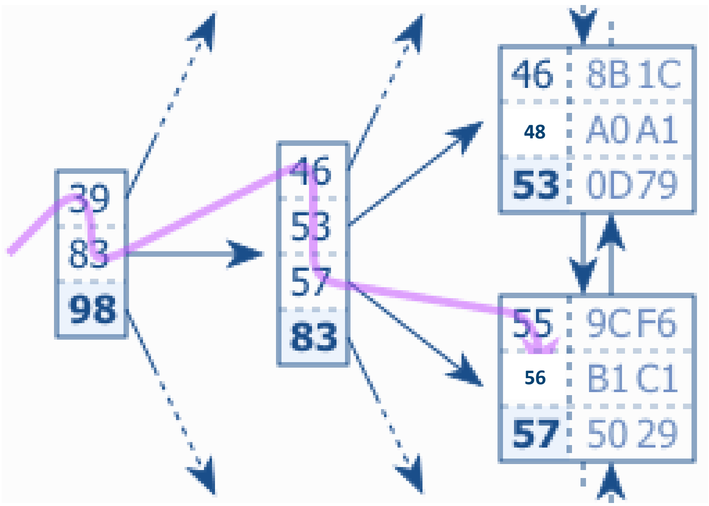
B-tree Structure



- An index with 30 entries.
- An implementation of B-Tree that is used in some databases: Each branch node entry corresponds to the biggest value in the respective leaf node.

B-Tree Traversal

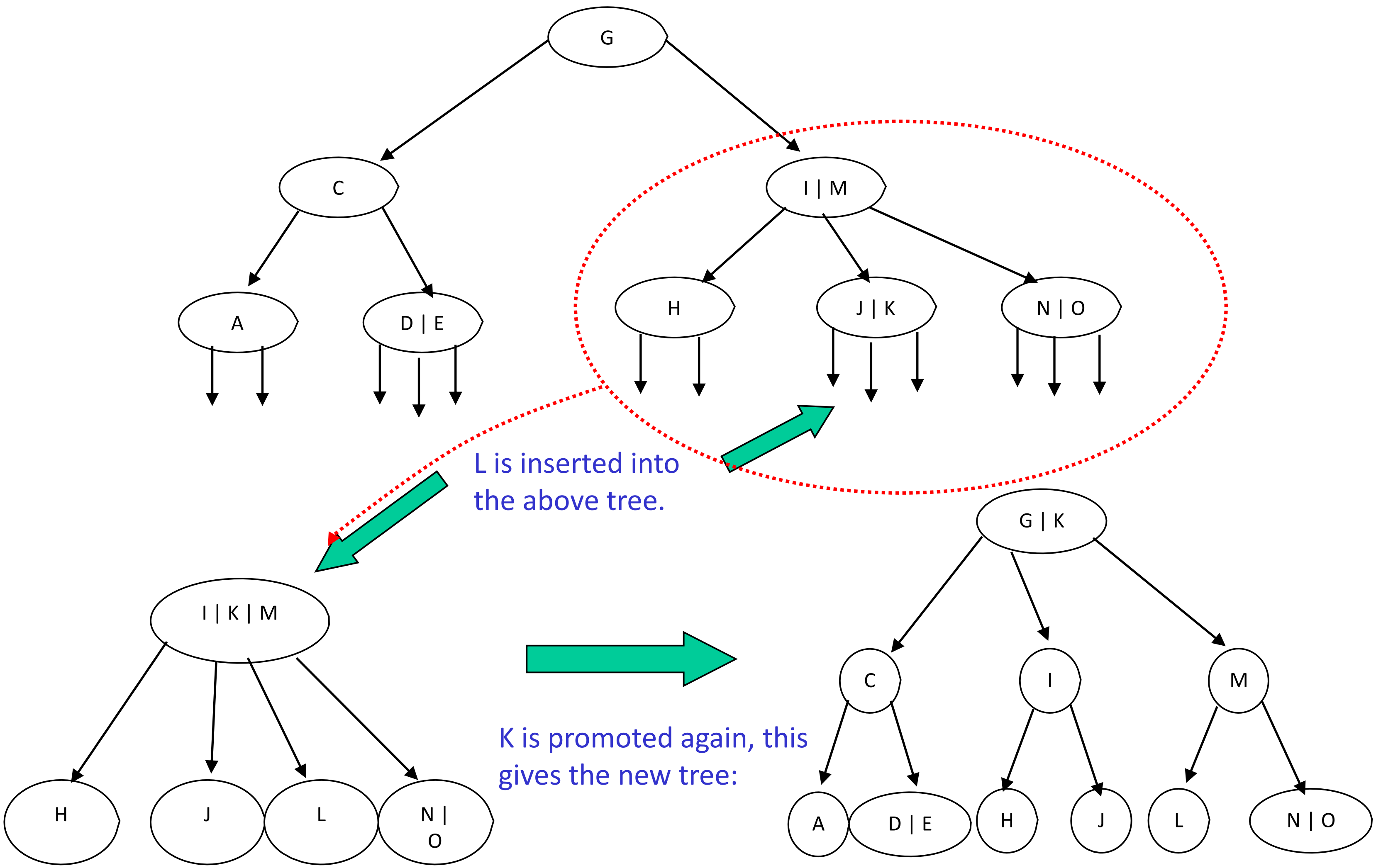
The figure shows an index fragment to illustrate a search for the key "**56**". The tree traversal starts at the root node on the left-hand side. Each entry is processed in ascending order **until a value is greater than or equal** to (>=) the search term (56). In the figure it is the entry 83. The database follows the reference to the corresponding branch node and repeats the procedure until the tree traversal reaches a leaf node.



