Data Structures and Algorithms

Lecture 12: B^+ Trees I

Department of Computer Science & Technology
United International College

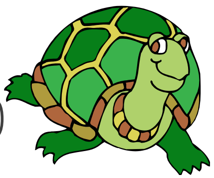
Agenda

- Why B^+ Trees?
- B^+ Tree Introduction
- B^+ Tree Operations
  - Search
  - Insertion
  - Deletion

Motivation

- An AVL tree with N nodes is an excellent data structure for searching, indexing, etc.
  - The Big-Oh analysis shows most operations finish within O(log(N)) time
- The theoretical conclusion works as long as the entire structure can fit into the main memory
- When the data size is too large and has to reside on disk, the performance of AVL trees may deteriorate rapidly

A Practical Example



- A 500-MIPS machine, with 7200 RPM hard disk
  - 500 million instruction executions, and approximately 120 disk accesses each second
- The machine is shared by 20 users
  - Thus for each user, can handle 120/20=6 disk access/sec
- A database with 10,000,000 items,
  - 256 bytes/item (assume it doesn't fit in main memory)
  - The typical searching time for one user
    - A successful search need $\log_2(10{,}000{,}000) = 24$ disk access,
    - Takes around 24/6=4 sec.
    - This is way too slow!!

- We want to reduce the number of disk access to a very small constant

From Binary to M-ary

- Idea: allow a node in a tree to have many children
  - Less disk access = smaller tree height = more branching
- As branching increases, the depth decreases
- An M-ary tree allows M-way branching
  - Each internal node has at most M children
- A complete M-ary tree has height that is roughly $\log_M(N)$ instead of $\log_2(N)$
  - if $M = 20$ then $\log_{20}(2^{20}) < 5$
  - Thus, we can speedup the search significantly

M-ary Search Tree

- Binary search tree has one key to decide which of the two branches to take
- M-ary search tree needs M-1 keys to decide which branch to take

- M-ary search tree should be balanced in some way too
  - We don't want an M-ary search tree to degenerate to a linked list, or even a binary search tree
    - Thus, require that each node is at least **half** full!

B^+ Tree

- A B^+-tree of order M (M>3) is an M-ary tree with the following properties:
  1. The data items are stored in leaves
  2. The root is either a leaf or has between two and M children
  3. The non-leaf nodes store up to M-1 keys to guide the searching; key *i* represents the smallest key in subtree *i+1*
  4. All non-leaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children
  5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data items, for some L (usually L << M, but we will assume M=L in most examples)

There are various definitions of B-trees that differ in minor ways. The above definition is one of the popular forms.

Keys in Internal Nodes

- Which keys are stored at the internal nodes?
  - There are several ways to do it. Different books adopt different conventions.

- We will adopt the following convention:
  - key *i* in an internal node is the smallest key in its *i+1* subtree (i.e. right subtree of key *i*)

- Even following this convention, there is no unique B^+-tree for the same set of records.

B+ Tree Example 1 (M=L=5)

