Lecture 22: B Trees I

2023/10/12

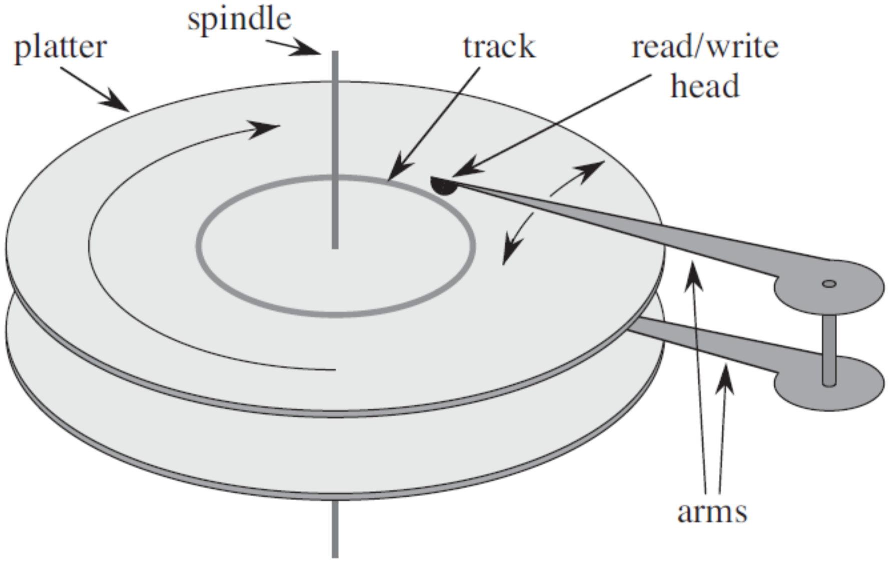
詹博华（中国科学院软件研究所）

B Trees

- Another (balanced) search tree data structure. Aim to minimize disk I/O operations for trees stored on a disk.
- Applications in database systems.
- Large branching factor, compared to 2 for binary search trees.

Properties of disk drive

- Read/write time significantly longer than main memory.
- Disk access not just one item but pages at a time, with each page 2^{11} to 2^{14} bytes (2kb – 16kb) in length.
- Design algorithms to minimize number of disk I/O (rather than total number of computation steps).



Aside: big-O notation: What is an operation?

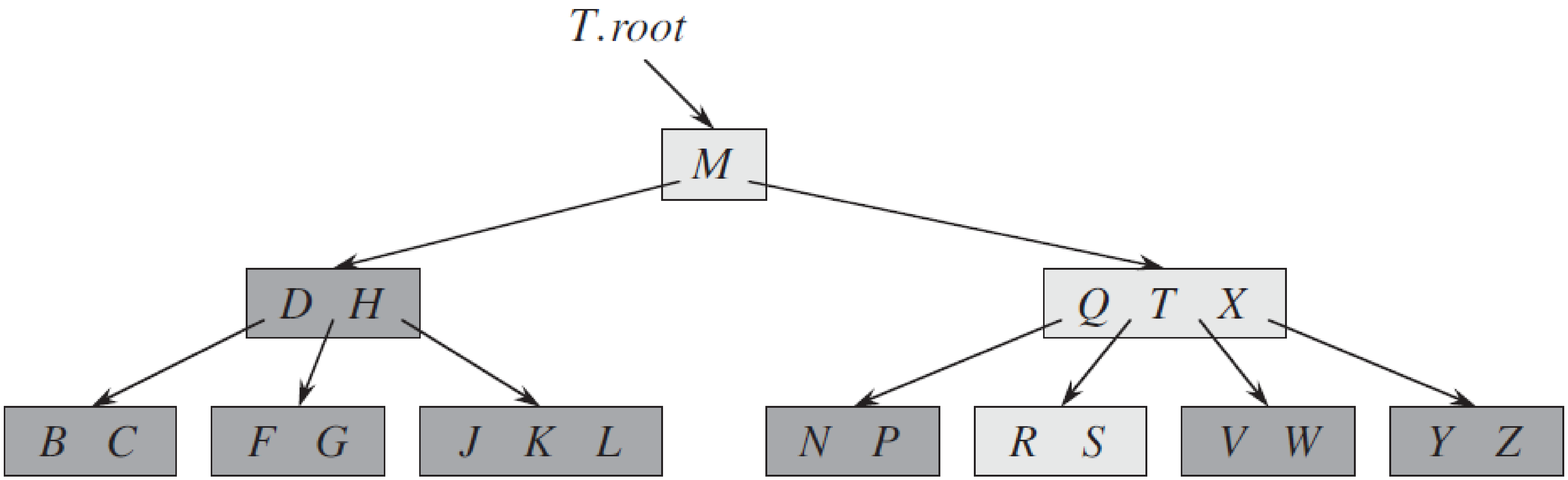
- We often say an algorithm is $O\big(f(n)\big)$, meaning the number of operations (or computation steps) grows as at most $c \cdot f(n)$. This depends on what counts as an operation.
- For example, if an algorithm involve multiplication of integers. We may choose to consider one multiplication $x \cdot y$ as one step, or as several steps, the number of which depends on the size of x and y (e.g. $O(\log(x) \cdot \log(y))$ steps).
- In our case, we only consider read/write from disk as operations, and ignore the other computations (this can be added back in a more refined analysis).