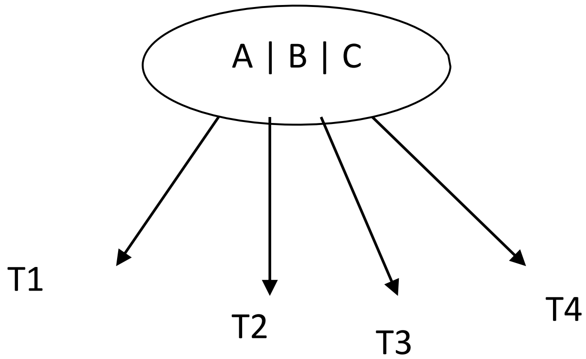
Insertion

- Insert k into B-tree of order m.
  - We find the insertion point (in a leaf) by doing a **search**.
  - **If there is room then enter k**, keeping the node's elements ordered.
  - Else, **promote the middle key** to the parent & **split** the node into 2 nodes around the middle key.
    - If the splitting backs up to the root, then make a new root containing the middle key.
- Note: the tree grows from the leaves, **balance is always maintained.**

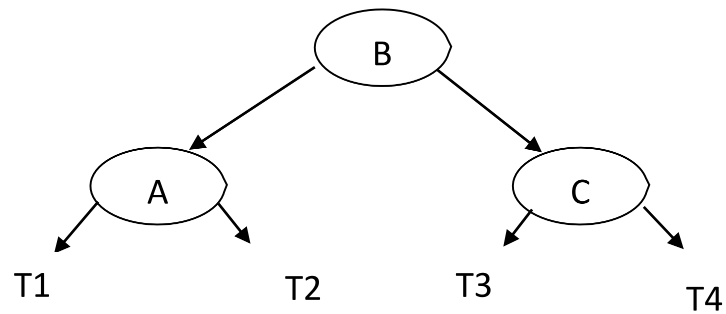
Insertion Example – B-Tree of order 3



