B Tree

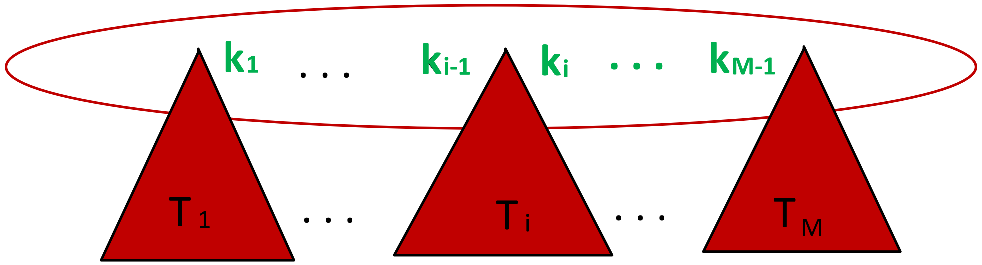
Joy Mukherjee

Assistant Professor
Computer Science & Engineering
IIT Bhubaneswar

Definition

A **B-tree** T is a rooted tree having the following properties:

1. Each internal node contains n keys and n+1 children
2. Keys are stored in increasing order
3. The keys separate the ranges of keys stored in each subtree



Definition

4. All leaves have the same depth
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:
   a. Every internal node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is full if it contains exactly $2t - 1$ keys.

Degree of B-tree: Minimum number of children of a non-root internal node

Order of B-tree: Maximum number of children of a non-root internal node

Maximum number of keys in a B-Tree

- Degree of B-tree = t
- Root node is at level-0
- In level-0, maximum number of keys = $(2t – 1)$
- In level-1, maximum number of keys = $2t\ (2t – 1)$
- In level-2, maximum number of keys = $(2t)^2\ (2t – 1)$
- …
- In level-h, maximum number of keys = $(2t)^h\ (2t – 1)$

Maximum number of keys in a B-Tree

- $n \le (2t - 1)\,[1 + 2t + (2t)^2 + \ldots + (2t)^h\,]$
- $n \le (2t - 1)\,[((2t)^{h+1} - 1)/\,(2t - 1)\,]$
- $n \le (2t)^{h+1} - 1$
- $h \ge \log_{2t}(n + 1) - 1$
- Maximum number of keys in a B-Tree of degree t is **$(2t)^{h+1} - 1$**
- Minimum height of a B-tree of degree t is **$\log_{2t}(n + 1) - 1$**

Minimum number of keys in a B-Tree

- Degree of B-tree = t
- Root node is at level-0
- In level-0, minimum number of keys = 1
- In level-1, minimum number of keys = $2(t - 1)$
- In level-2, minimum number of keys = $2t\,(t - 1)$
- …
- In level-h, minimum number of keys = $2t^{h-1}\,(t - 1)$

Minimum number of keys in a B-Tree

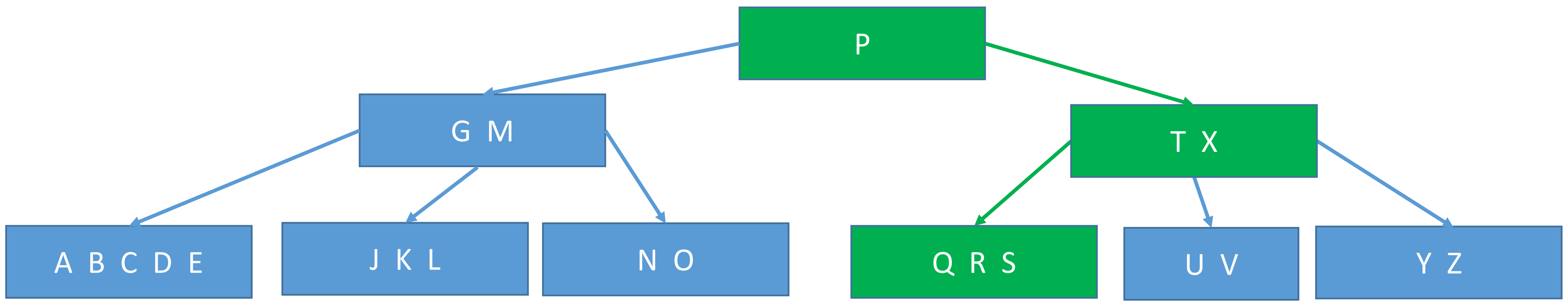
- $n \geq 1 + 2(t - 1)\ [1 + t + t^2 + \ldots + t^{h-1}\ ]$
- $n \geq 1 + 2(t - 1)\ [(t^h\ - 1)/\ (t - 1)\ ]$
- $n \geq 1 + 2(t^h - 1)$
- $n \geq 2t^h - 1$
- $h \leq \log_t((n + 1)/2)$
- Minimum number of keys in a B-Tree of degree t is **$2t^h - 1$**
- Maximum height of a B-tree of degree t is **$\log_t((n + 1)/2)$**

Search in a B Tree

Search in a B-Tree

```
B-Tree-Search(root, k)    // root is in main memory
    i ← 1
    while (i ≤ n[root]  &&  key_i [root] < k)
          i = i + 1
    if (i ≤ n[root]  && key_i [root] == k)
        return (root, i)
    if (root == NULL)              // leaf node
        return NULL
    else
        DISK-READ(c_i [root]) // Load the child node into main memory from disk
        return B-Tree-Search(c_i [root], k)
```