Properties of B Trees

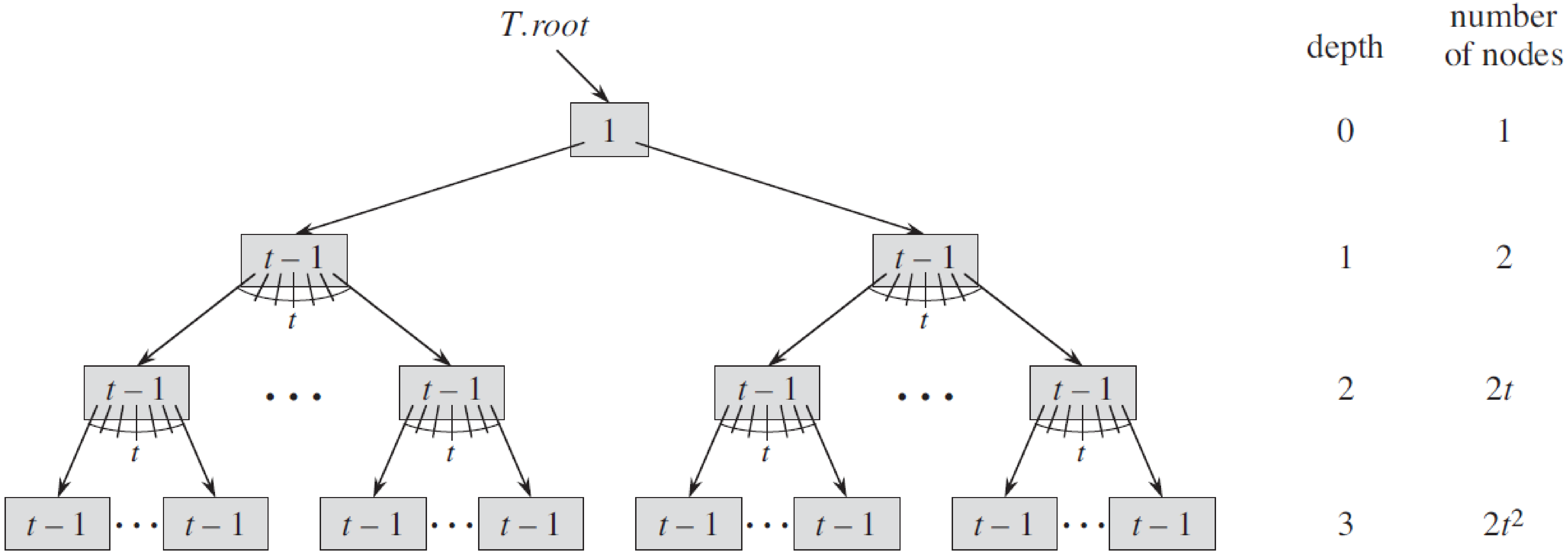
- Each node x stores the number of keys $x.n$, and a list of keys
$$x.key_1 \leq x.key_2 \leq \cdots \leq x.key_n.$$
- Each node is either internal node or a leaf. Each internal node contains $x.n + 1$ pointers to its children.
- The keys separate the range stored in each subtree.
- All leaves have the same depth h.
- Given a minimum degree t, each node other than the root must have at least $t - 1$ keys. Each node contains at most $2t - 1$ keys.