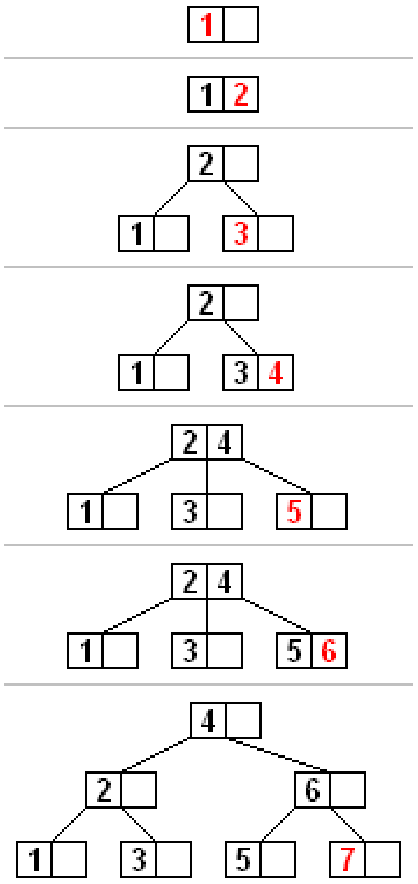
Splitting Nodes



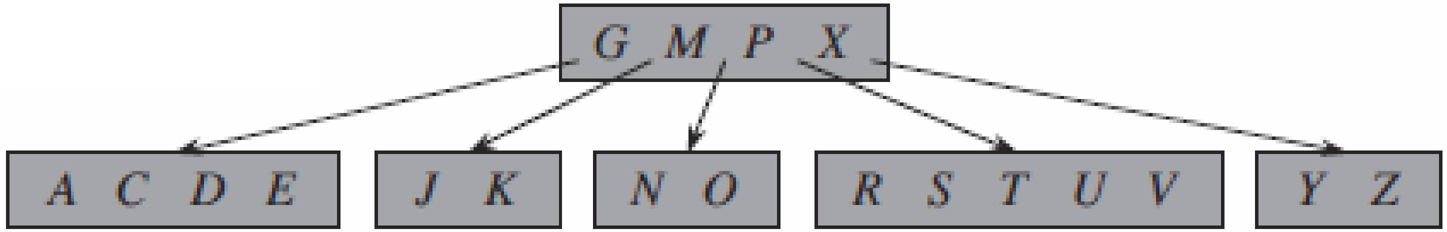
- Middle key is promoted
- Join its parent or create a new root

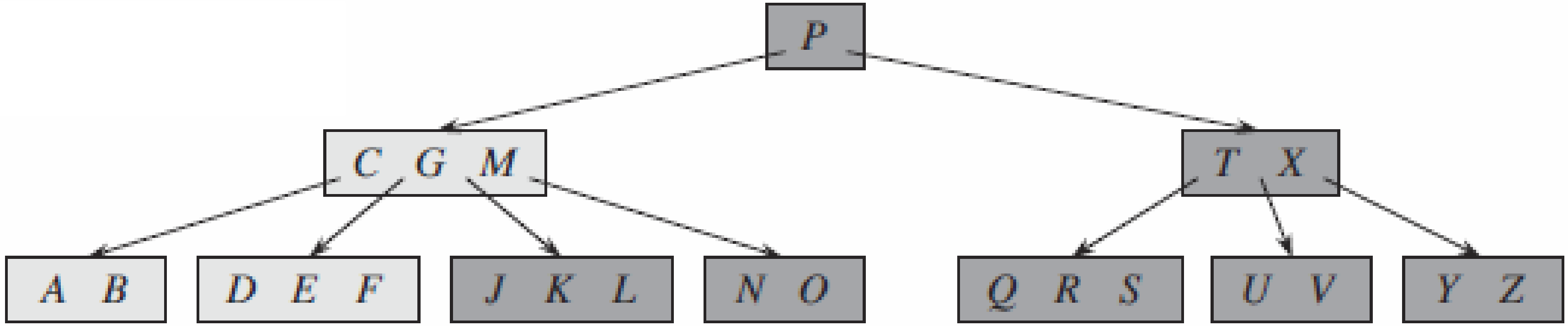


Insert the numbers 1 to 7 into a B-Tree of order 3



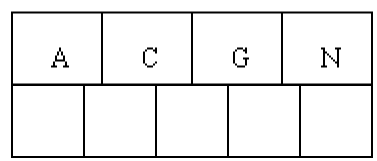
Show the final B-Tree after inserting keys B, Q, L, and F into the following B-tree of order 6



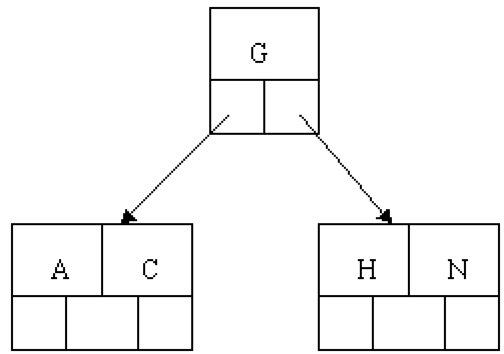


Insert the following letters into an empty B-tree of order 5:
C N G A H E K Q M F W L T Z D P R X Y S