Searching S



Running Time: Search

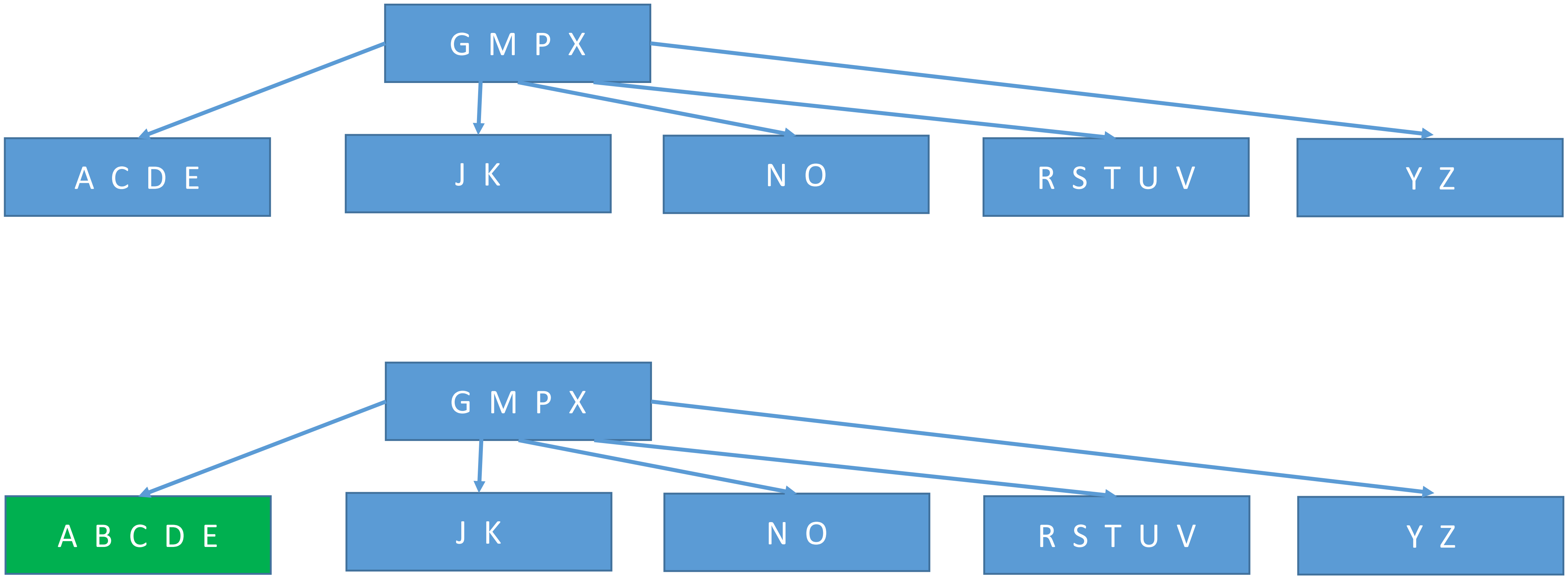
- Running time depends on
  - Number of disk accesses
  - CPU time
- The nodes encountered during the recursion form a path downward from the root of the B tree
- Number of disk pages read is $O(h) = O(\log_t n)$
- Since $n[x] < 2t$, searching in a node is $O(t)$
- Total CPU time is $O(th) = O(t \log_t n)$

Insertion in a B Tree

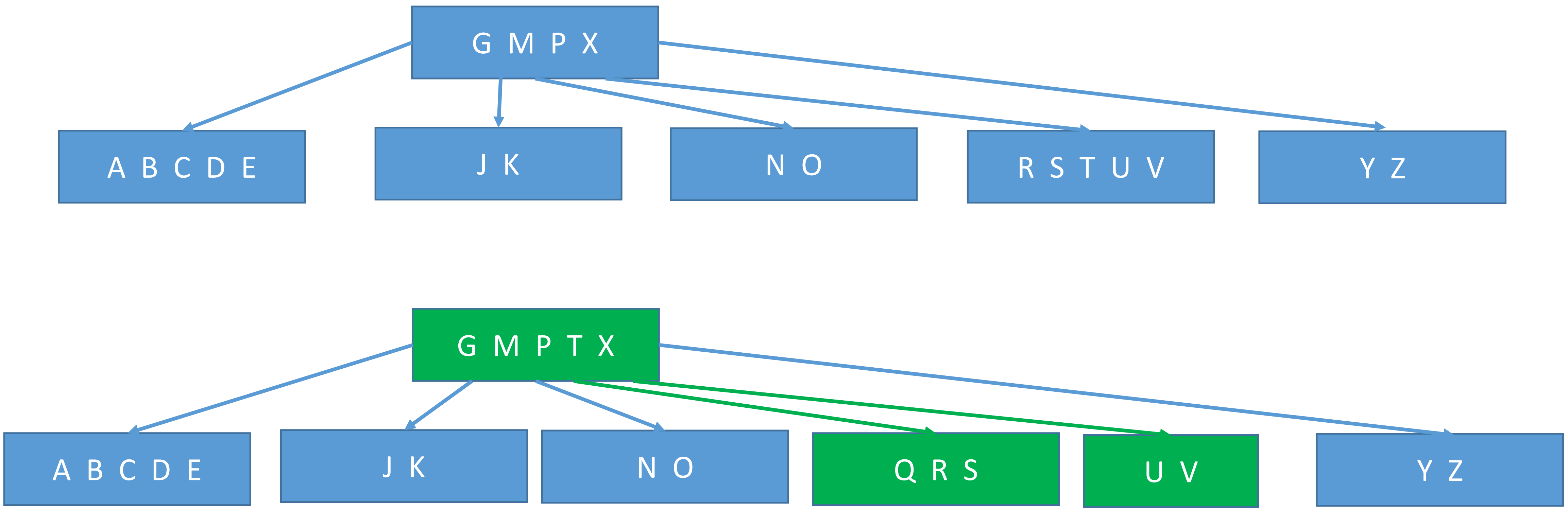
One Pass Insertion into a B-Tree

1. Initialize x as root.
2. While x is not leaf, do following
   a) Find the child of x that is going to be traversed next. Let the child be y.
   b) If y is not full, change x to point to y.
   c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than median key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3. The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