Example of B Trees

- B Tree with $h = 3$ and $t = 2$ (minimum 1, maximum 3 entries per node).



Height vs. size of B Trees



Height vs. size of B Trees: Computation

- Lower bound on size in term of height:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\frac{t^h - 1}{t-1} = 2t^h - 1$$

- This implies:

$$h \leq \log_t \frac{n+1}{2}$$

Search in B-Trees

- Standard procedure yields $O(h) = O(\log_t n)$ algorithm (in the number of read/write of blocks).
- Note however the while loop on line 2-3, giving an extra factor of t if we also consider these computations.

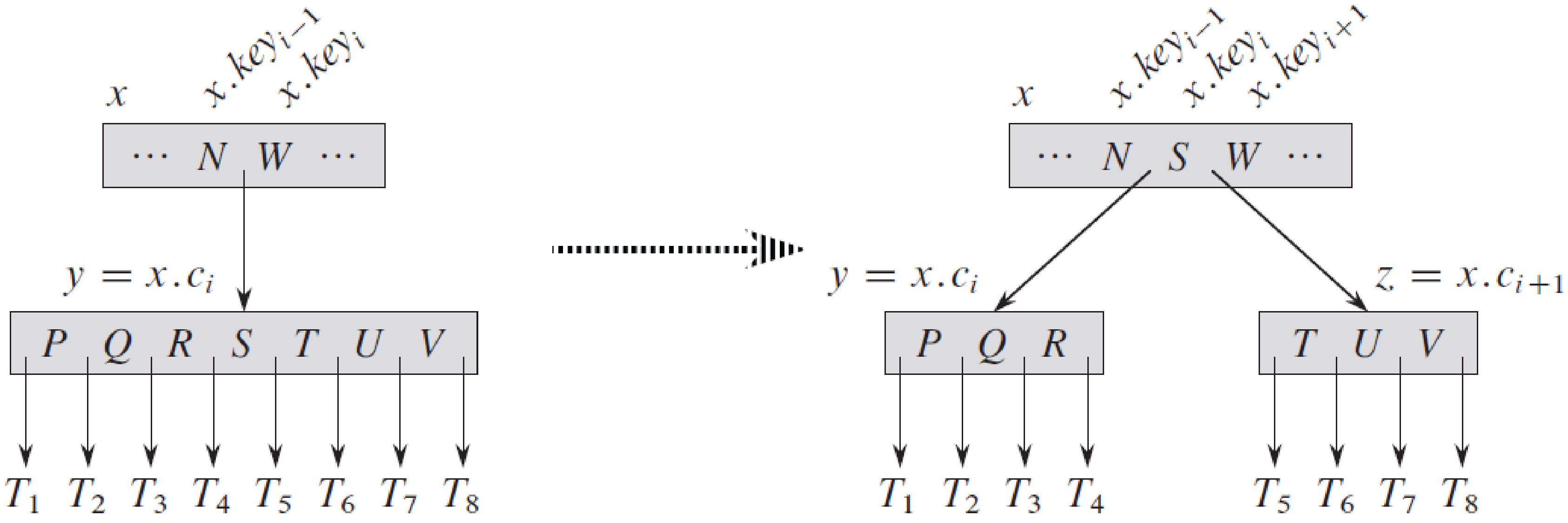
```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3      i = i + 1
4  if i ≤ x.n and k == x.key_i
5      return (x, i)
6  elseif x.leaf
7      return NIL
8  else DISK-READ(x.c_i)
9      return B-TREE-SEARCH(x.c_i, k)
```

Insertion in B-Trees: Naïve solution

- Insert as in ordinary search tree.
- Split nodes that become too large in the process.
- Require backing-up the tree – extra constant factor in number of disk I/O's.

