B-Trees (Chapter 18)

Bayer and McCreight (1972)

What does the “B” stand for?

Background

- A B-tree is a balanced search tree which is used to store the keys to large file and database systems
- Because:
  - only a part of a B-tree can be stored in primary (main) memory at a time and
  - secondary storage is much, much slower than primary memory

  it is advantageous to read/write as much information as possible from/to secondary storage and thereby minimize disk accesses

- Each node of a B-tree therefore is usually as large as a disk page and has a branching factor between 50 and 2000 depending on the size of the key relative to the size of a page

- Hence, a B-tree is wide and shallow

Five properties of a B-tree

Property 1 *

Each internal node has
a minimum degree of t (except for the root) and a
maximum degree of 2t.

* The notation B-tree (t=k) is used to denote a B-tree with a minimum degree of k

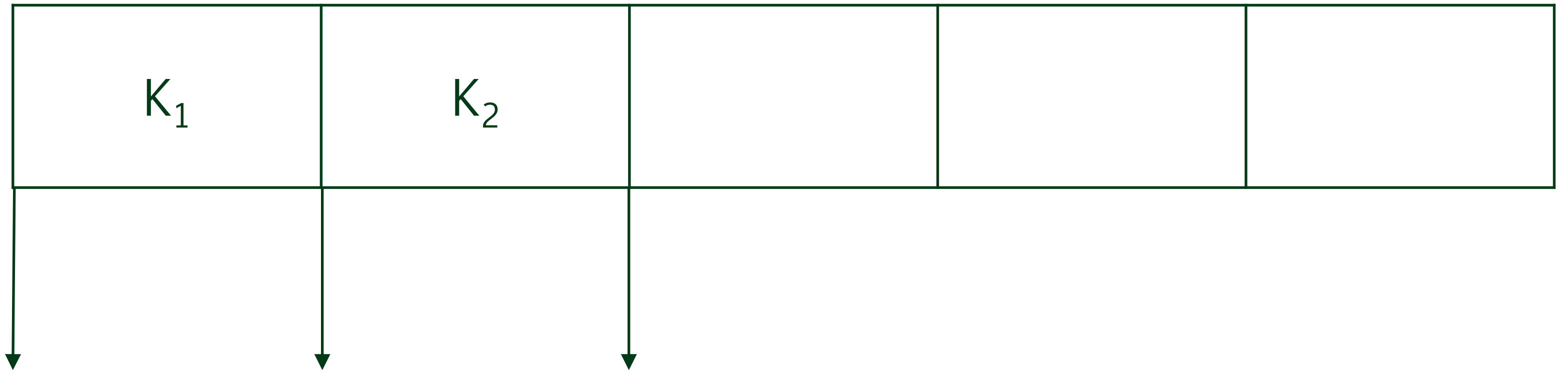
Five properties of a B-tree

Property 2

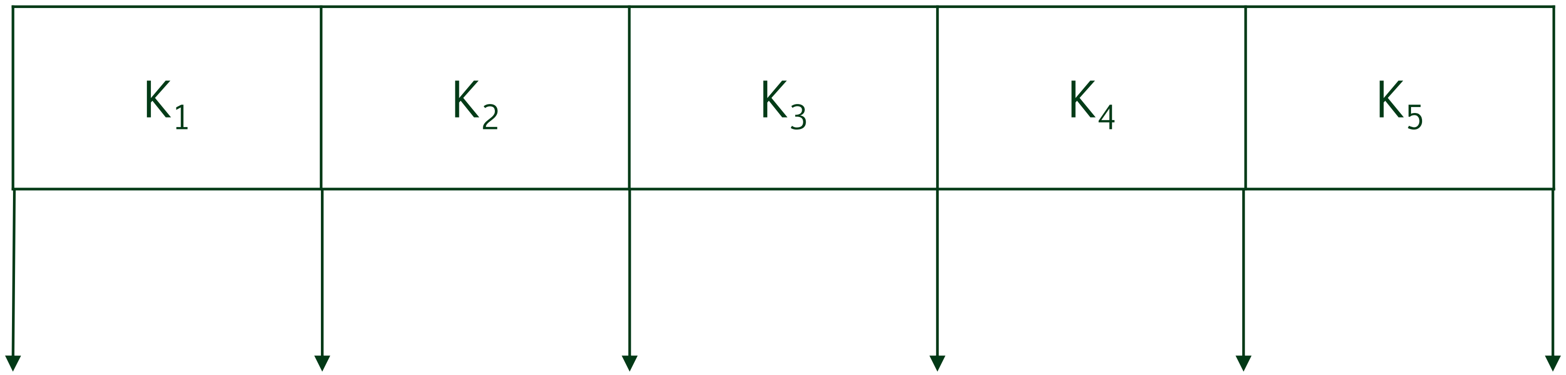
Each node has
a minimum of $t-1$ keys (except the root) and a
maximum of $2t-1$ keys.
If an internal node has degree d ($t \leq d \leq 2t$) then
it stores $d-1$ keys.

Properties 1 and 2 for B-tree (t=3)

Minimum degree



Maximum degree



Five properties of a B-tree

Property 3

The keys for each node are stored in ascending order.
Therefore,
$K_1 < K_2 < \ldots < K_{d-1}$
for a node with degree d.

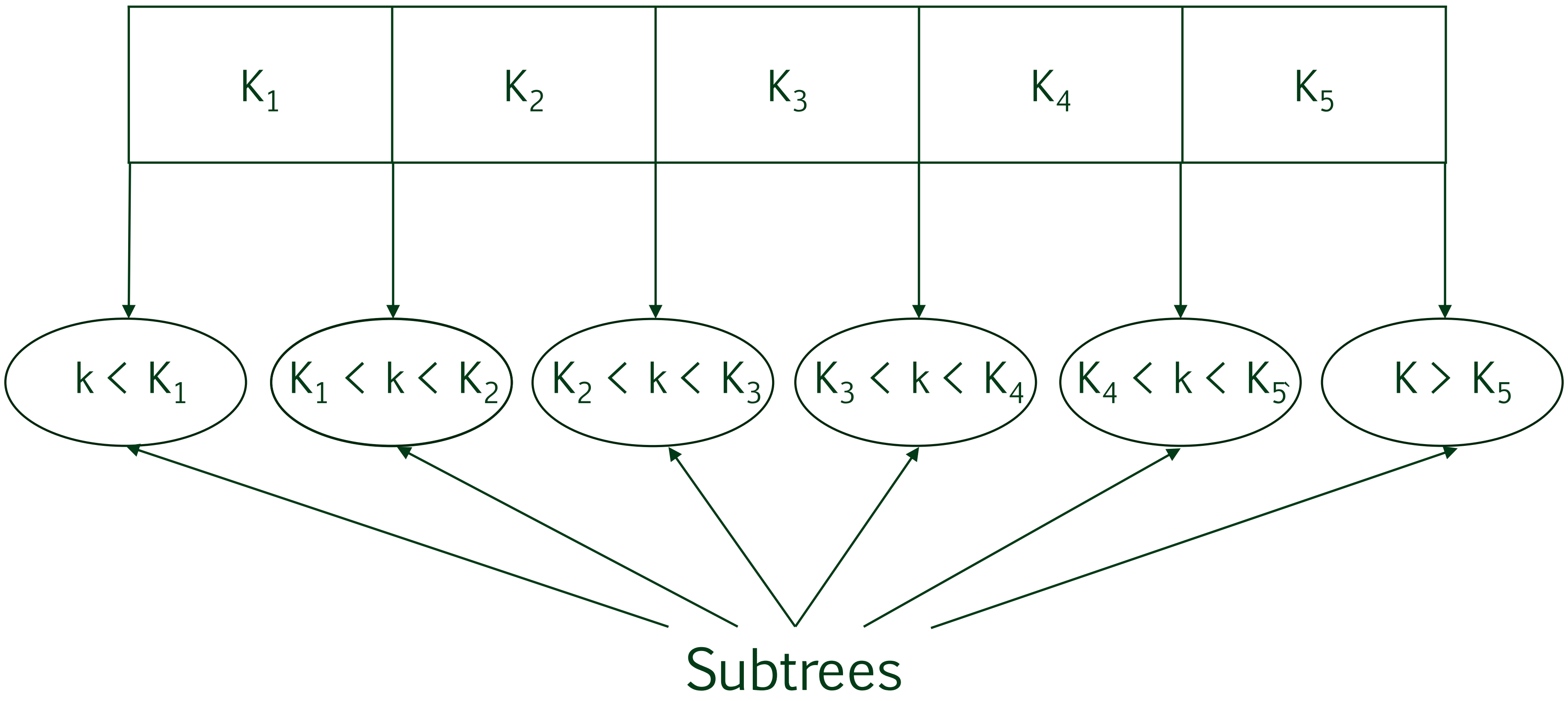
Five properties of a B-tree

Property 4

Key values in the i^{th} subtree of an internal node with degree d are

less than K_1	for $i=1$,
fall between K_{i-1} and K_i	for $2 \leq i \leq d-1$, and
greater than K_{d-1}	for $i=d$.

Properties 3 and 4 for B-Tree (t=3)



Five properties of a B-tree

Property 5

All leaf nodes have the same depth.