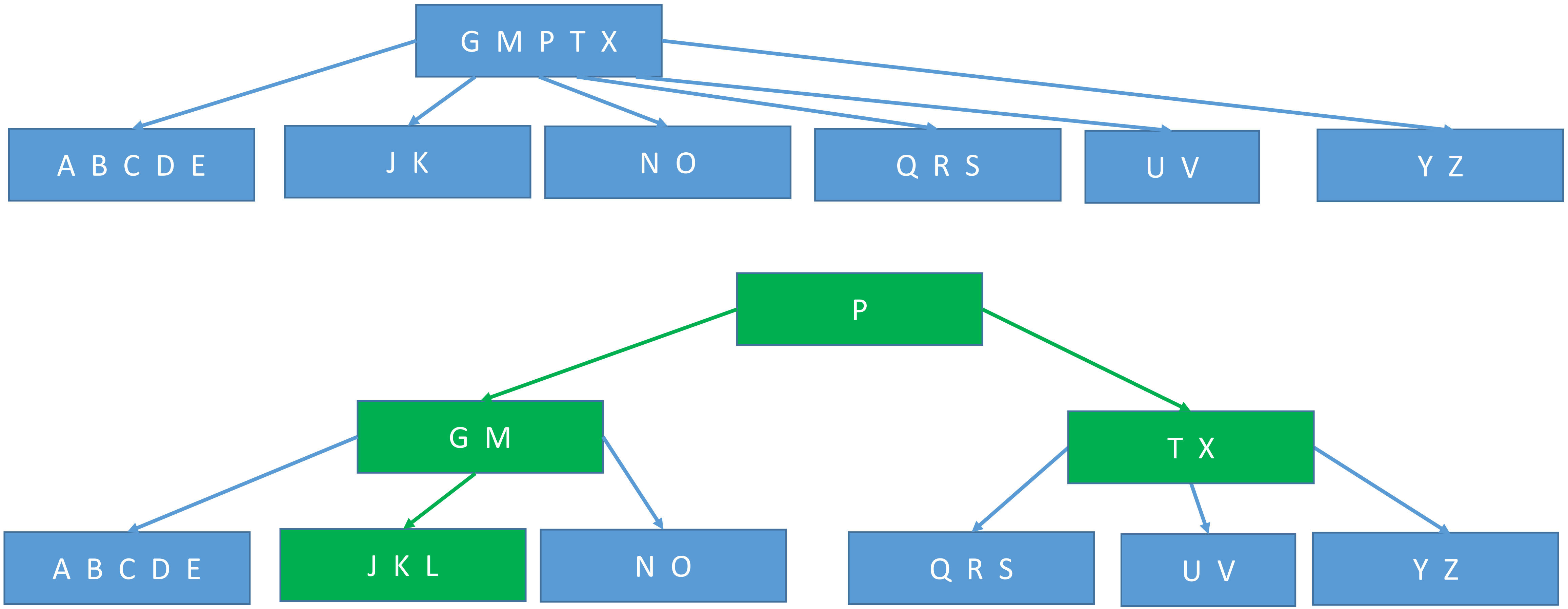
Inserting B



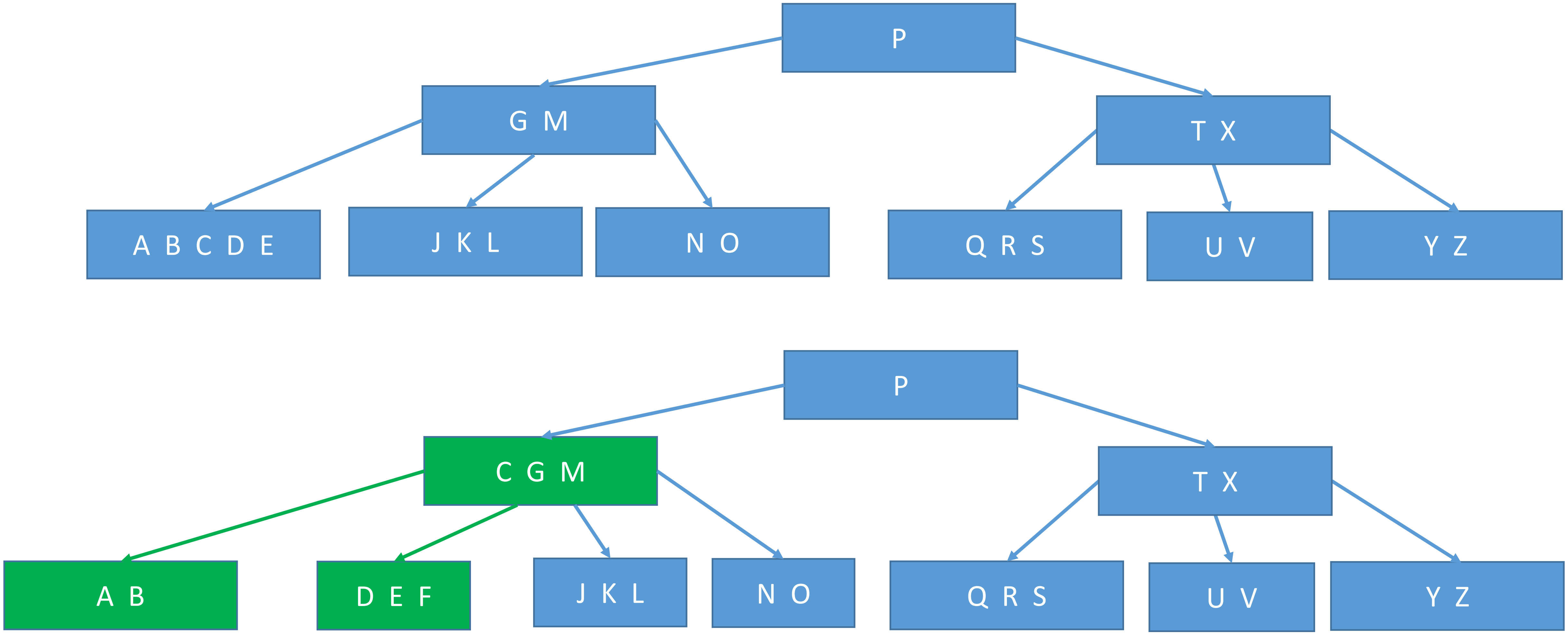
Inserting Q



Inserting L



Inserting F



Running Time: Insertion

- The nodes encountered during the recursion form a path downward from the root of the B tree
- Insertion involves only $O(h) = O(\log_t n)$ disk operations for a B-tree of height h, since only O(1) calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure.
- Since $n[x] < 2t$, searching in a node is $O(t)$
- Total CPU time is $O(th) = O(t \log_t n)$