Splitting of nodes

- Starting from a node that is full ($2t - 1$ nodes).
- End with two nodes that have minimum size ($t - 1$ nodes).
- Example for $t = 4$:

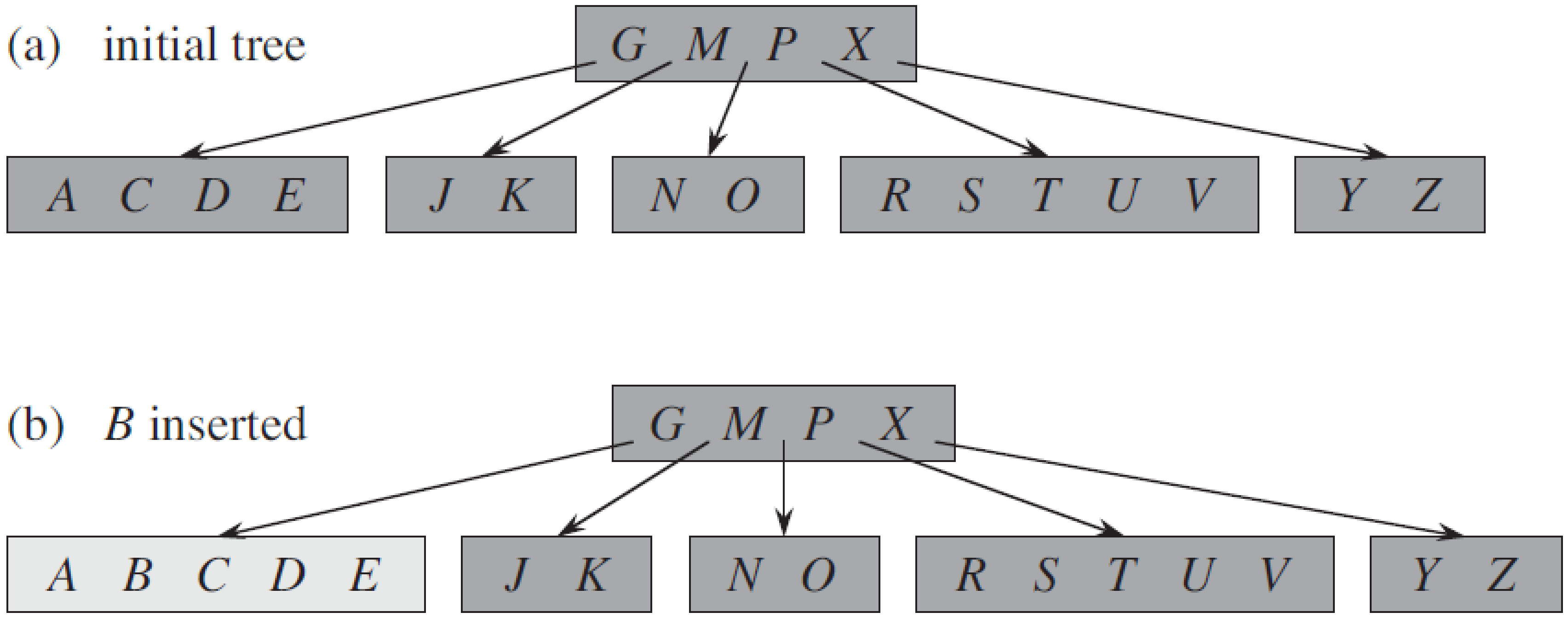


Insertion in a single pass

- During the search for location of insertion, split any full node that is encountered.
- This ensures that insertion is always made to non-full nodes.