Maximum height h of a B-tree on n keys

Theorem (p489)

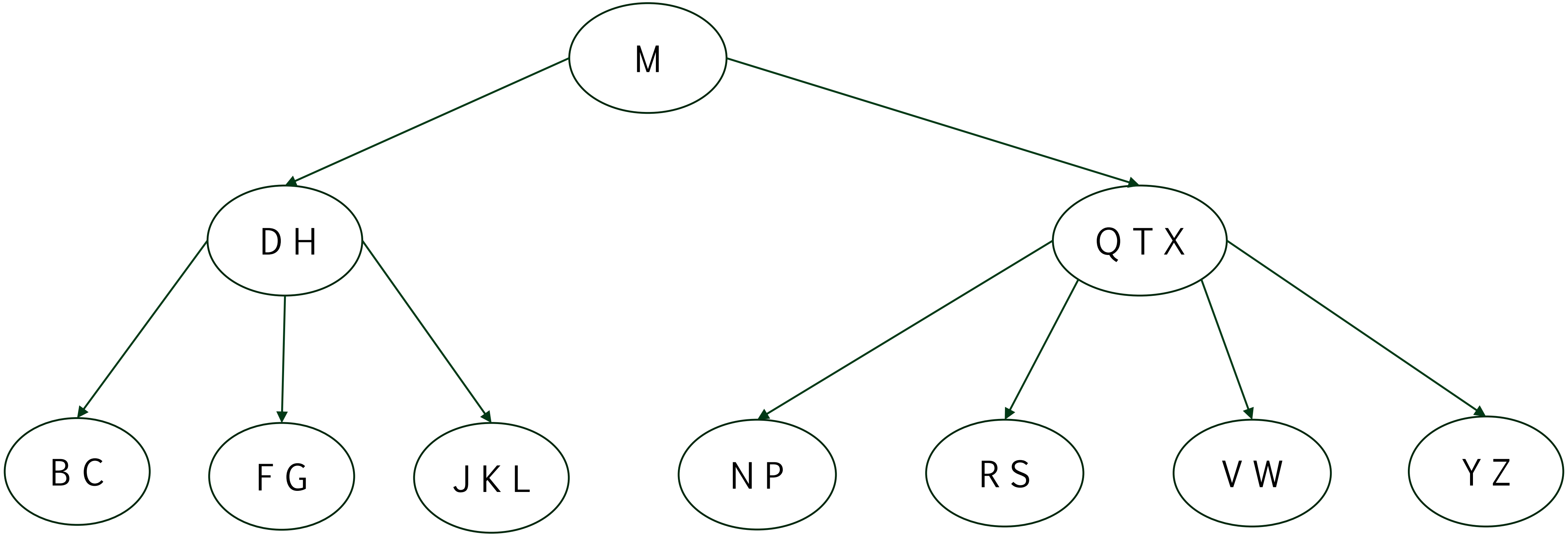
The maximum height on n keys for a B-tree with minimum degree t is

$h \leq \log_t (n+1)/2$ which is $O(\log n)$.

Exercises

- Look up and review the proof on the previous slide.
- Calculate the minimum height on n keys for a B-tree with minimum degree t.
- Show all the legal B-trees (t=2) that represent the keys {1, 2, 3, 4, 5}.
- For what values of t is the example tree on the next slide a legal B-tree?

Example tree *



* From CLRS (3rd Edition)

Data structure

```
class Node<T>
{
    private int n;              // number of keys
    private bool leaf;          // true if a leaf node; false otherwise
    private T[ ] key;           // array of keys
    private Node<T>[ ] c;       // array of child references

    public Node (int t) {
        n = 0;
        leaf = true;
        key = new T[2*t - 1];
        c = new Node<T>[2*t];
    }
}
```

Data structure

```
class BTree<T> where T : IComparable
{
    private Node<T> root;                // root node
    private int t;                       // minimum degree

    public BTree (int t) {
        this.t = t;
        root = new Node<T>(t);
    }
    ...
}
```

Primary methods

- public bool Search (T k)
  - returns true if the key k is found in the B-tree; false otherwise
- public void Insert (T k)
  - insert key k into the B-tree
  - duplicate keys are not permitted
- public void Delete (T k)
  - delete key k from the B-tree

Support method

- private void Split (Node<T> x, int i)
  - splits the i^{th} (full) child of x into 2 nodes

“One-pass” algorithms

- The primary methods Search, Insert, and Delete are implemented as “one-pass” algorithms where each algorithm proceeds down the B-tree from the root without having to back up to a node *

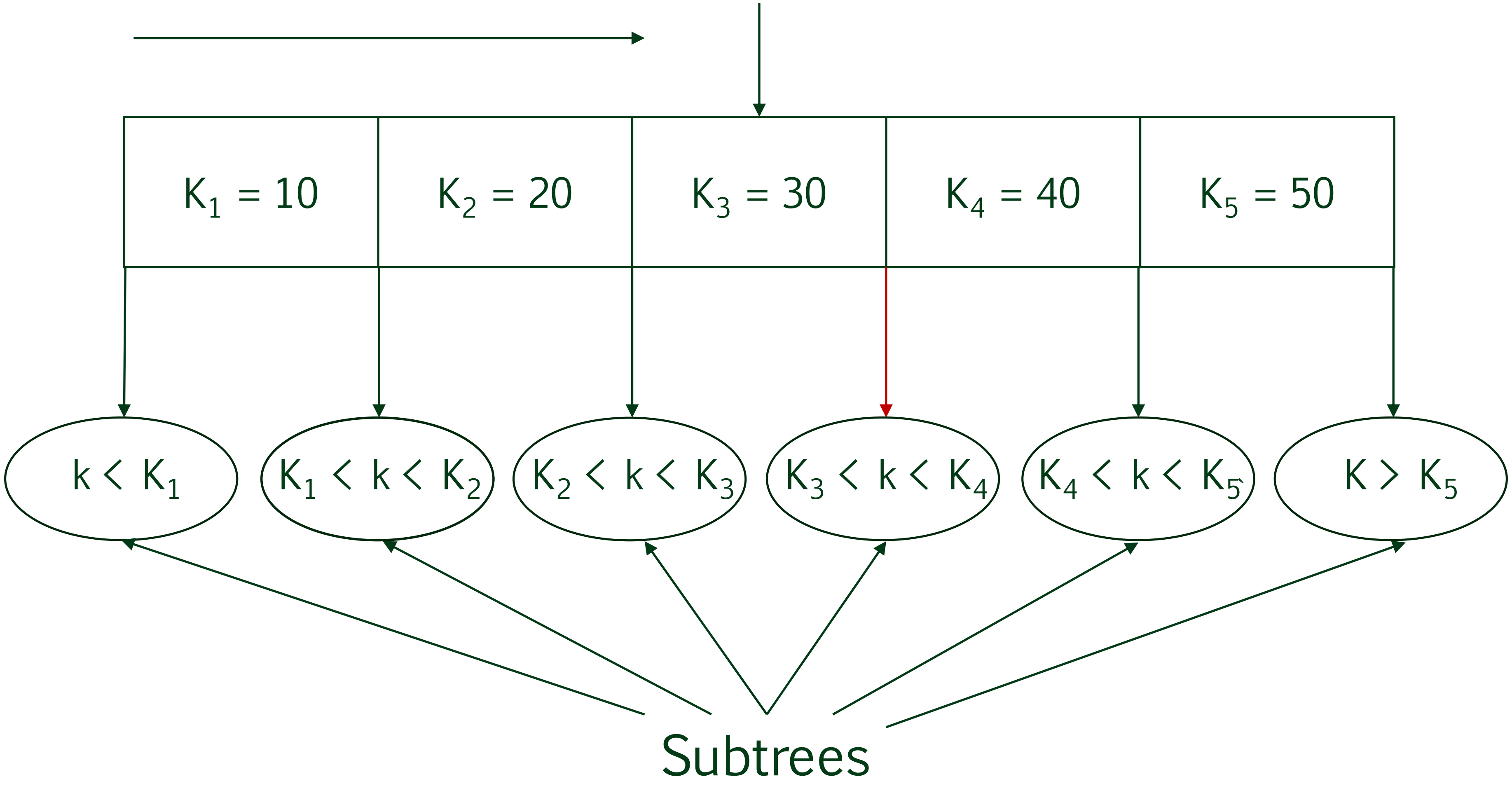
- This minimizes the number of read and write operations from secondary memory (disk)