Deletion in a B Tree

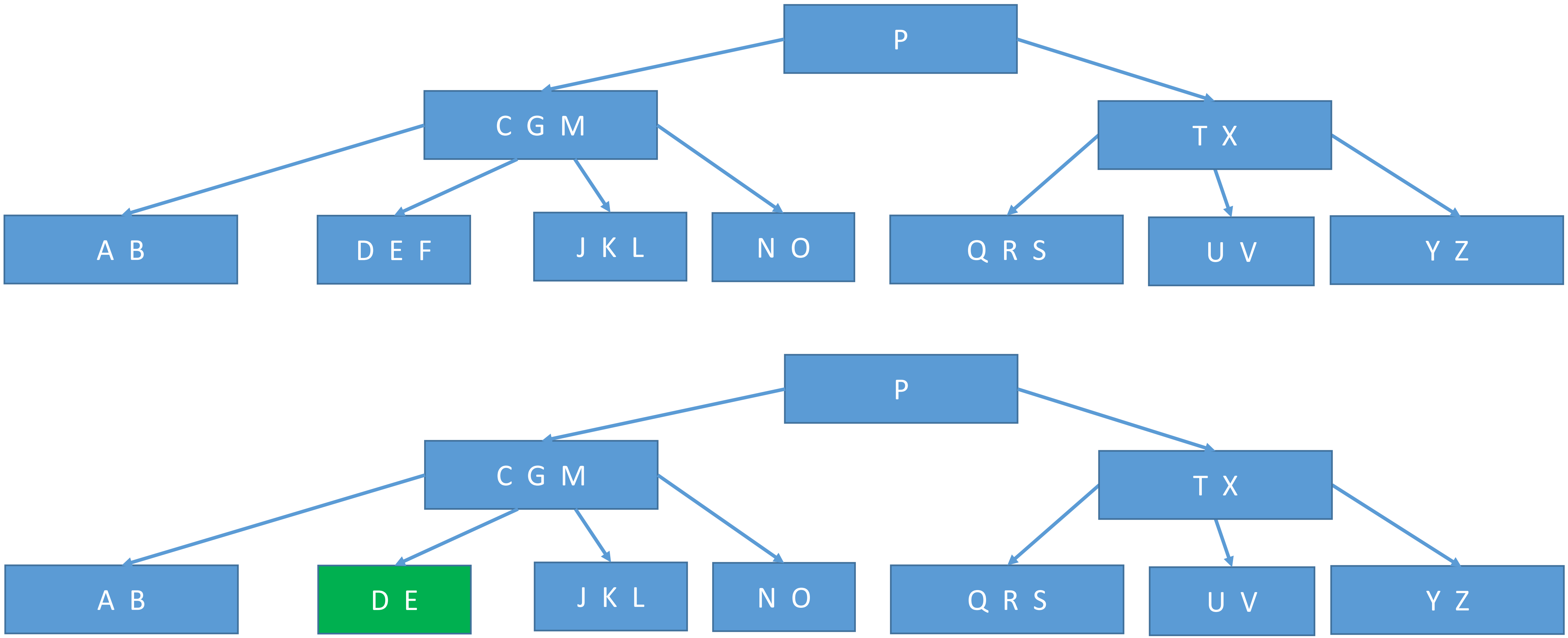
One Pass Deletion from a B-Tree

1. If the key k is in a leaf node x, delete k from x.
2. If the key k is in an internal node x, do the following.
   a. If the child y that precedes k in node x has at least t keys, then find the predecessor m of k in the sub-tree rooted at y. Replace k by m in x, and recursively delete m.
   b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor m of k in the subtree rooted at z. Replace k by m in x, and recursively delete m.
   c. If both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z, and recursively delete k from y.