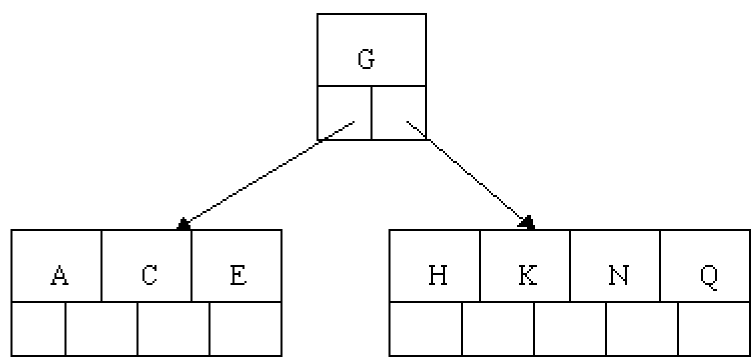
- Order 5 means that a node can have a maximum of 5 children and 4 keys.
- All nodes other than the root must have a minimum of 2 keys.
- The first 4 letters get inserted into the same node, resulting in this picture:



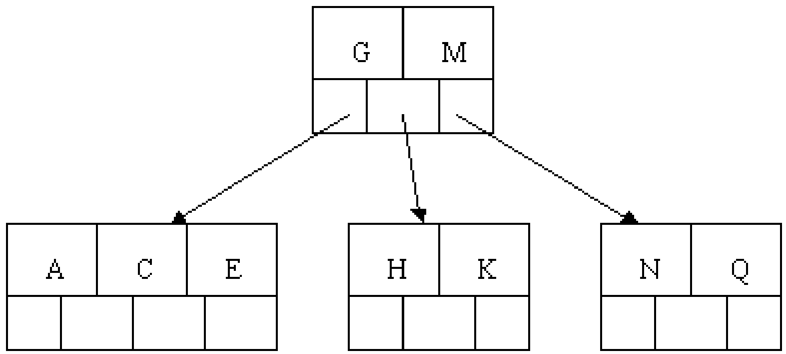
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node.



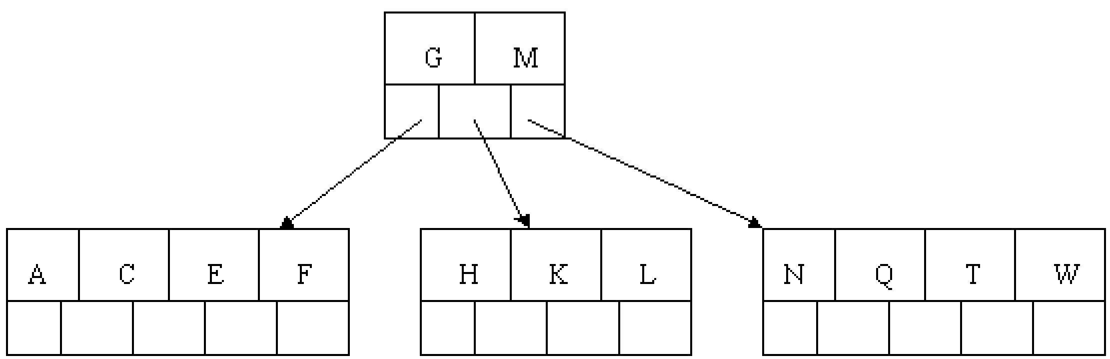
Inserting E, K, and Q proceeds without requiring any splits:



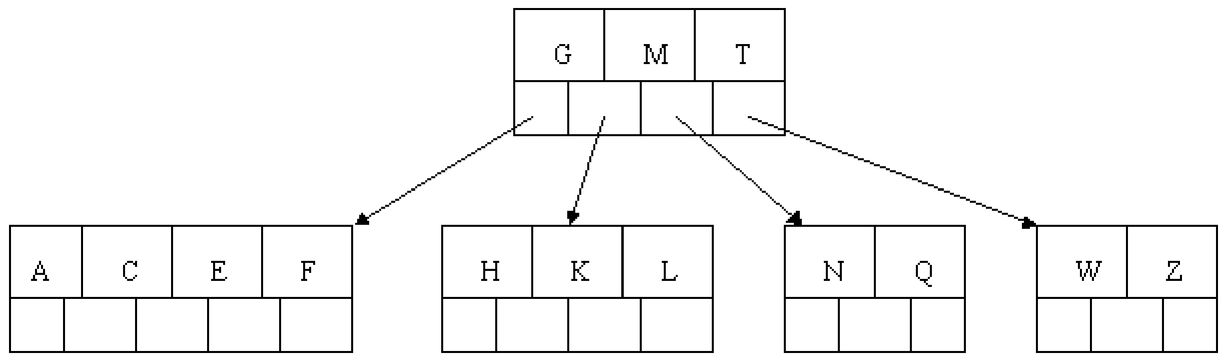
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