* with one exception for deletion

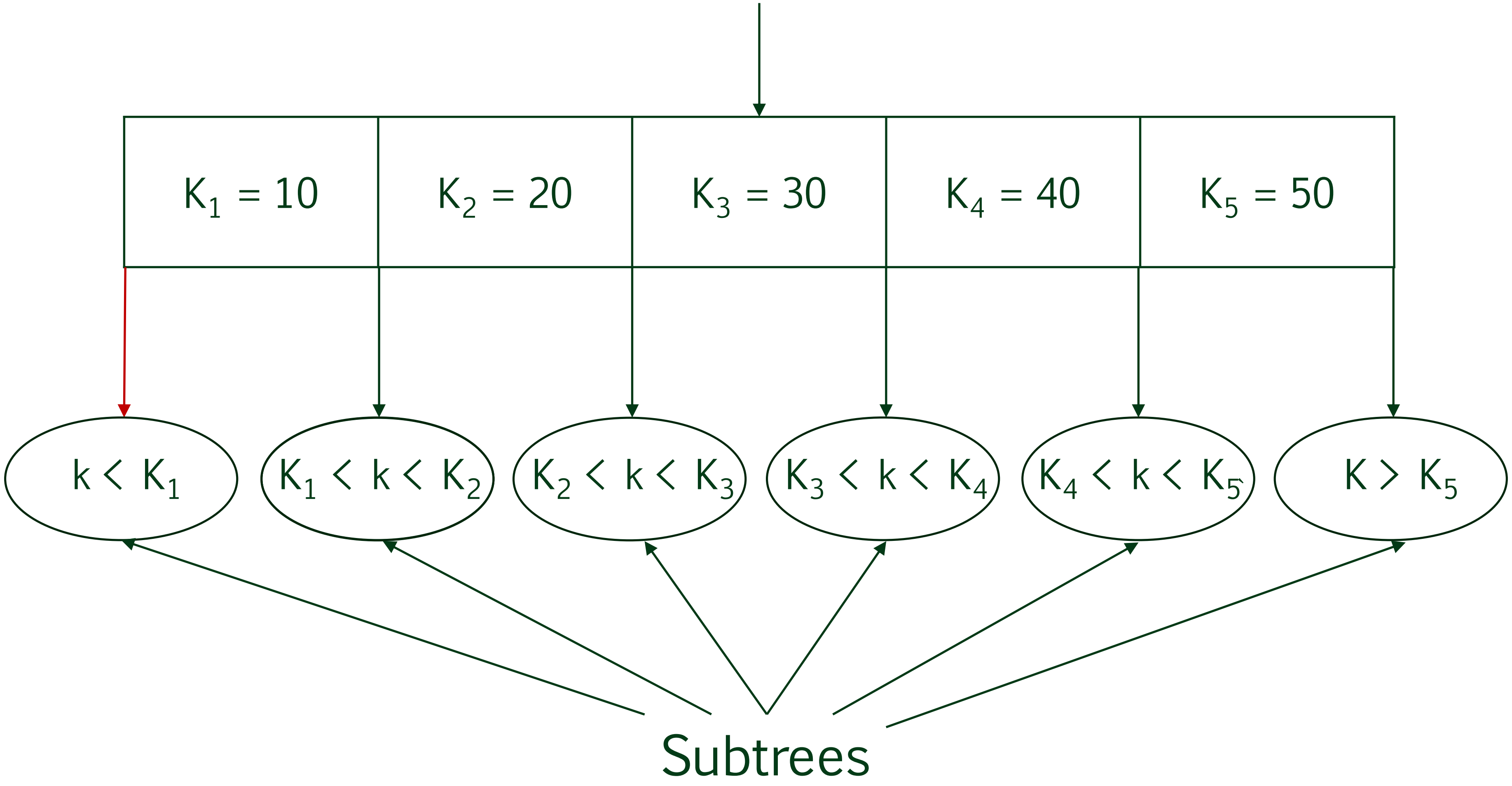
Search

- Basic strategy
  - Set p to the root node
  - If the given key k is found at p then return true
  - If the key k is not found and p is a leaf node then return false else move to the subtree which would otherwise contain k