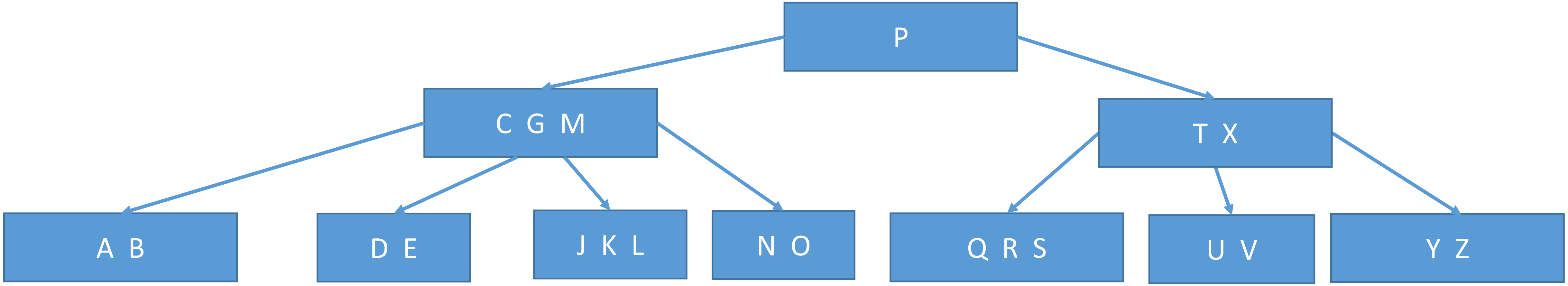
One Pass Deletion from a B-Tree

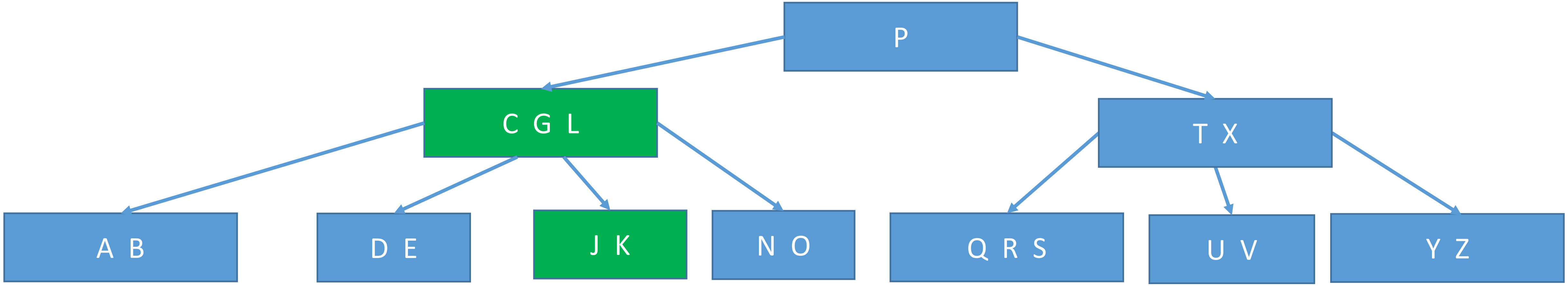
3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursion on the appropriate child of x.
    a. If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) ’s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).
    b. If x.c(i) and both of x.c(i)’s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Deleting F