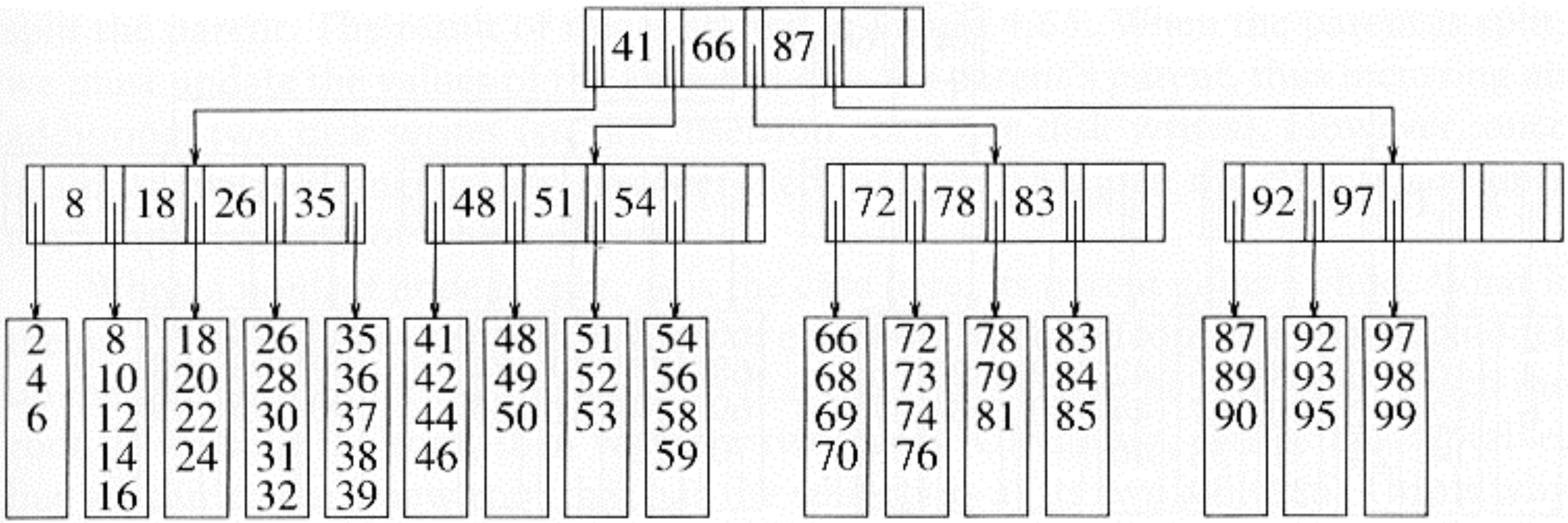
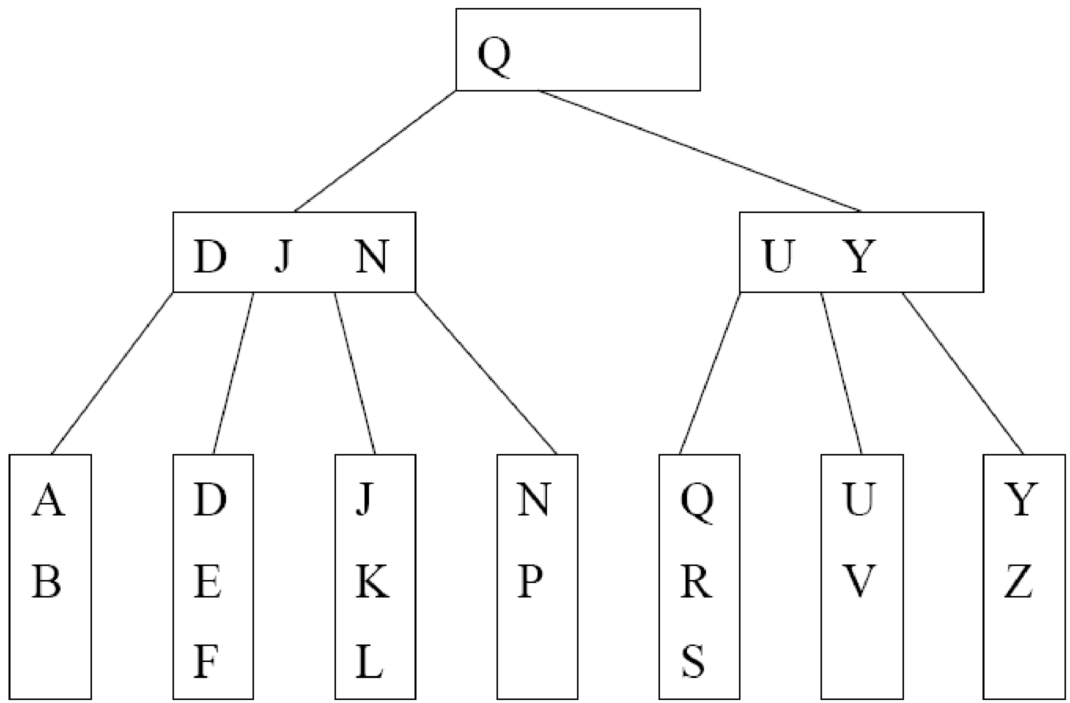


Figure 4.62 B-tree of order 5

- Records are stored at the leaves (we only show the keys here)
- Since L=5, each leaf has between 3 and 5 data items
- Since M=5, each nonleaf nodes has between 3 to 5 children

- Requiring nodes to be half full guarantees that the B+ tree does not degenerate into a simple binary tree

B+ Tree Example 2 (M=4,L=3)