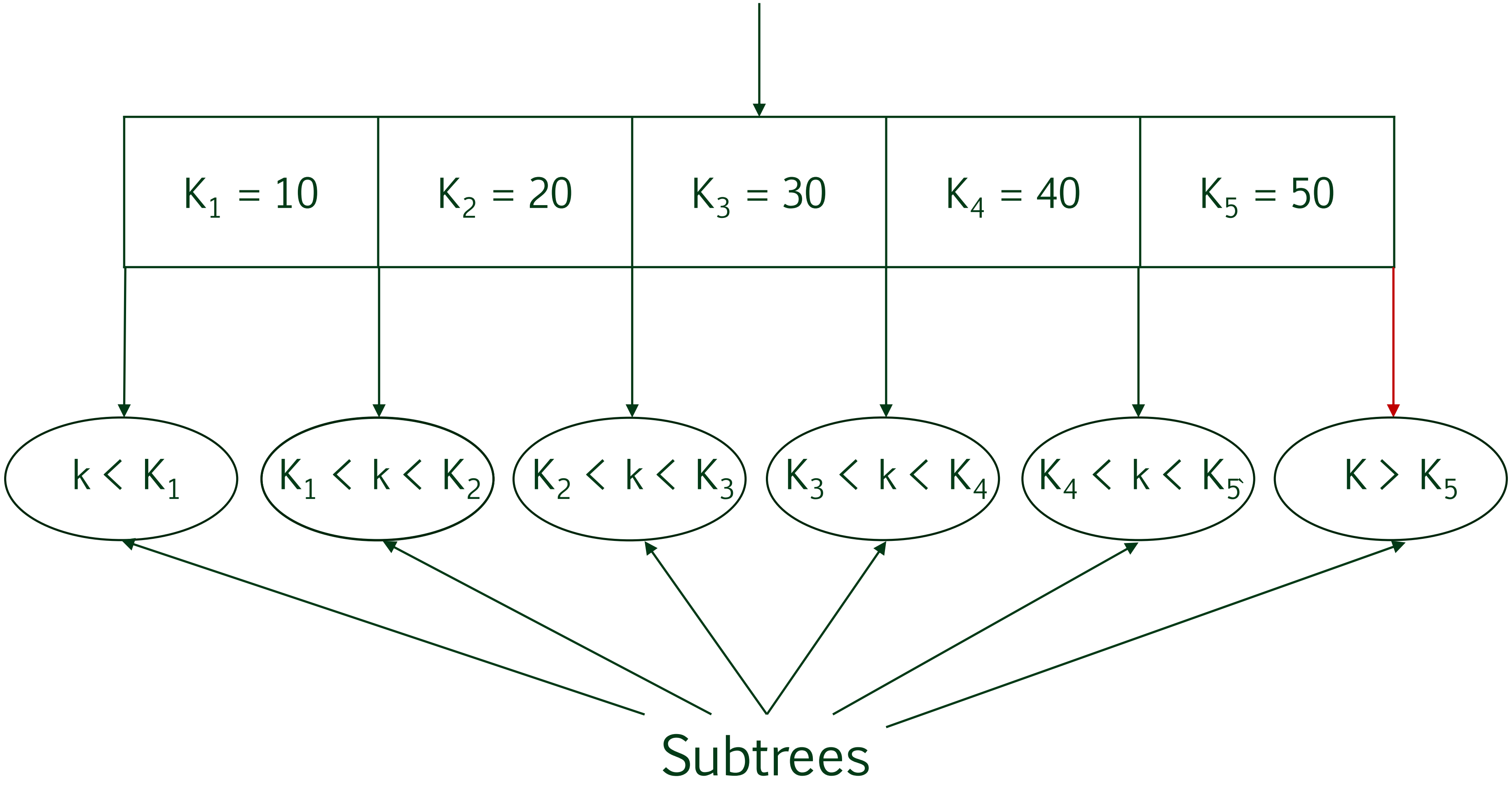
Search for key k = 35



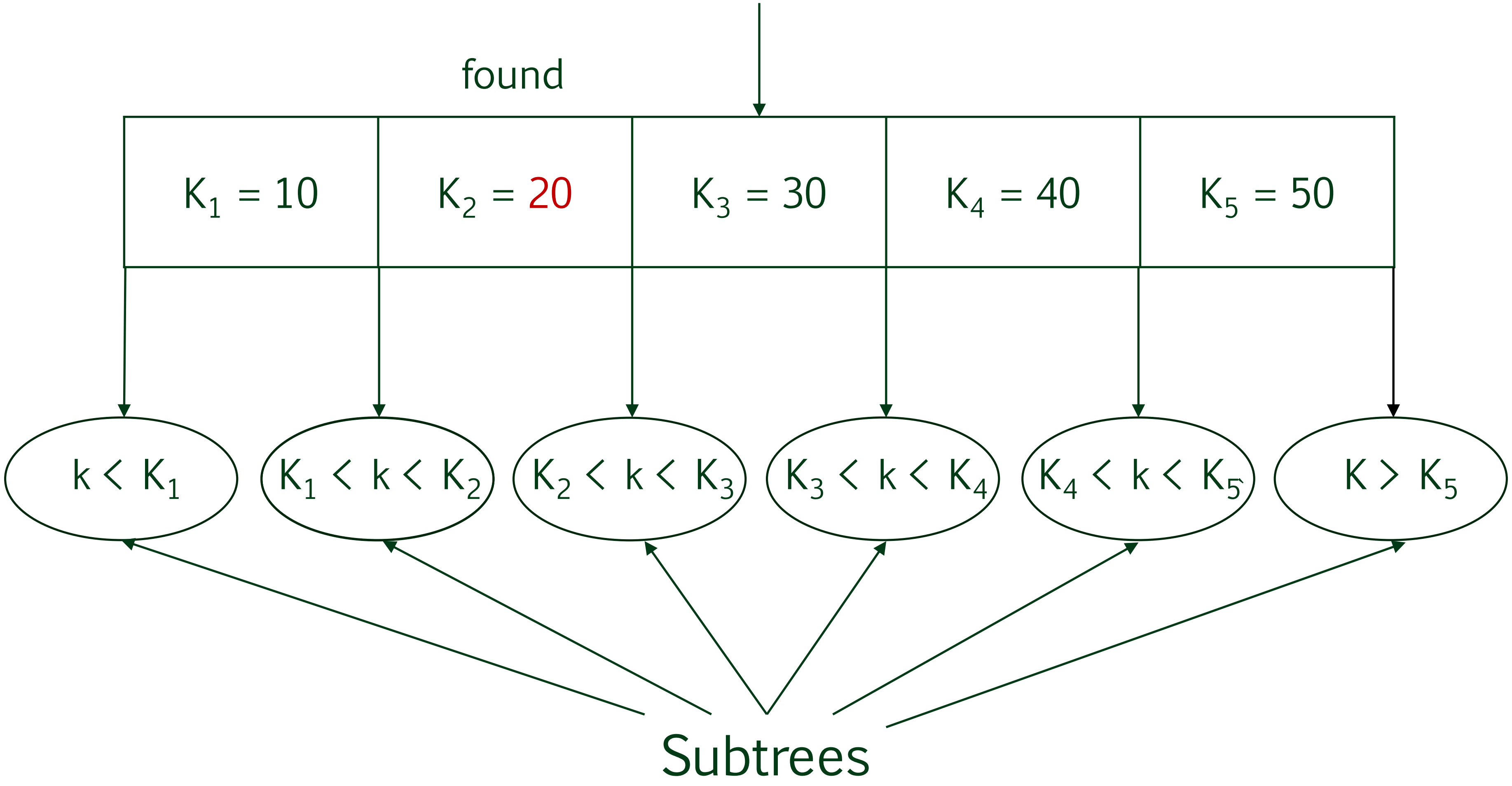
Search for key k = 7



Search for key k = 80



Search for key k = 20



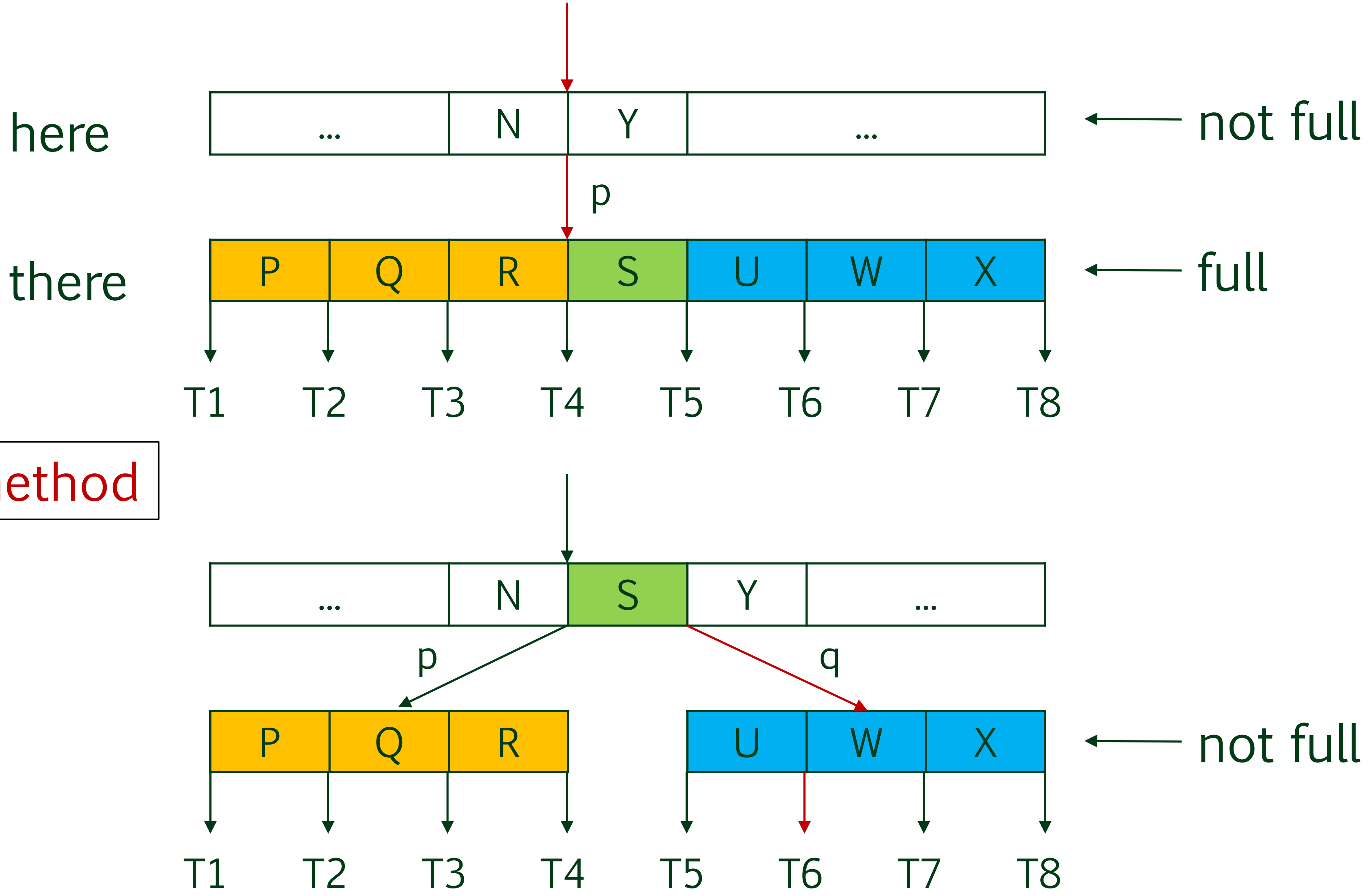
Time complexity

- The length of the path from the root to a leaf node is O(log n)
- At each node along the path, it takes O(t) time to determine which path to follow
- Total time complexity is O(t log n)

Insert

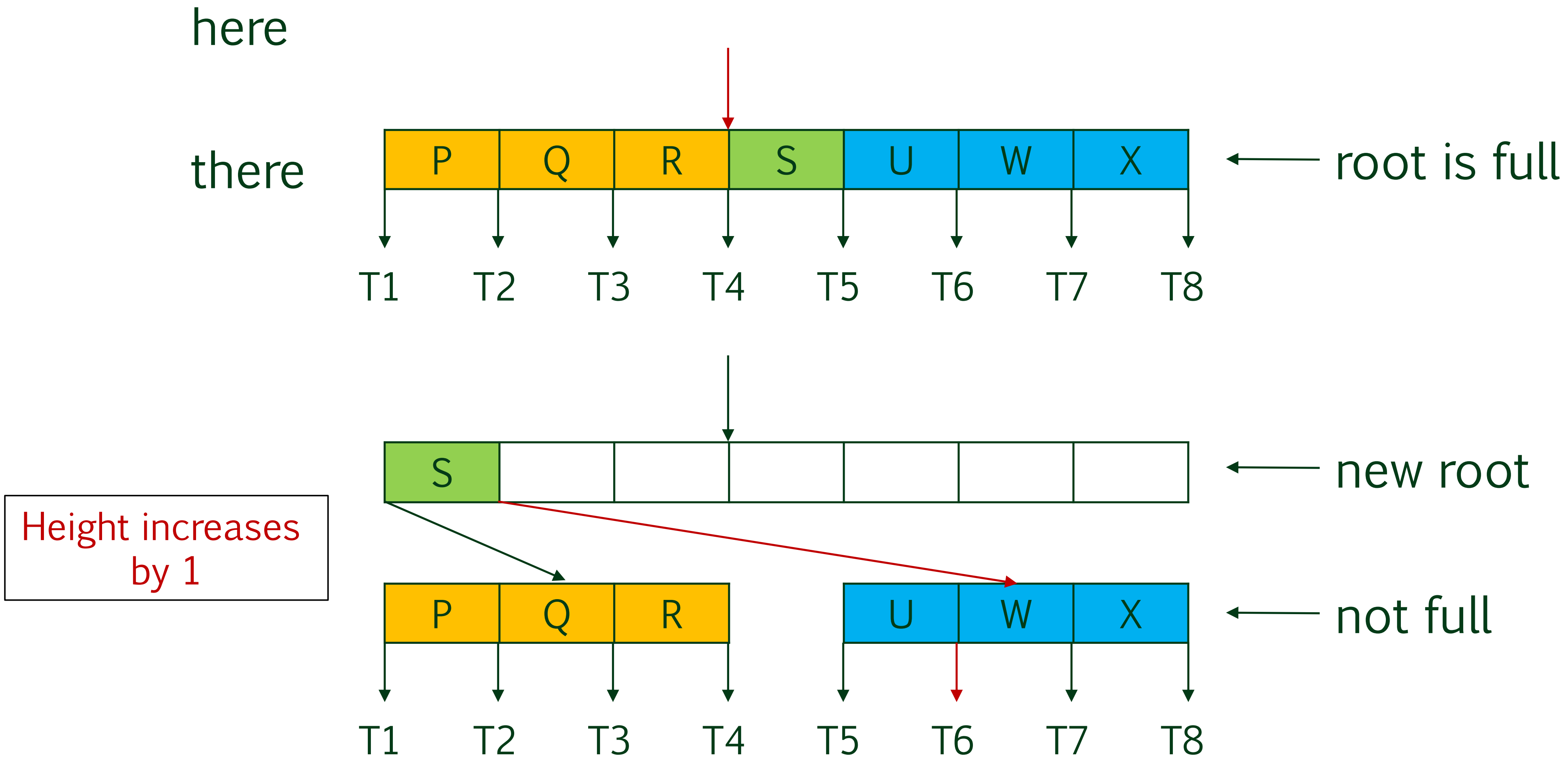
- Basic strategy
  - Descend along the path based on the given key k from the root to a leaf node. Each node along the path though cannot be full i.e., have 2t-1 keys. Therefore, before moving down to a node p (including the root), check first if p is full. If p is full then split p into two where the last t-1 keys of p are placed in a new node q and the median (middle) key of p is moved up to the parent node. Note: The parent node is guaranteed to have enough space to store the extra key (Why?)
  - Insert the given key k at the leaf node. If the key is found along the path from the root to the leaf node, then no insertion takes place.