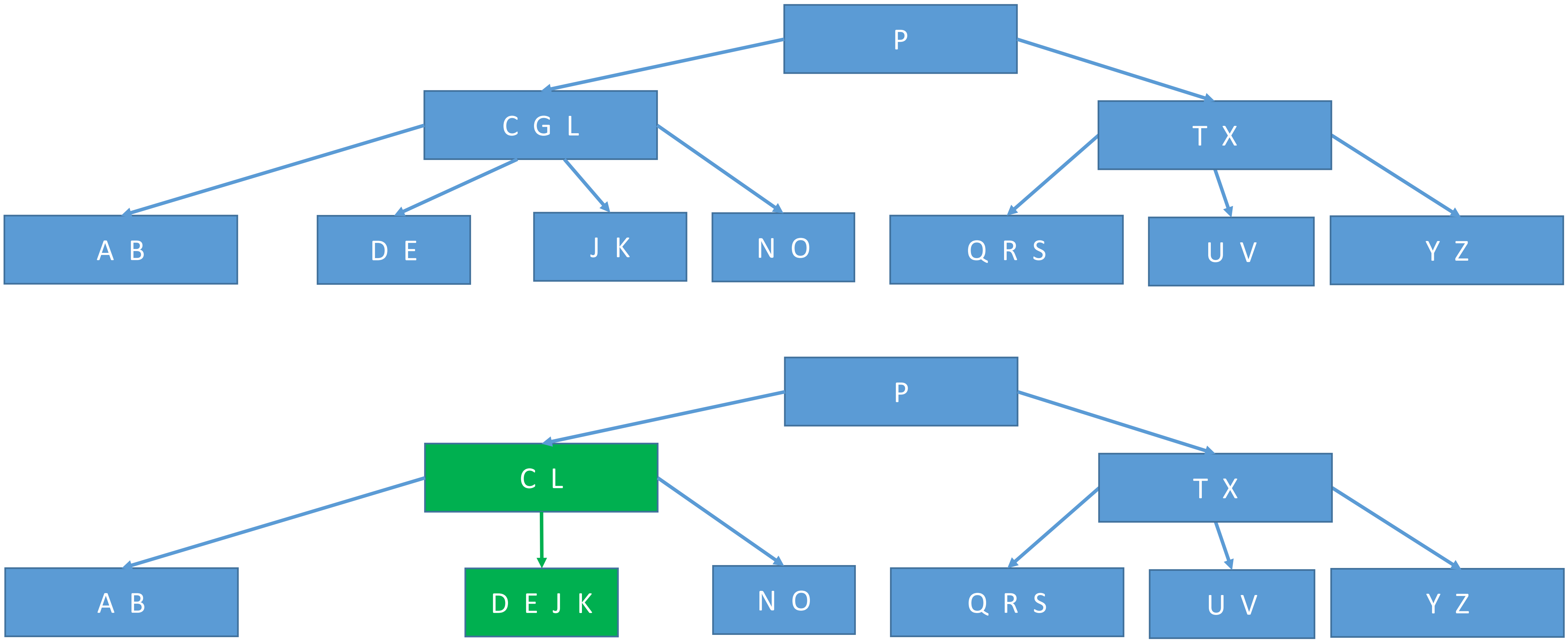


Deleting M

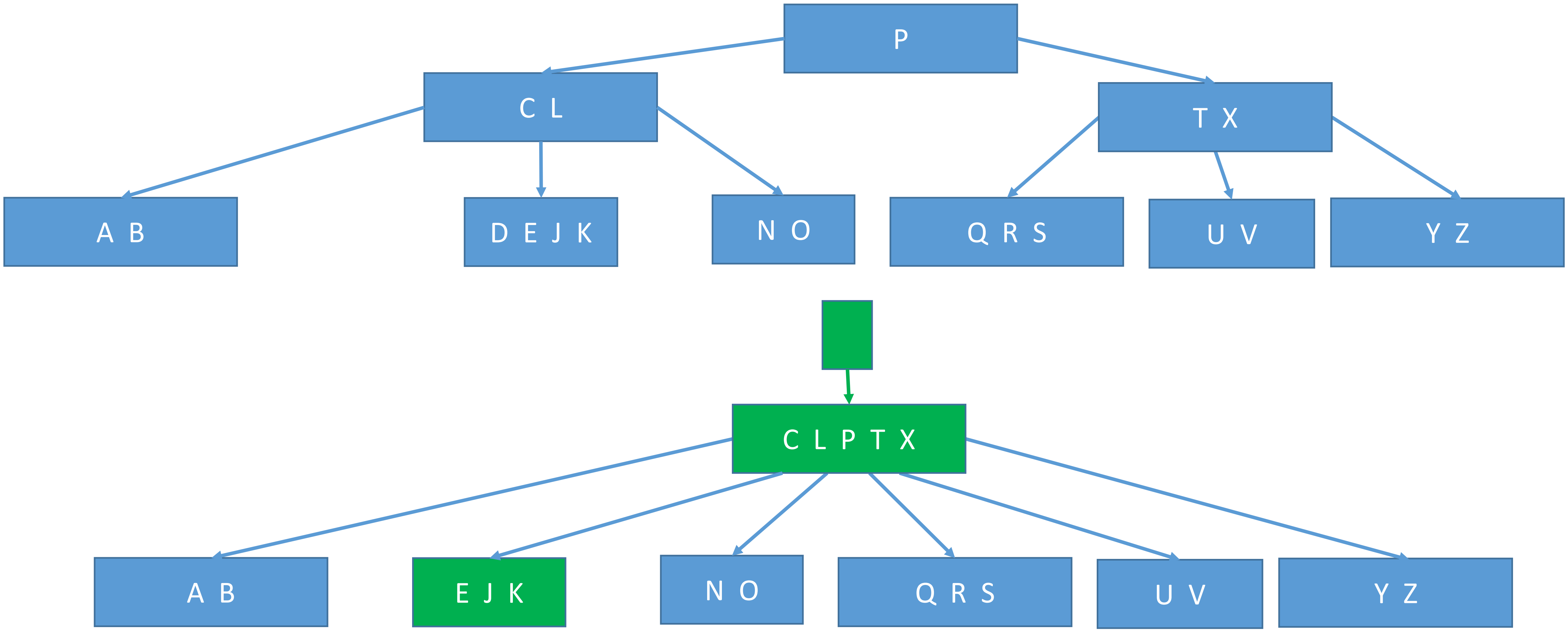




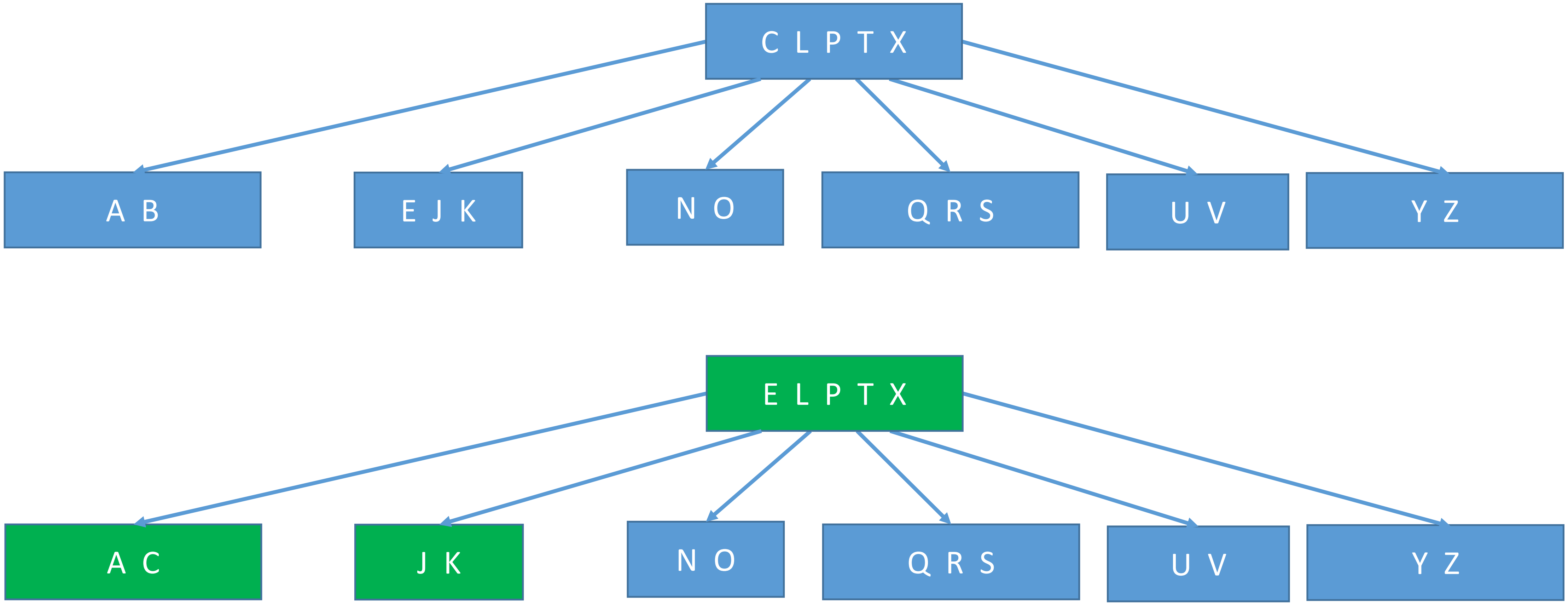
Deleting G



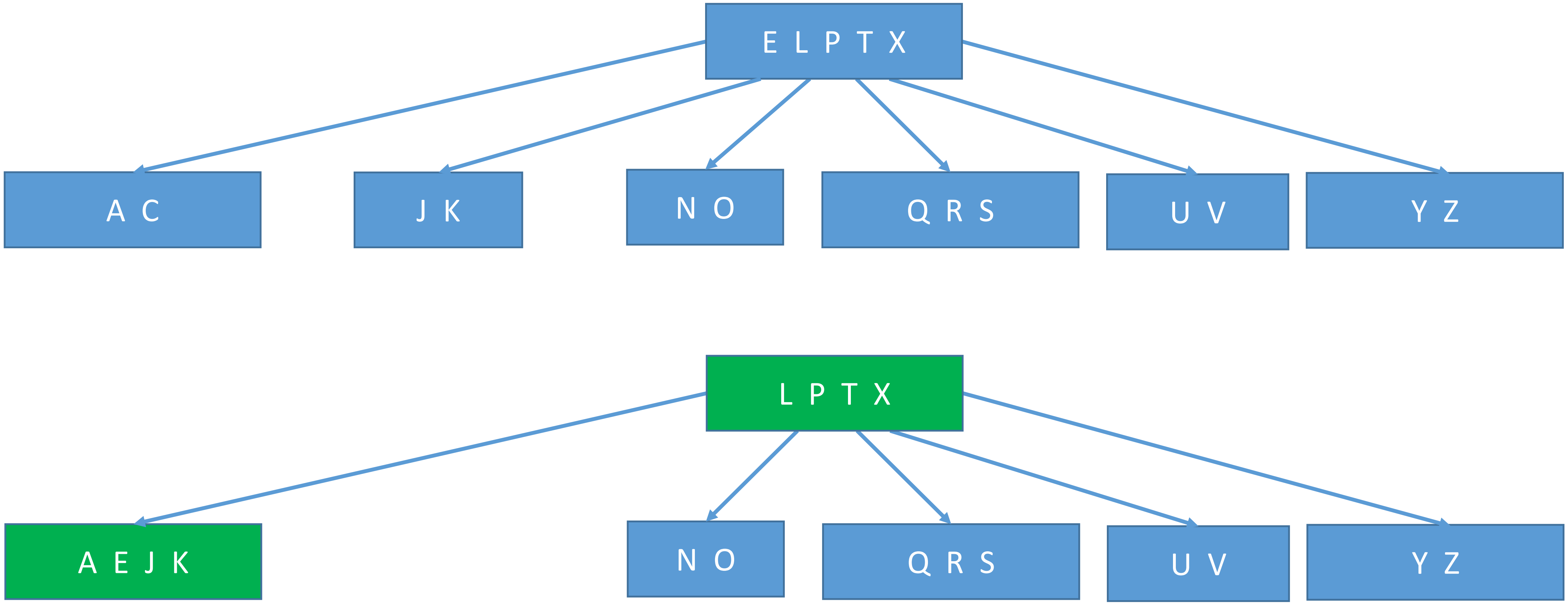
Deleting D



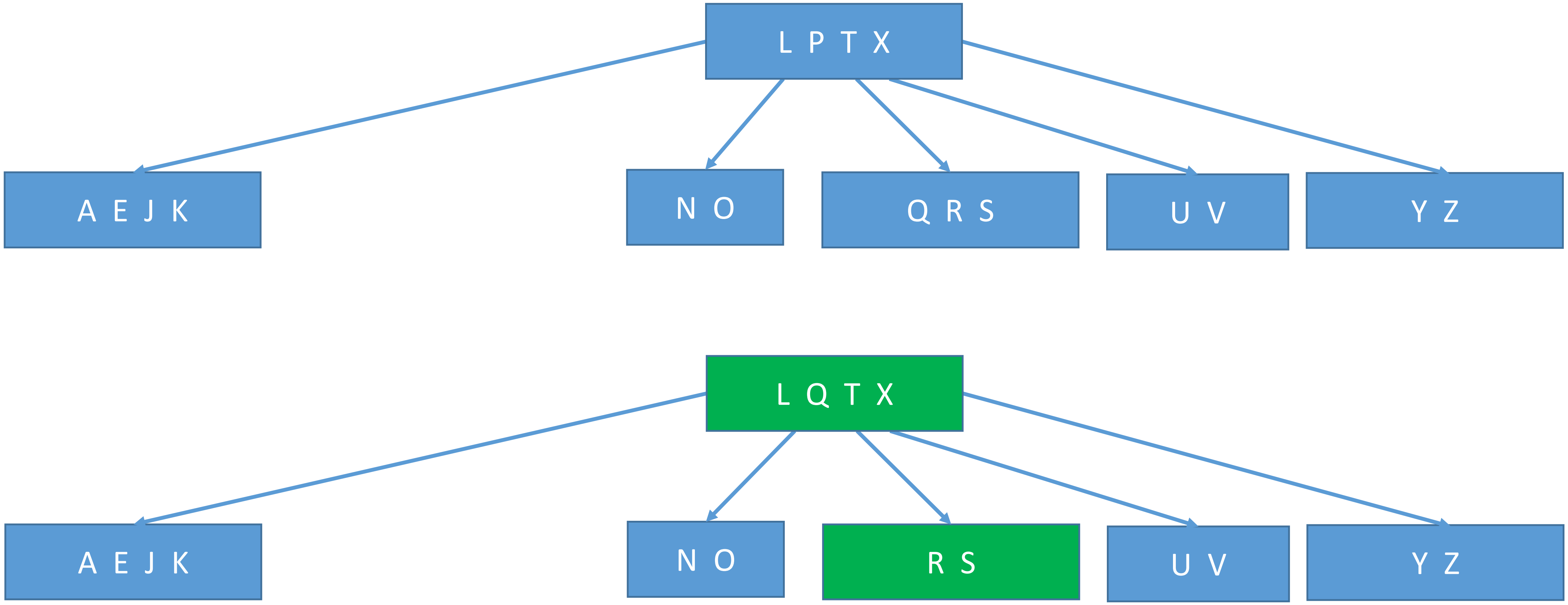
Deleting B