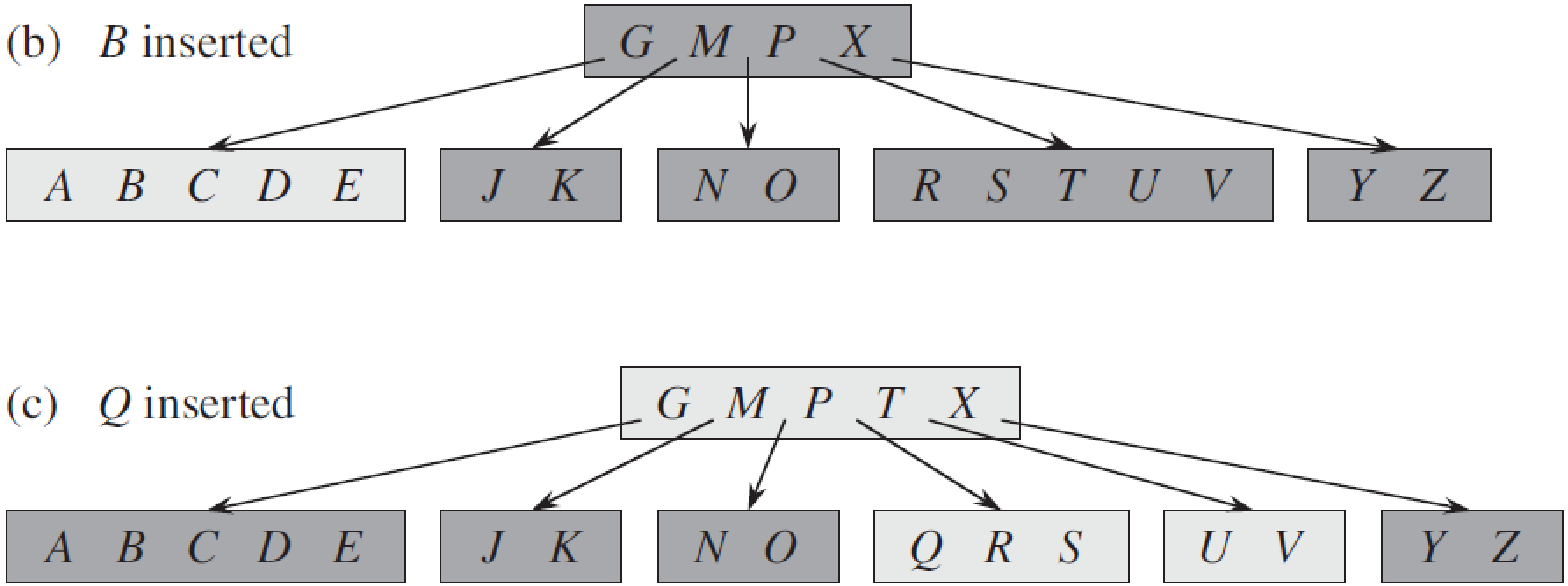
Insertion: example ($t = 3$)

- Insert B, resulting in a full leaf.