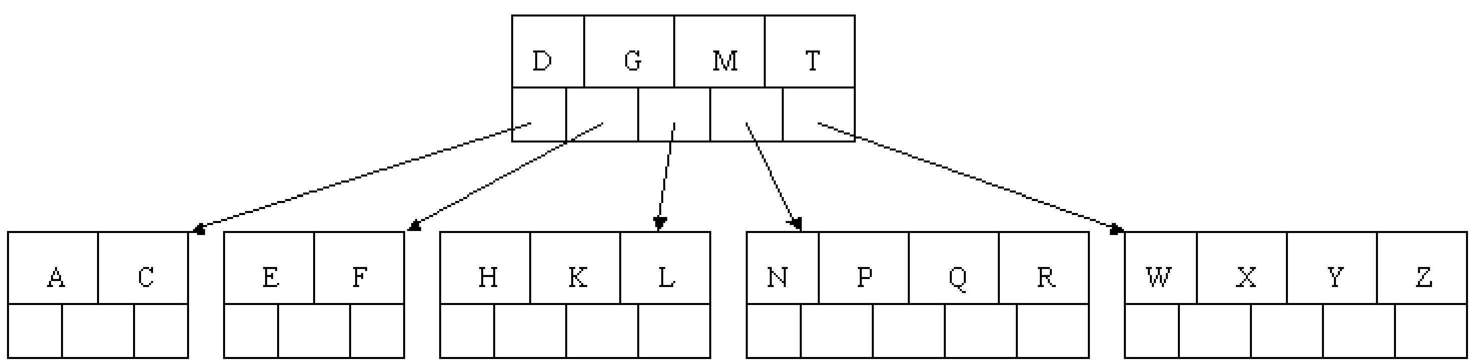
The letters F, W, L, and T are then added without needing any split.



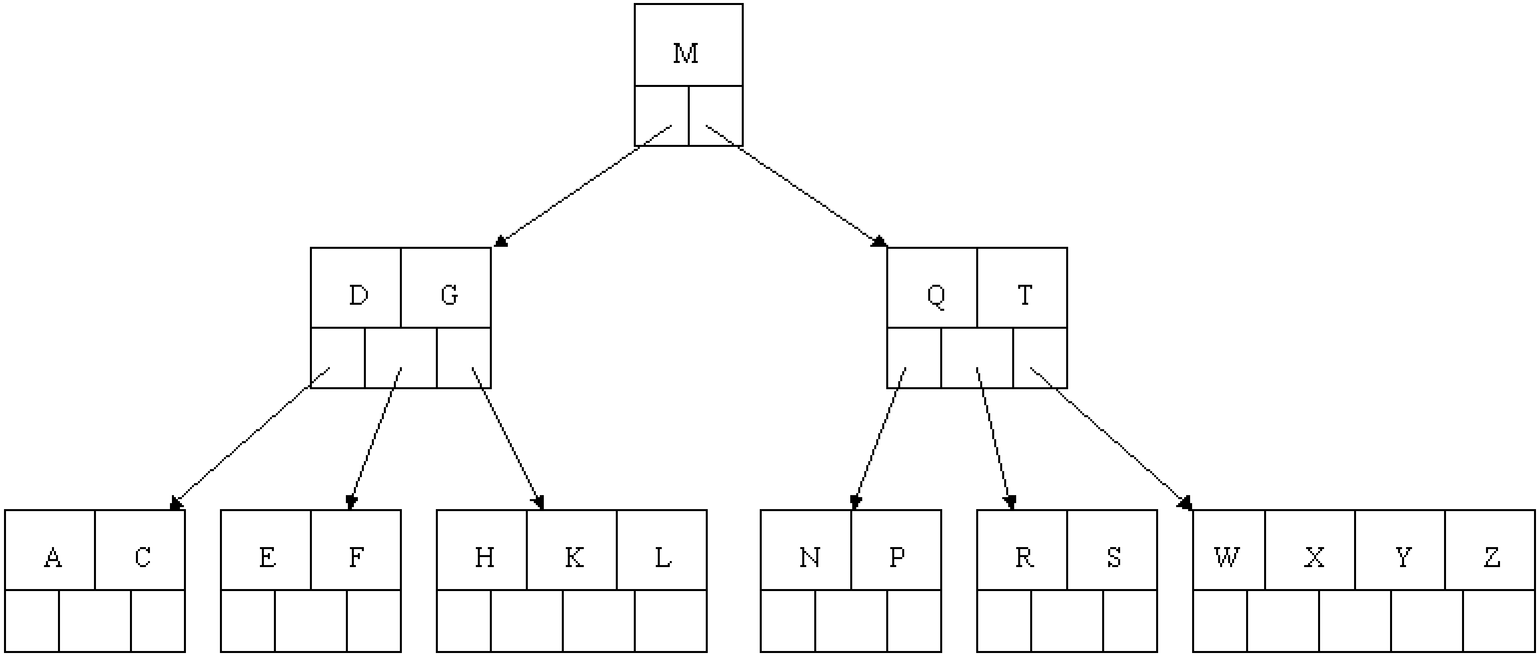
When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