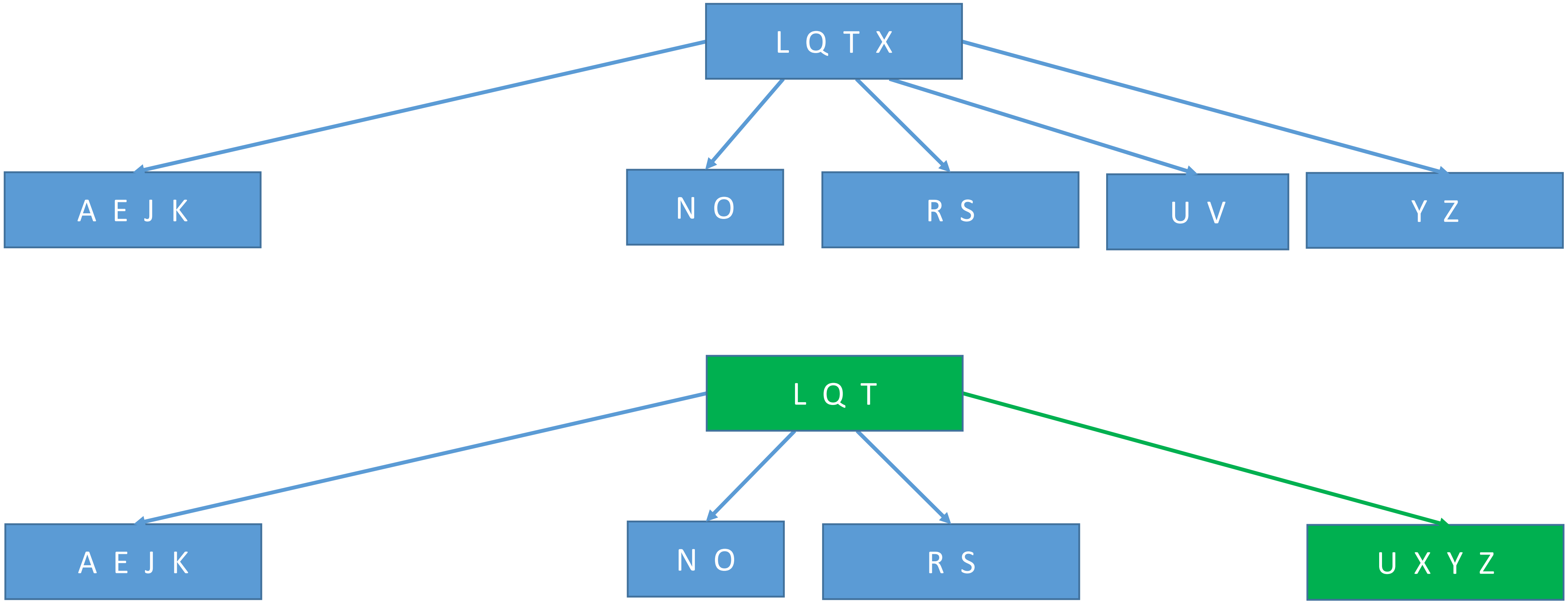
Deleting C



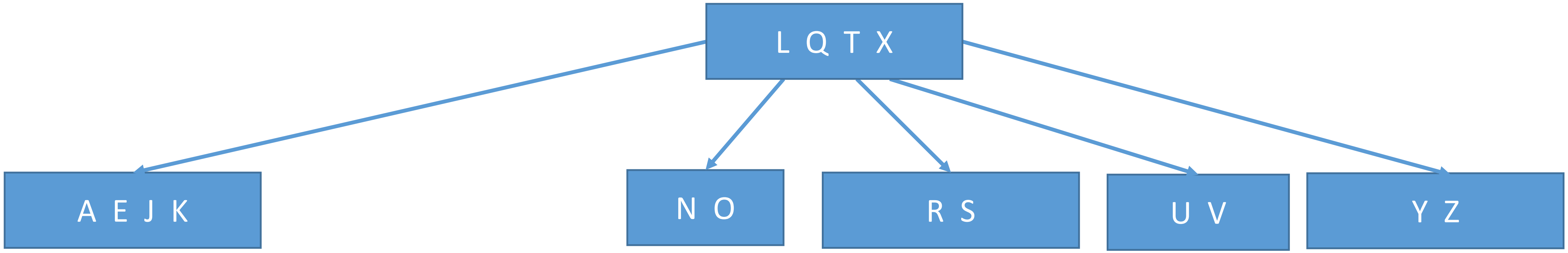
Deleting P

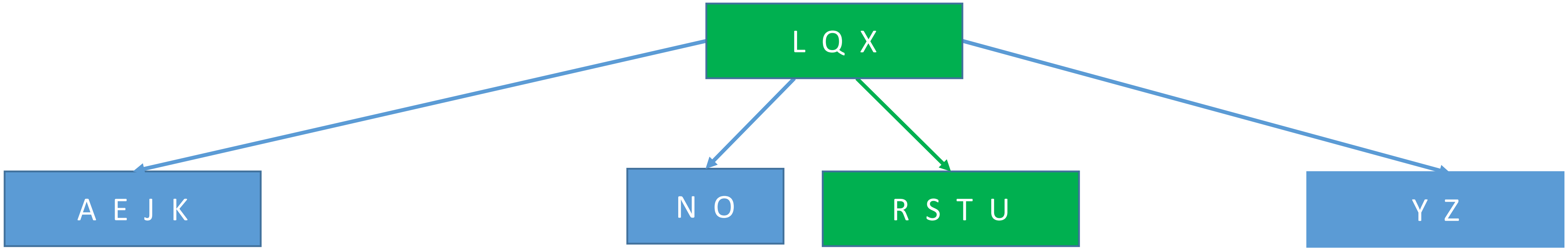


Deleting V



Deleting V





Running Time: Deletion

- The nodes encountered during the recursion form a path downward from the root of the B tree
- Deletion involves only $O(h) = O(\log_t n)$ disk operations for a B-tree of height h, since only O(1) calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure.
- Since $n[x] < 2t$, searching in a node is $O(t)$
- Total CPU time is $O(th) = O(t \log_t n)$