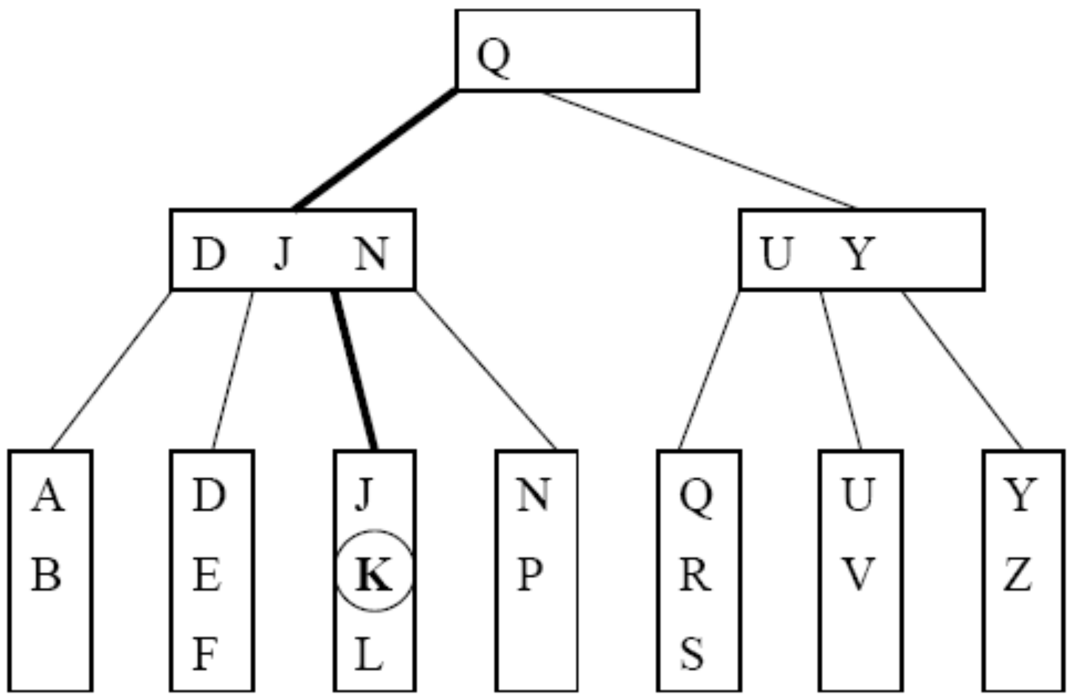
- We can still talk about left and right child pointers
- E.g. the left child pointer of N is the same as the right child pointer of J
- We can also talk about the left subtree and right subtree of a key in internal nodes

B+ Tree in Practical Usage

- Each internal node/leaf is designed to fit into one I/O block of data. An I/O block usually can hold quite a lot of data. Hence, an internal node can keep a lot of keys, i.e., large M. This implies that the tree has only a few levels and only a few disk accesses can accomplish a search, insertion, or deletion.
- B^+-tree is a popular structure used in commercial databases. To further speed up the search, the first one or two levels of the B^+-tree are usually kept in main memory.
- The disadvantage of B^+-tree is that most nodes will have less than M-1 keys most of the time. This could lead to severe space wastage. Thus, it is not a good dictionary structure for data in main memory.
- The textbook calls the tree B-tree instead of B^+-tree. In some other textbooks, B-tree refers to the variant where the actual records are kept at internal nodes as well as the leaves. Such a scheme is not practical. Keeping actual records at the internal nodes will limit the number of keys stored there, and thus increasing the number of tree levels.

Searching Example

- Suppose that we want to search for the key K. The path traversed is shown in bold



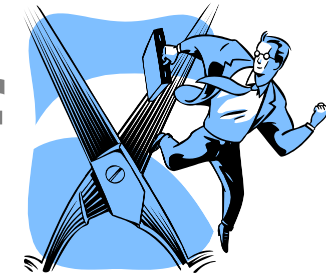
Searching Algorithm

- Let x be the input search key.
- Start the searching at the root
- If we encounter an internal node v, search (linear search or binary search) for x among the keys stored at v
  - If $x < K_{min}$ at v, follow the left child pointer of K_{min}
  - If $K_i \leq x < K_{i+1}$ for two consecutive keys K_i and K_{i+1} at v, follow the left child pointer of K_{i+1} (same as the right child pointer of K_i)
  - If $x \geq K_{max}$ at v, follow the right child pointer of K_{max}
- If we encounter a leaf v, we search (linear search or binary search) for x among the keys stored at v. If found, we return the entire record; otherwise, report not found.

Insertion Procedure

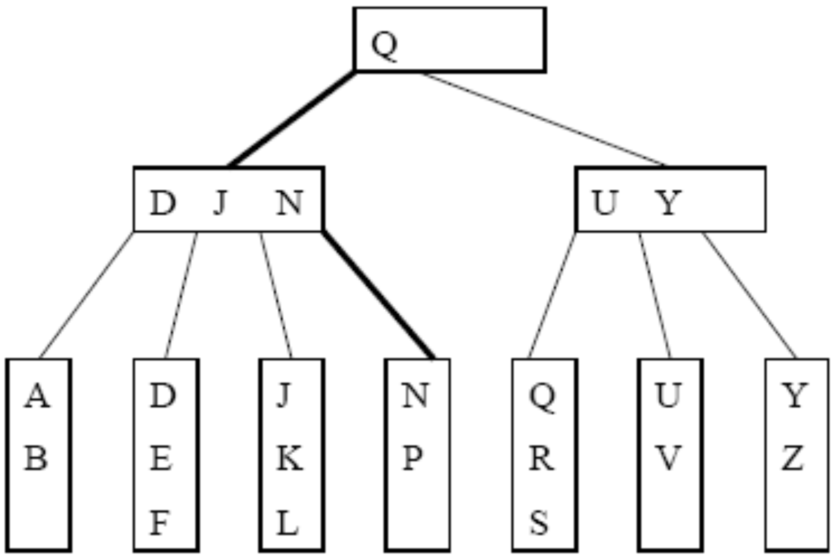
- Suppose that we want to insert a key K and its associated record.
- Search for the key K using the search procedure
- This will bring us to a leaf x
- Insert K into x
  - Splitting (instead of rotations in AVL trees) of nodes is used to maintain properties of B^+-trees [next slide]