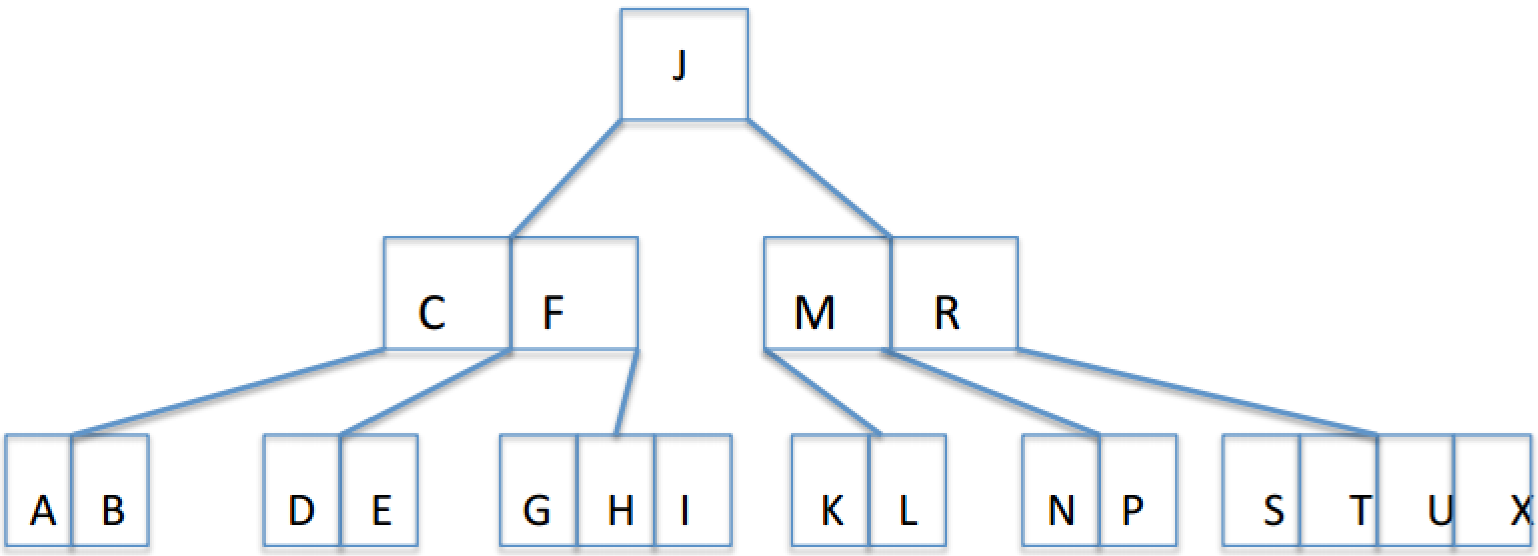
The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