Insert V into a B-tree (t=4)

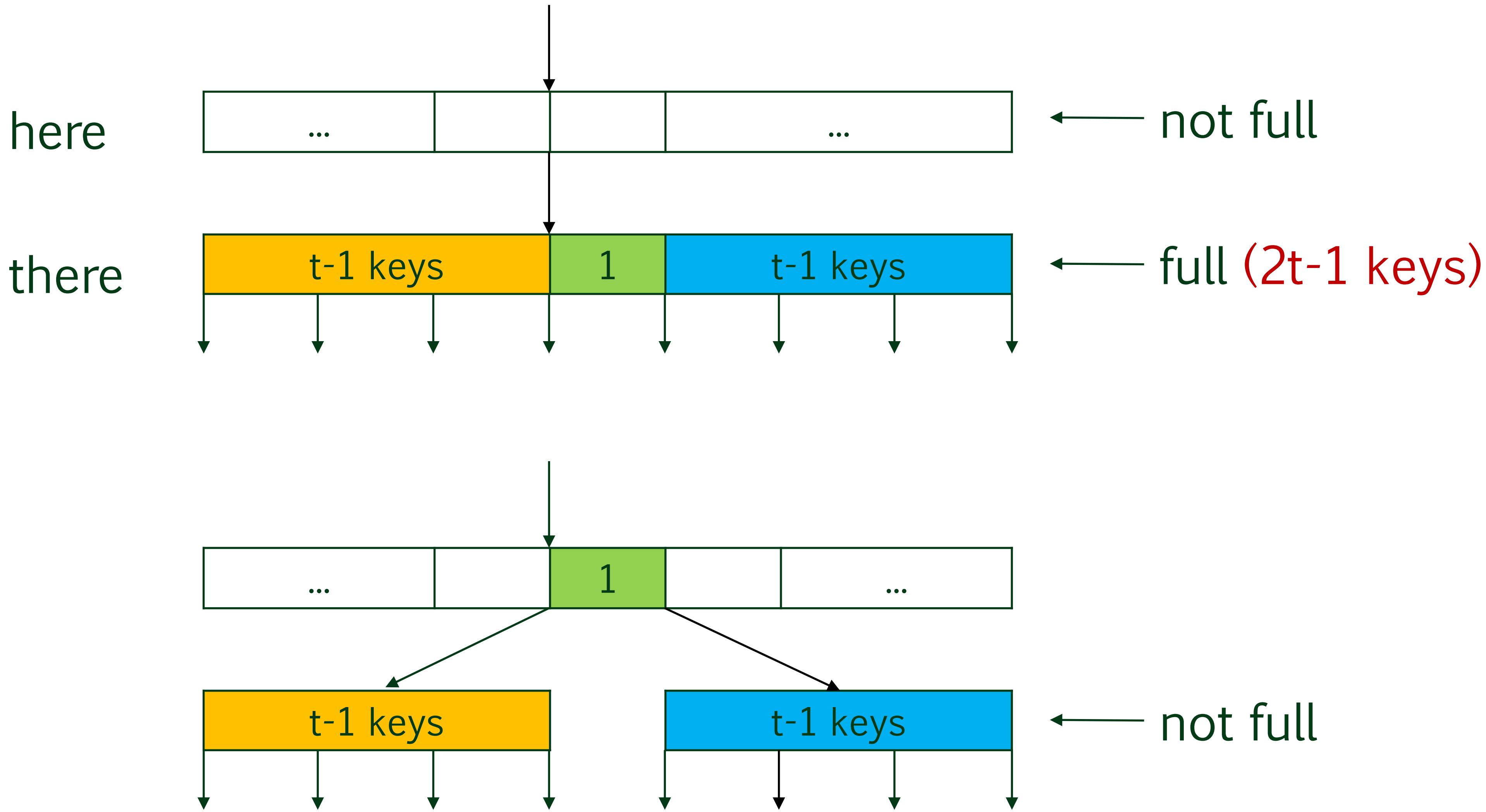


Split method

Insert V into a B-tree (t=4) when the root is full

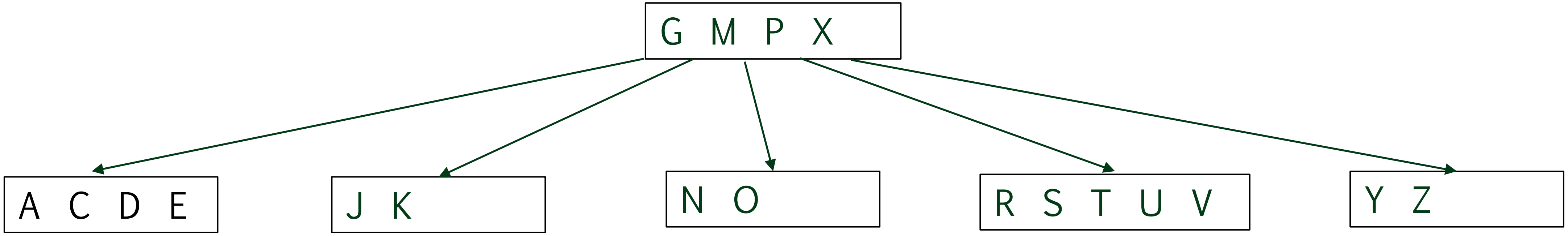


In general

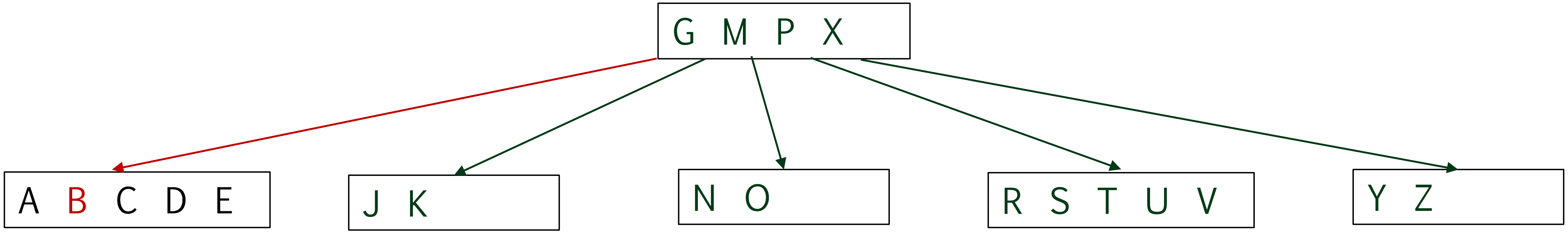


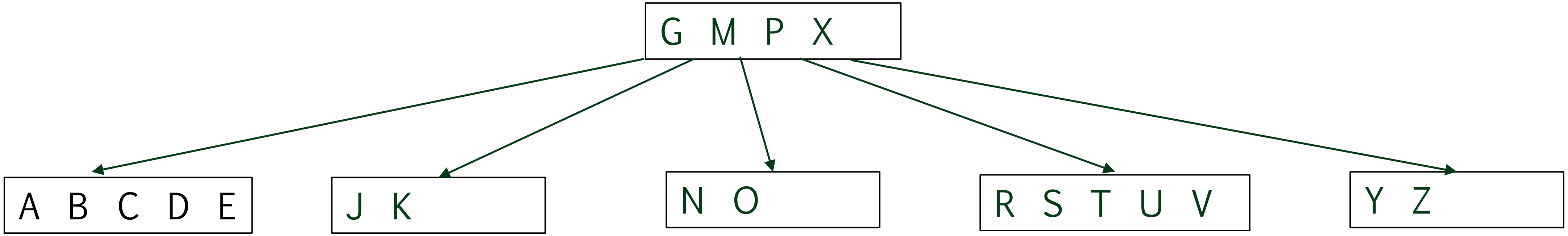
here
…
…
not full
there
t-1 keys
1
t-1 keys
full (2t-1 keys)
…
1
…
t-1 keys
t-1 keys
not full