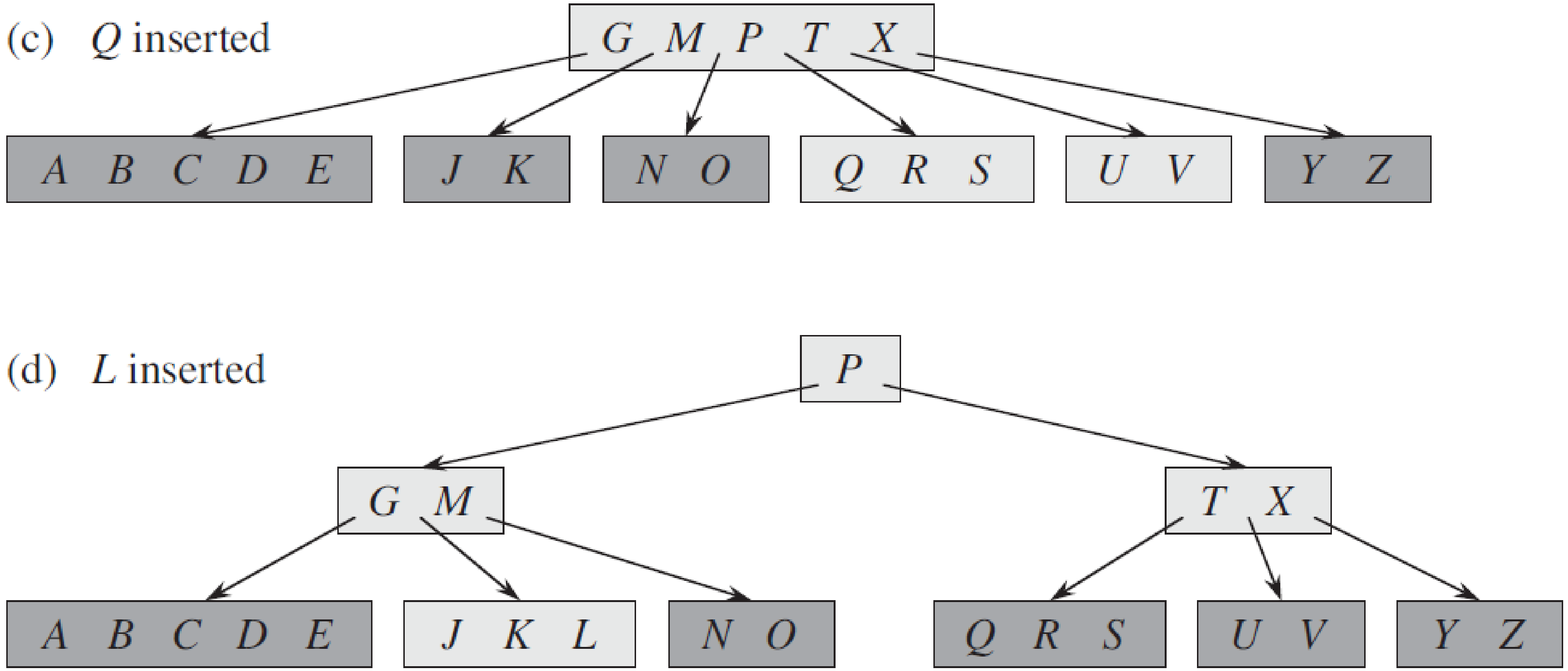
Insertion: example

- To insert Q, first split the node $\langle R\ S\ T\ U\ V \rangle$, moving T to the root.