Insertion into a Leaf

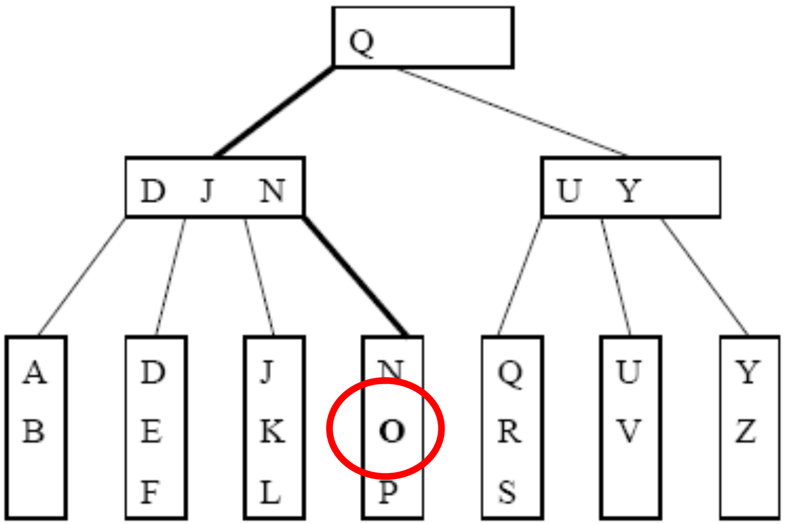


- If leaf x contains < L keys, then insert K into x (at the correct position in node x)
- If x is already full (i.e. containing L keys) then split x
  - Cut x off from its parent
  - Insert K into x, pretending x has space for K. Now x has L+1 keys.
  - After inserting K, split x into 2 new leaves x_L and x_R, with x_L containing the $\lfloor (L+1)/2 \rfloor$ smallest keys, and x_R containing the remaining $\lceil (L+1)/2 \rceil$ keys. Let J be the minimum key in x_R
  - Make a copy of J to be the parent of x_L and x_R, and insert the copy together with its child pointers into the old parent of x.

Inserting into a Non-full Leaf (L=3)

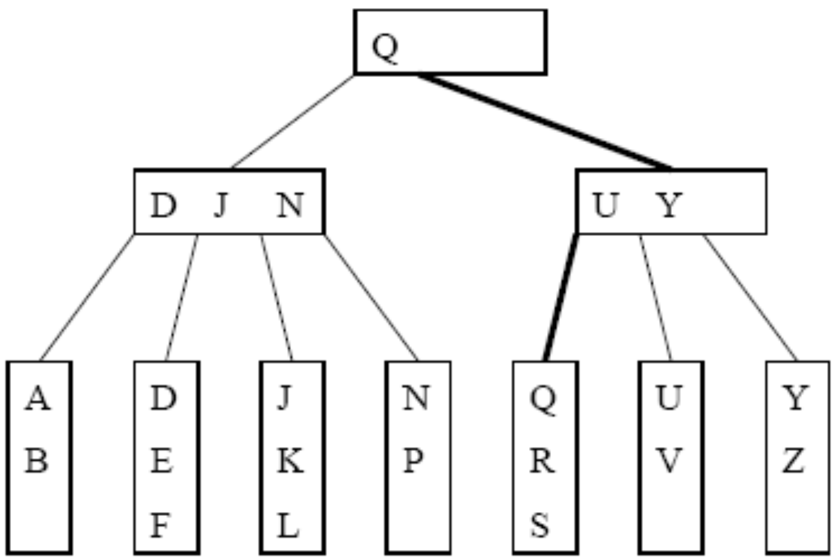


Search for O.