Initial B-Tree (t=3) => Do not descend to nodes with 5 keys

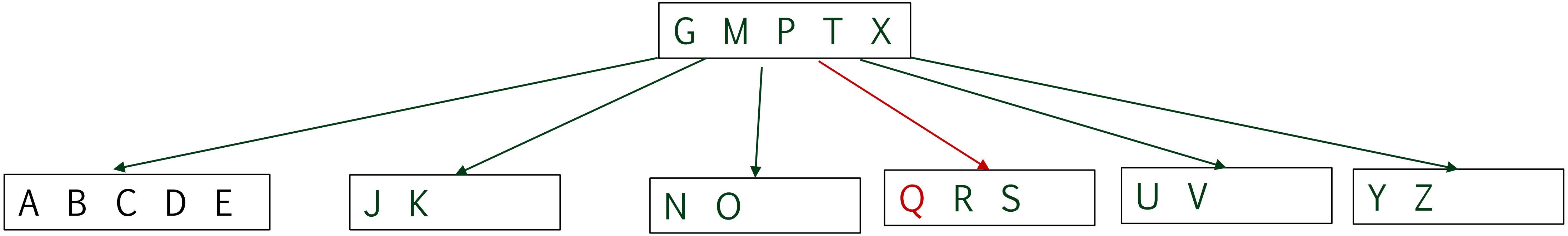


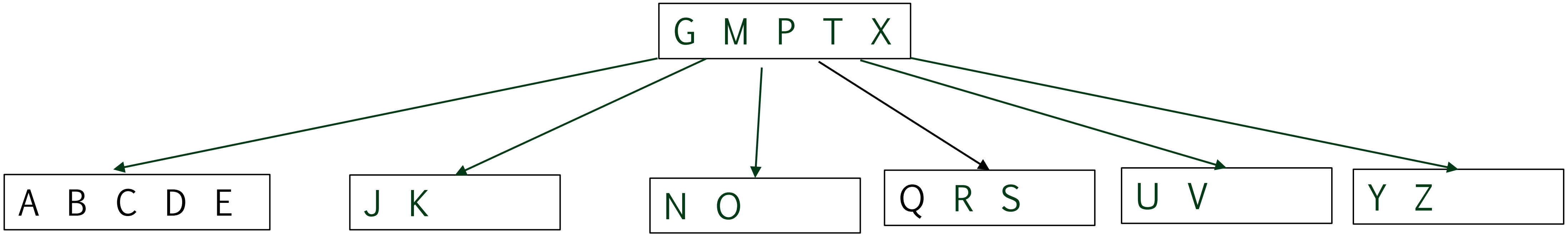
Insert B



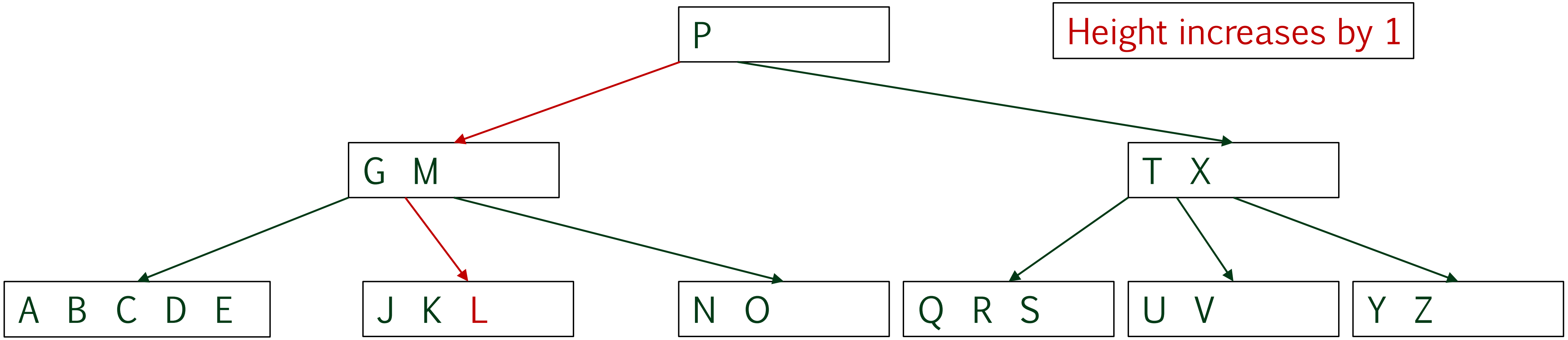


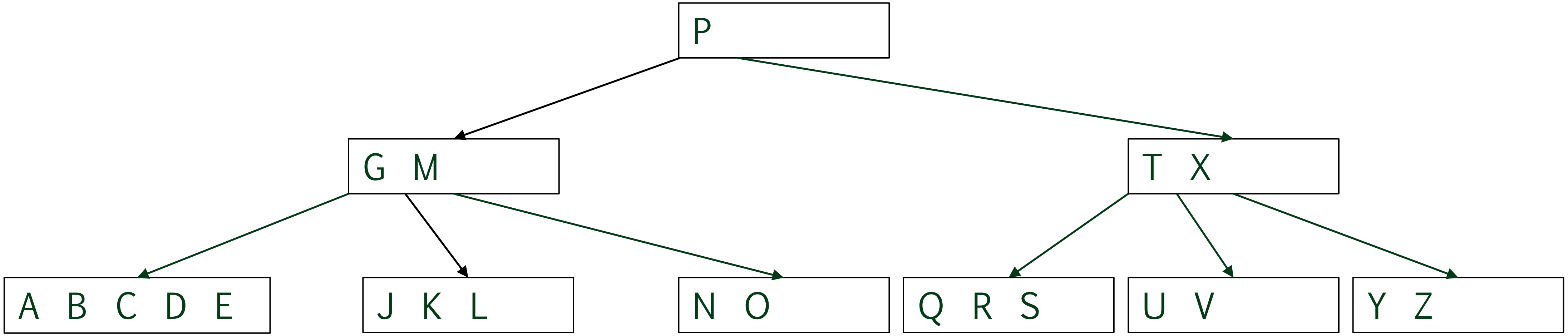
Insert Q



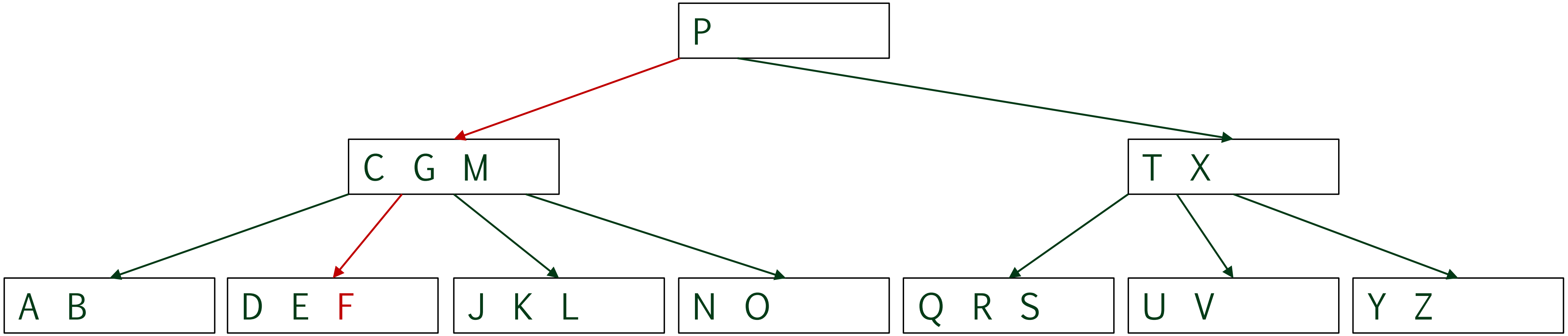

G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z

Insert L


P
Height increases by 1
G M
T X
A B C D E
J K L
N O
Q R S
U V
Y Z



Insert F



Observations

- Only descend to a node if it is not full
- The B-tree only grows in height when a key is inserted into a B-tree with a full root i.e., when the root node is split
- When a node is split into two, each of the two resultant nodes is assigned the minimum number of keys i.e., (t-1)

Time complexity

- The length of the path from the root to a leaf node is O(log n)
- At each node along the path, it takes O(t) time:
  - To determine which path to follow and
  - To split a node (if required)
- Total time complexity is O(t log n)

Exercises

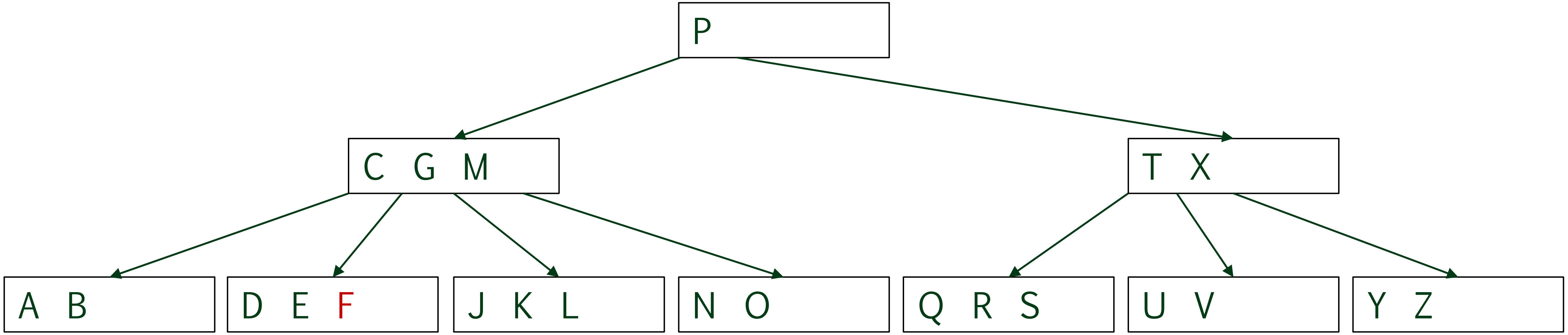
- Randomly insert a permutation of the alphabet into initially empty B-trees (t=2, t=3). Call the resultant trees B2 and B3.
- Calculate the minimum and maximum heights of B-trees (t=2, t=3) for 26 keys.
- Explain how to find the predecessor of a given key in a B-tree.