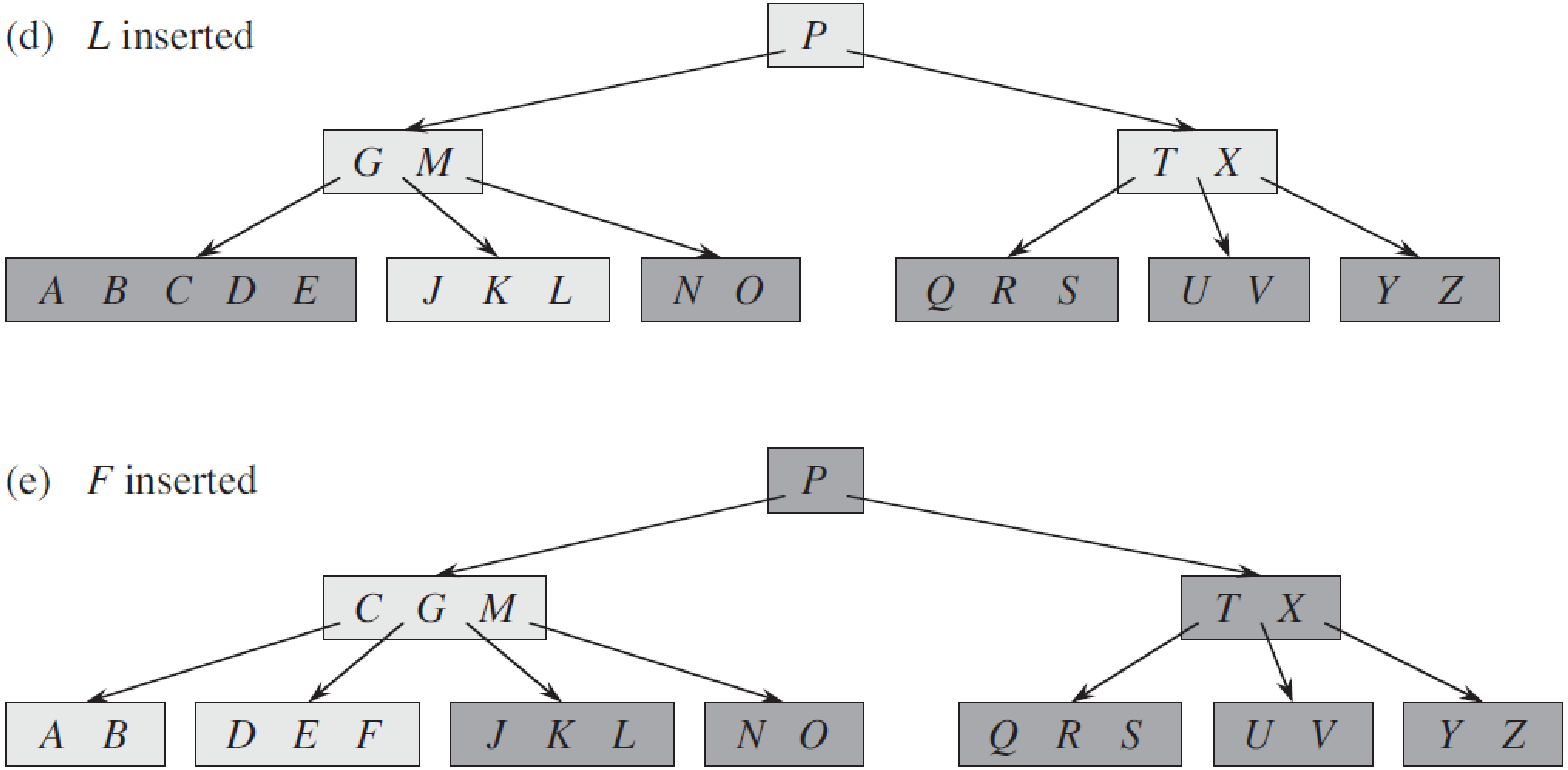
Insertion: example

- Before insertion L, first split root node $\langle G\ M\ P\ T\ X \rangle$.



Insertion: example

- Before inserting F, first split node $\langle A\ B\ C\ D\ E \rangle$.



Insertion: implementation

- Line 2-8: when a full node is encountered, split node.
- Line 9, 10: insert into non-full nodes.

```
B-TREE-INSERT(T, k)
1  r = T.root
2  if r.n == 2t - 1
3      s = ALLOCATE-NODE()
4      T.root = s
5      s.leaf = FALSE
6      s.n = 0
7      s.c1 = r
8      B-TREE-SPLIT-CHILD(s, 1)
9      B-TREE-INSERT-NONFULL(s, k)
10 else B-TREE-INSERT-NONFULL(r, k)
```

Insertion: non-full case

- Line 2-8: leaf case.
- Line 9-17: non-leaf case.

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3      while i ≥ 1 and k < x.key_i
 4          x.key_{i+1} = x.key_i
 5          i = i - 1
 6      x.key_{i+1} = k
 7      x.n = x.n + 1
 8      DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10          i = i - 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t - 1
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```