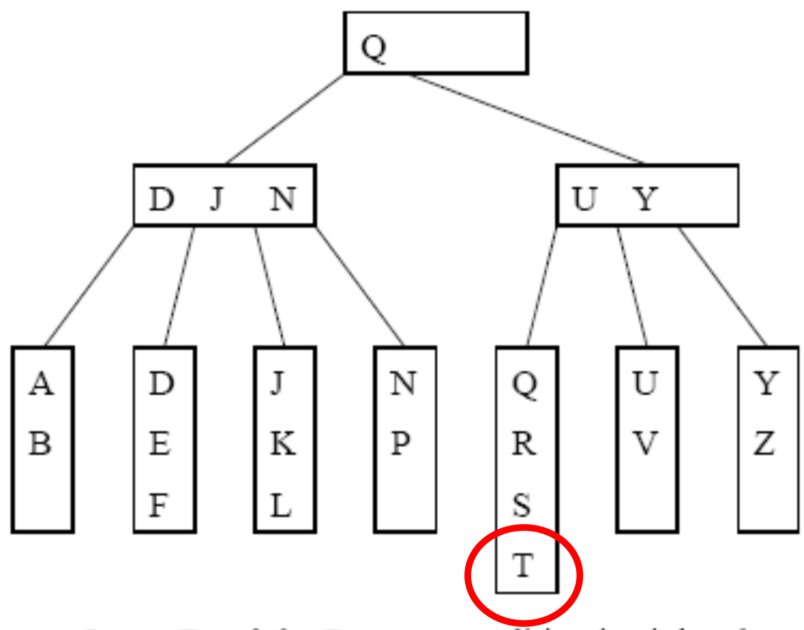


Insert O and maintain the order.

Splitting a Leaf: Inserting T

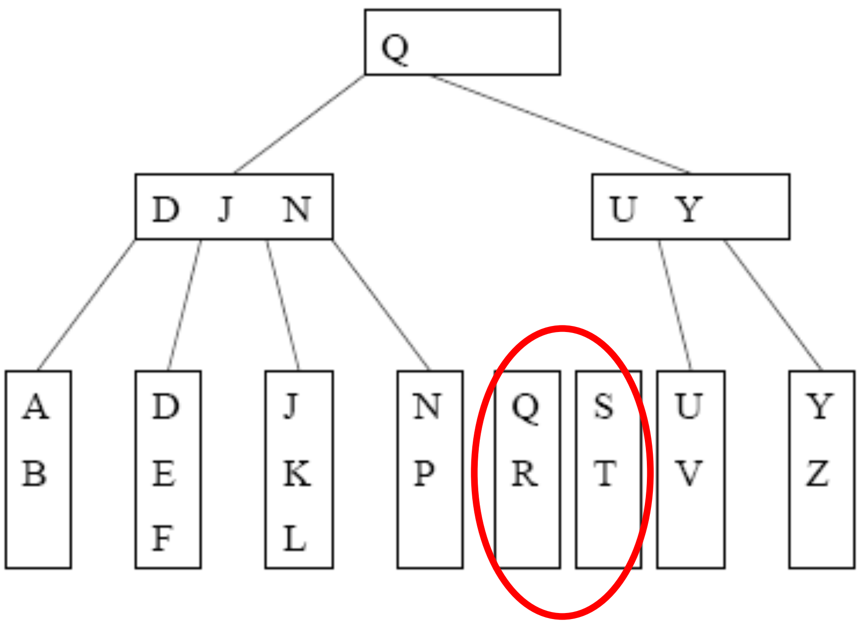


Search for T.

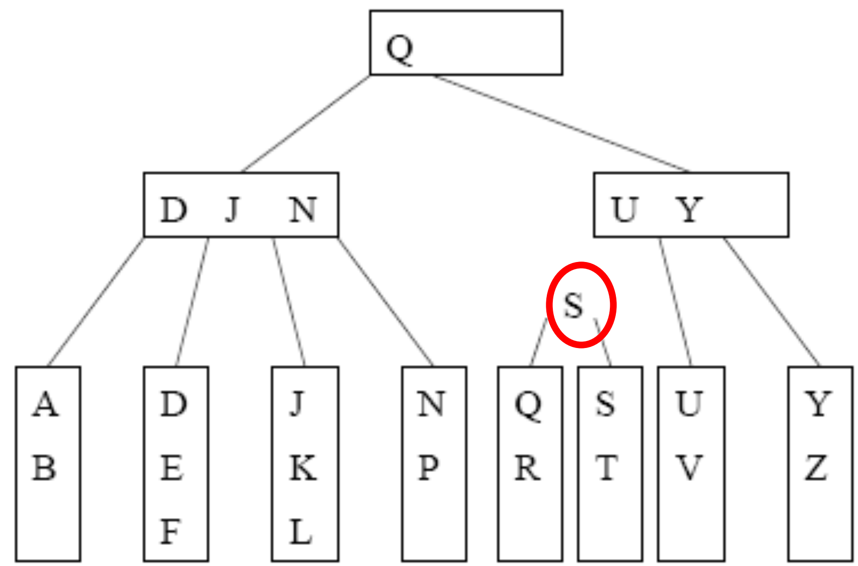


Insert T and the B+−tree condition is violated.