Delete

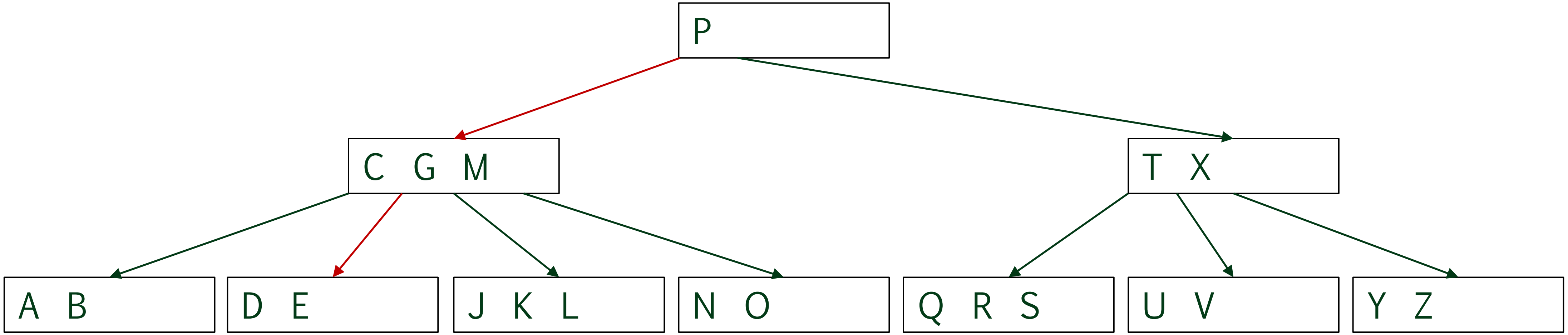
- Basic strategy
  - Descend along the path based on the given key k from the root until the key is found. Each node along the path though must have at least t keys. Therefore, before moving down to a node p (excluding the root), check if p has at least t keys. If not, node p either:
    - borrows a key from a sibling (if possible) or
    - merges with an adjacent node where the t-1 keys of each node plus one from the parent node yield a single node with 2t-1 keys. Note: The parent node is guaranteed to have an extra key (Why?)

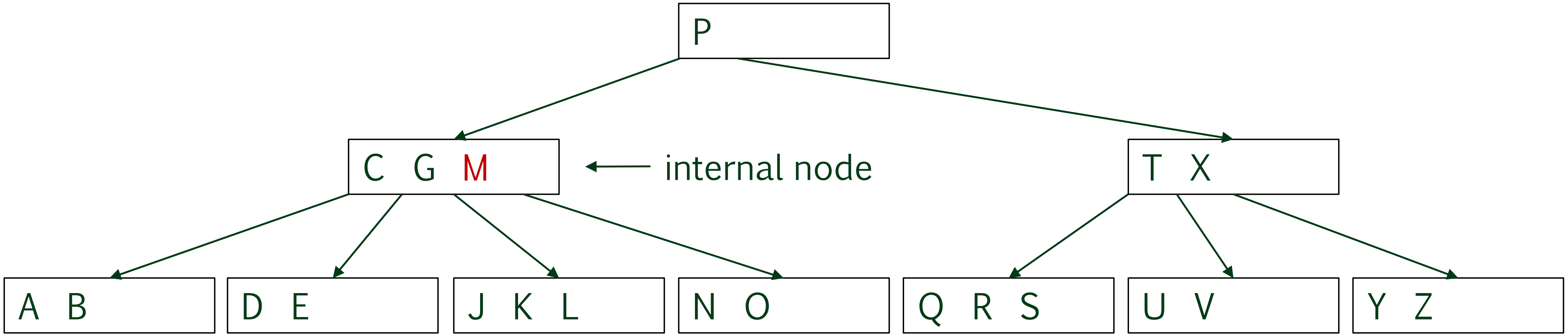
- Basic strategy (cont'd)
  - If the given key k is found at a leaf node, delete it. If the given the key k is found at an internal node p, then:
    - if the child node q that precedes k has t keys, then recursively delete the predecessor k' of k in the subtree rooted at q and replace k with k'.
    - if the child node r that succeeds k has t keys, then recursively delete the successor k' of k in the subtree rooted at r and replace k with k'.
    - otherwise, merge q and r with k from the parent to yield a single node s with 2t-1 keys. Recursively delete k from the subtree s.
  - If key k is not found, no deletion takes place.