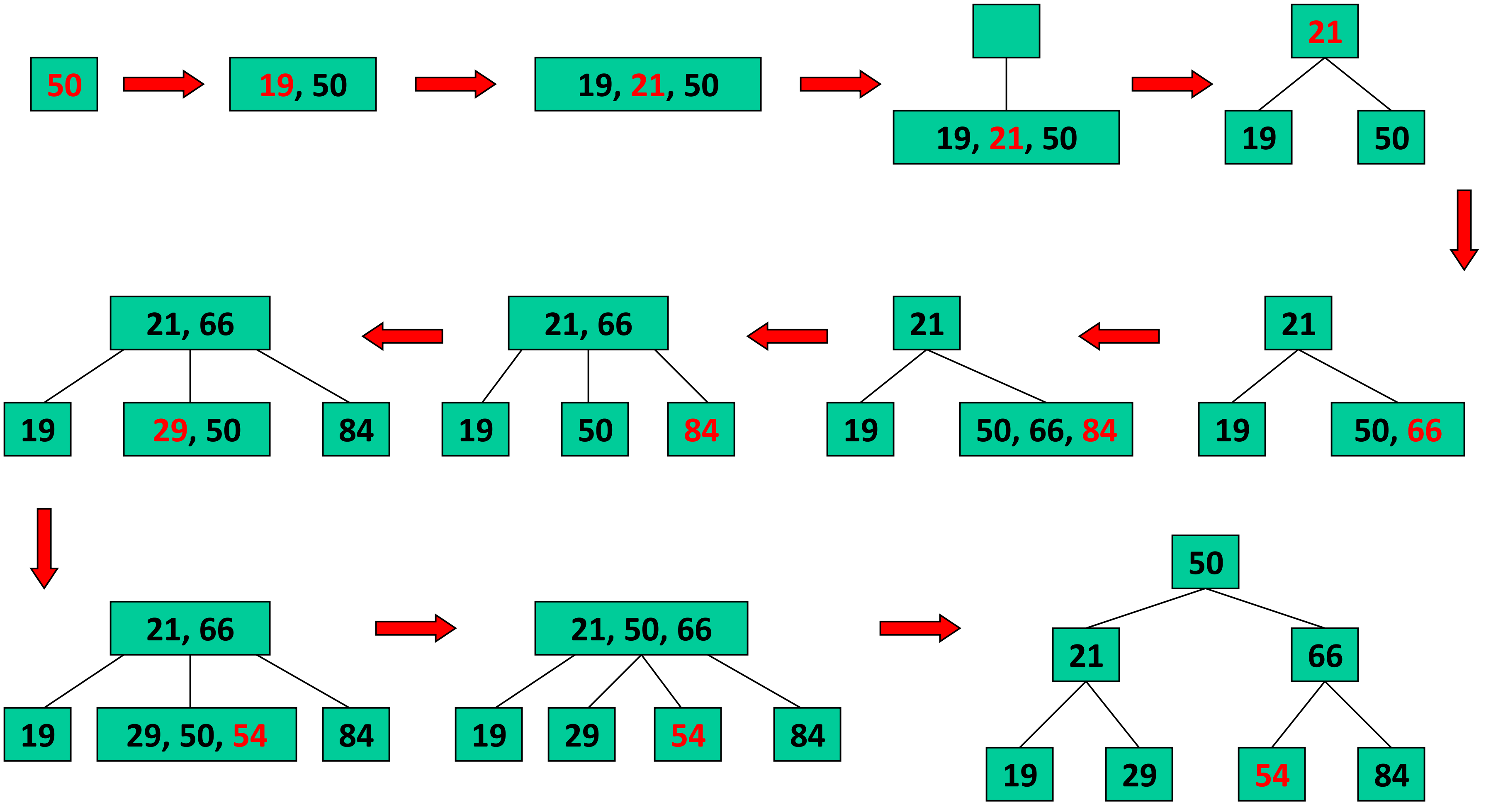
Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



Insert the following letters into an empty B-tree of order 5:
A G F B K D H M J E S I R X C L N T U P



Insert 50, 19, 21, 66, 84, 29, and 54 into a B-Tree with order 3



Useful links

- B-Tree applet https://www.cs.usfca.edu/~galles/visualization/BTree.html
- B-Tree talk
- http://www.csanimated.com/animation.php?t=B-tree