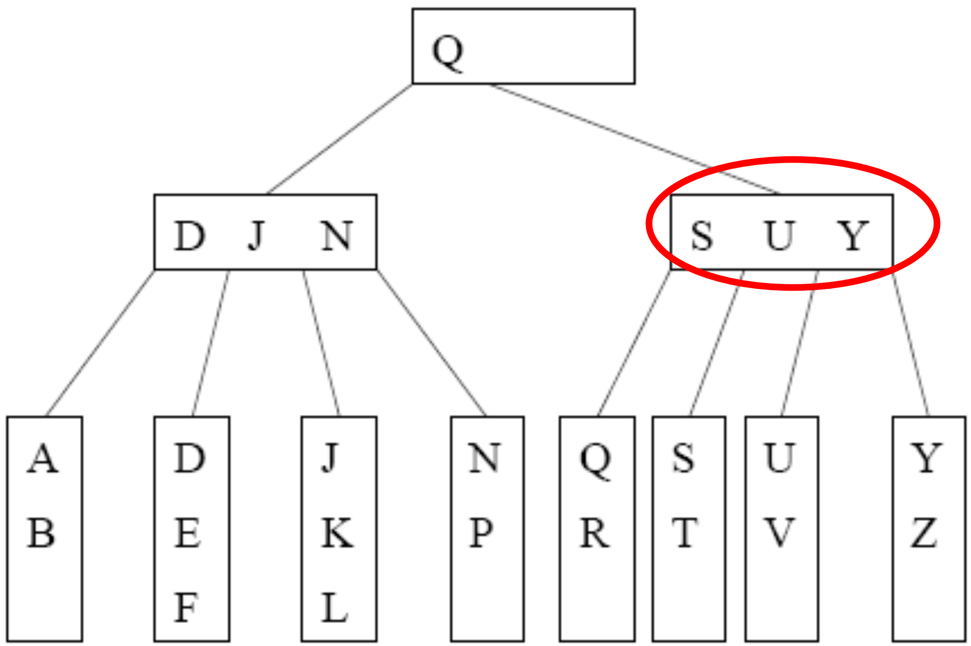
Splitting Example 1



Split the leaf and distribute the keys.



Make S the parent of the two new leaves.

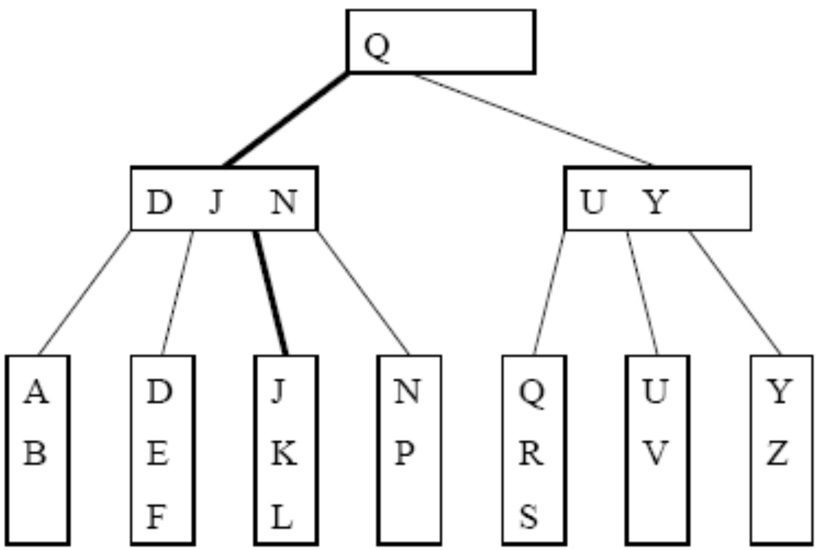


Insert S into the parent.

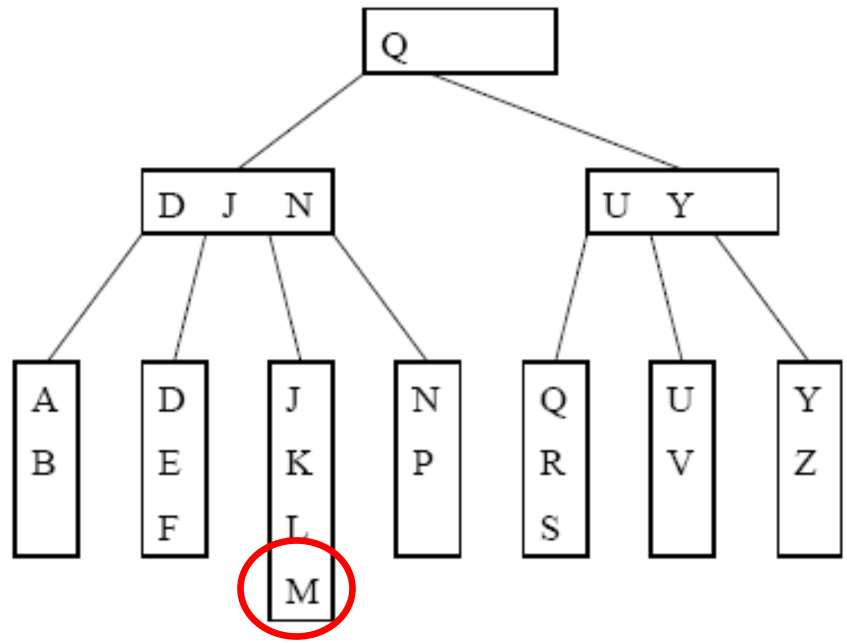
Maintain the order of keys and child pointers.