Initial B-Tree (t=3) => Do not descend to nodes with 2 keys or less

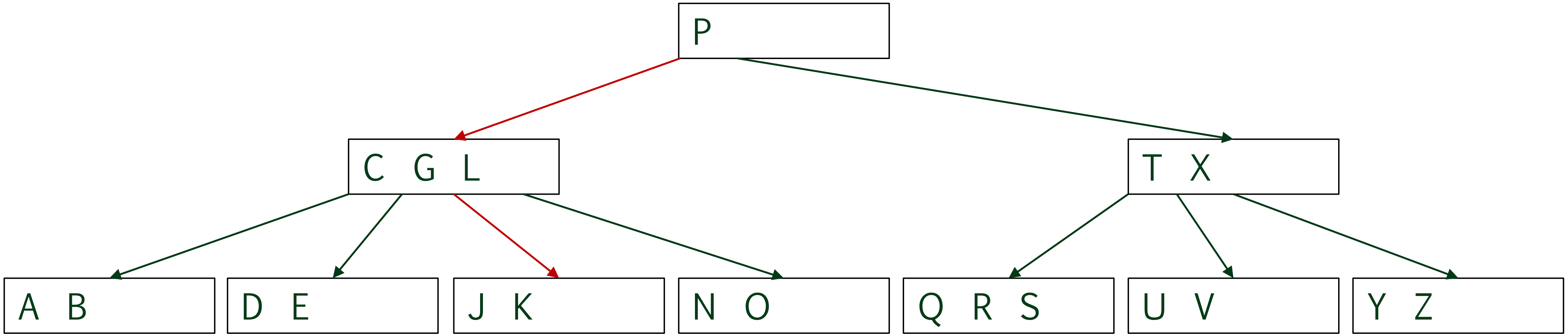


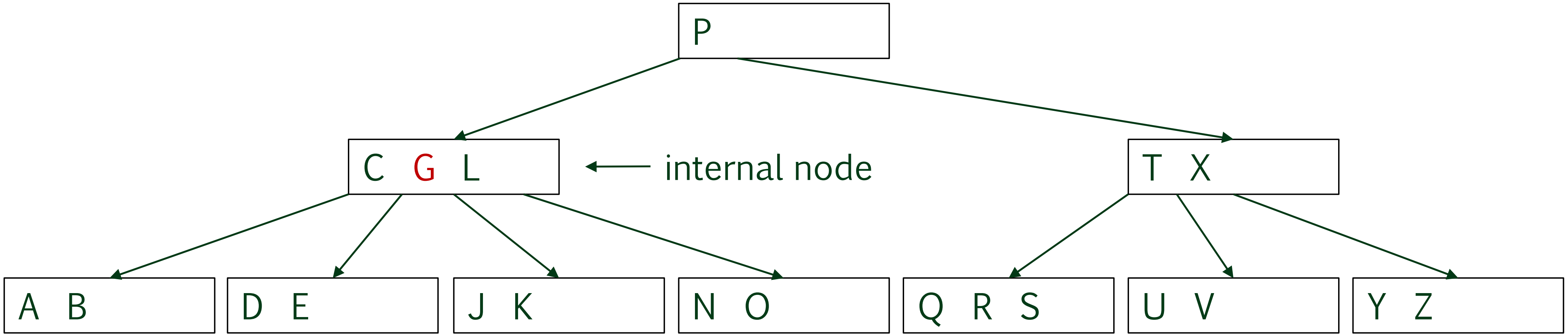
Delete F




P
C G M
internal node
T X
A B
D E
J K L
N O
Q R S
U V
Y Z

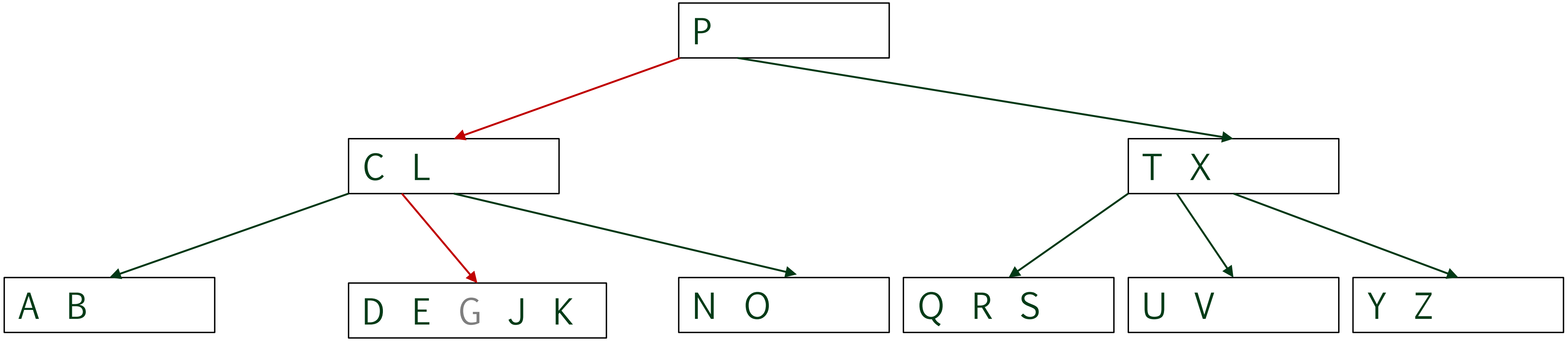
Delete M


P
C G L
T X
A B
D E
J K
N O
Q R S
U V
Y Z

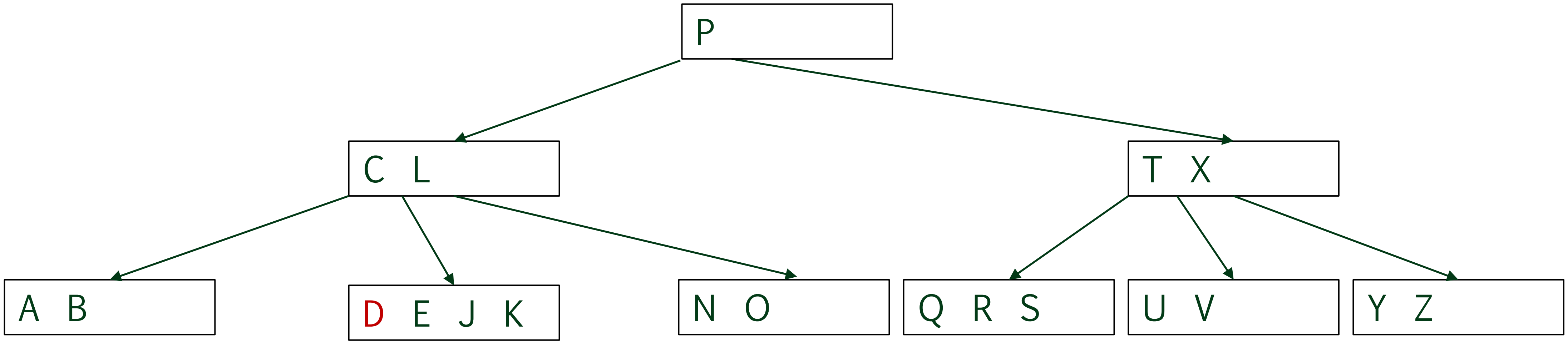


P
C G L
internal node
T X
A B
D E
J K
N O
Q R S
U V
Y Z

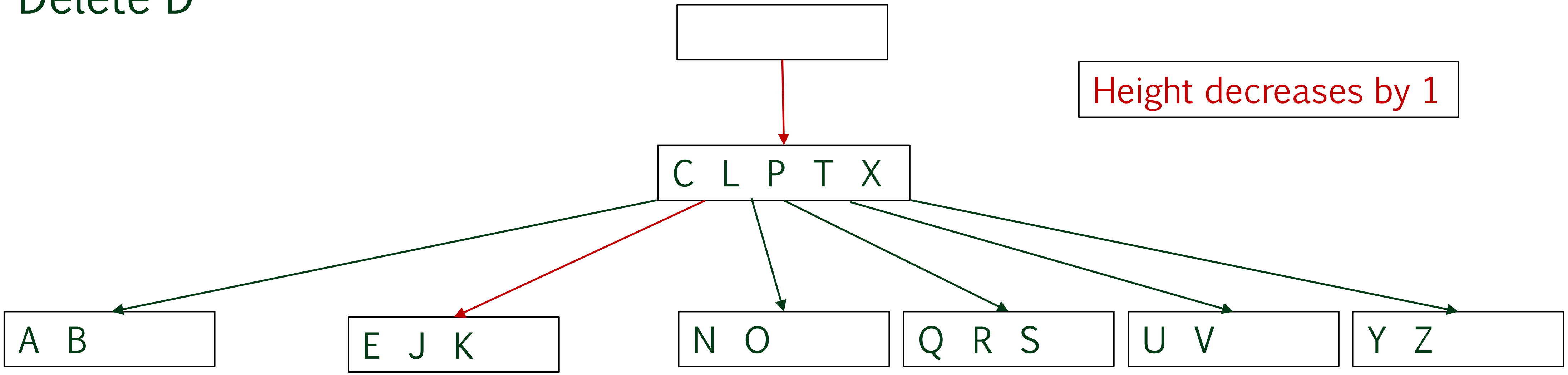
Delete G

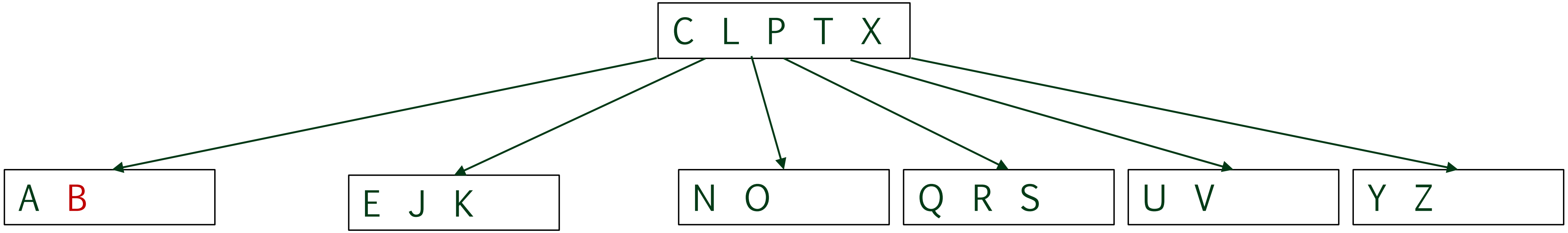


P
C L
T X
A B
D E G J K
N O
Q R S
U V
Y Z

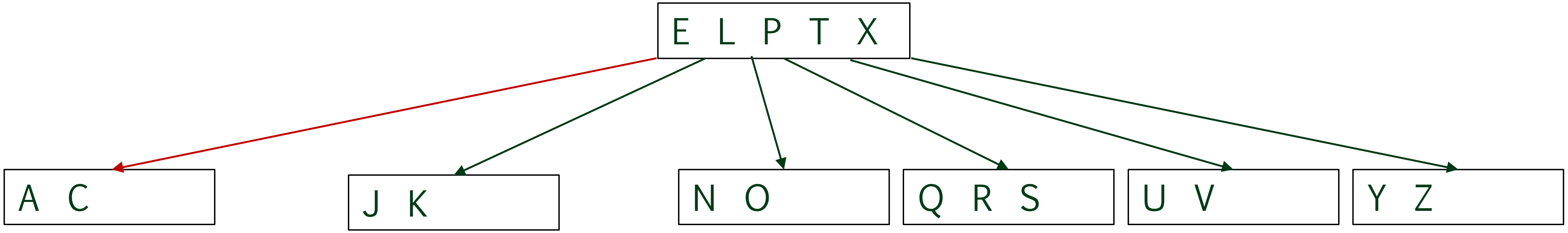

P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z

Delete D


C L P T X
Height decreases by 1
A B
E J K
N O
Q R S
U V
Y Z



Delete B



Observations

› The height of a B-tree only decreases if:

- the root node has a single key, and
- its two child nodes have t-1 keys each.

In this case, the two child nodes along with the key at the root are merged into one node with 2t-1 keys. This node now serves as the new root of the B-tree.

Time complexity

- Although the Delete method is quite complicated, its time complexity is the same as Search and Insert, $O(t \log n)$.

Exercises

- Successively delete in the same order the keys that were inserted to construct B2 and B3 (from the last exercise).
- Show why a node may need to be revisited (reread from disk) in the Delete method.
- Show that the number of read and write disk accesses is O(h) for Search, Insert, and Delete where h is the height of the B-tree.