- Two disk accesses to write the two leaves, one disk access to update the parent
- For L=32, two leaves with 16 and 17 items are created. We can perform 15 more insertions without another split

Splitting Example 2

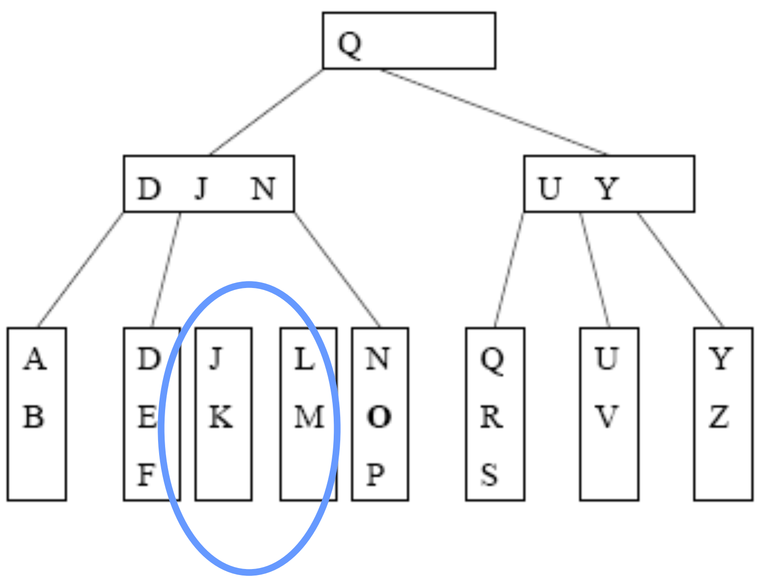


Search for M.

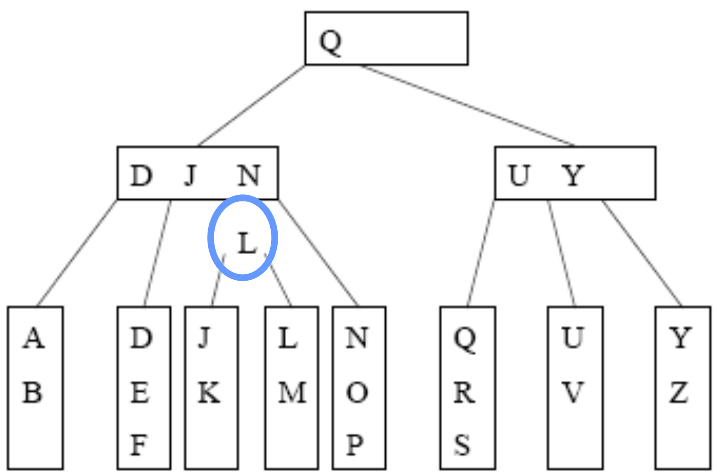


Insert M and the B+-tree condition is violated.

Cont’d



Split the leaf and distribute the keys.



Make L the parent of the two new leaves.

However, we cannot just insert L into the parent as it is already full.

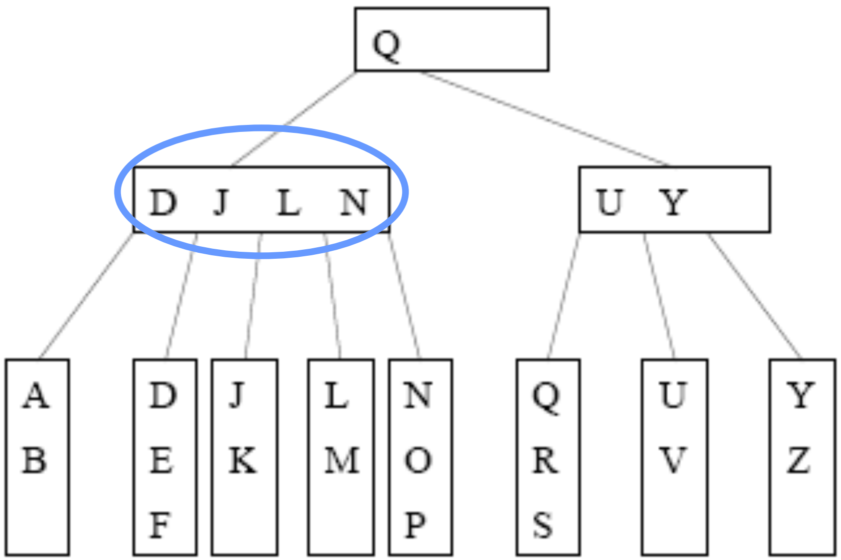
=> Need to split the internal node

Splitting an Internal Node

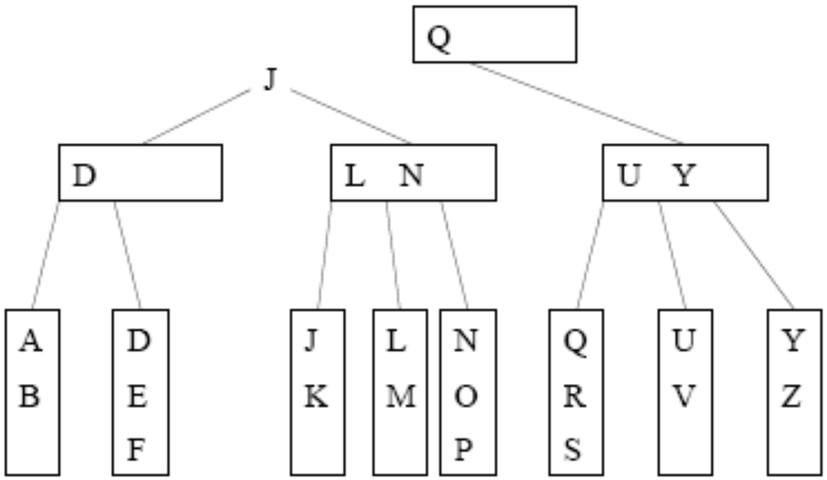
To insert a key K into a full internal node x:

- Cut x off from its parent
- Insert K and its left and right child pointers into x, pretending there is space. Now x has M keys (and M+1 pointers).
- Split x into 2 new internal nodes x_L and x_R, with x_L containing the $(\lceil M/2 \rceil - 1)$ smallest keys, and x_R containing the $\lfloor M/2 \rfloor$ largest keys. Note that the $(\lceil M/2 \rceil)^{th}$ key J is not placed in x_L or x_R
- Make J the parent of x_L and x_R, and insert J together with its child pointers into the old parent of x.

Example: Splitting Internal Node (M=4)



Insert L and its child pointers into the parent.

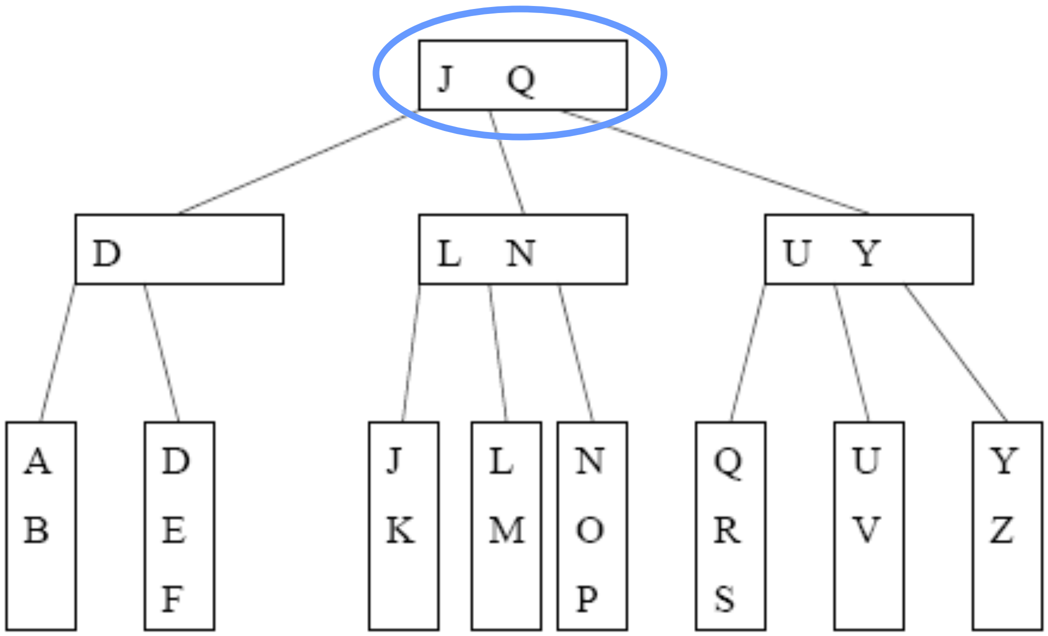


Split the parent.

The key J becomes the parent of the two new internal nodes.

Insert J into the next parent.

Cont'd



Split the parent.

The key J becomes the parent of the two new internal nodes.

Insert J into the next parent.

Termination

- Splitting will continue as long as we encounter full internal nodes
- If the split internal node x does not have a parent (i.e. x is a root), then create a new root containing the key J and its two children