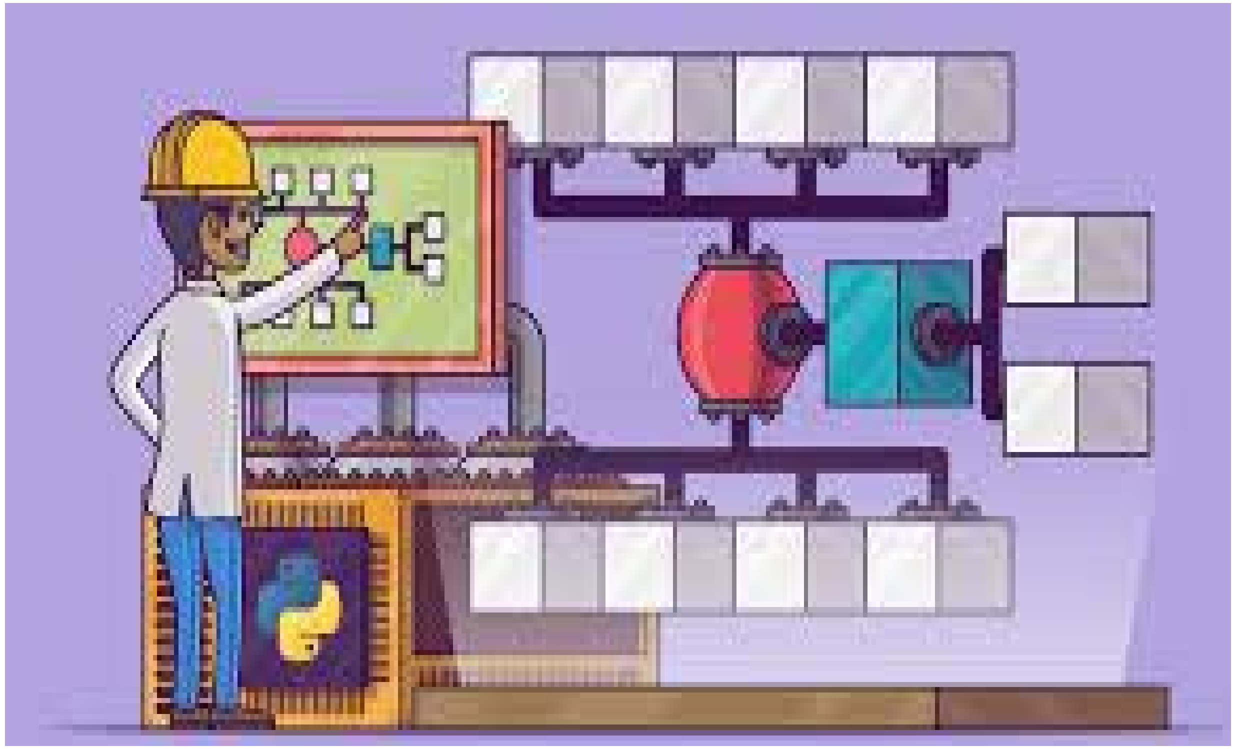
ساختمان داده ها



مدرس:
سمانه حسینی سمنانی

دانشگاه صنعتی اصفهان- دانشکده برق و کامپیوتر

درخت ها

- مفاهیم اولیه
- پیمایش درخت
- درخت دودویی معادل
- پیاده سازی درخت
- درخت جستجوی دودویی
- درخت عبارت
- Heap tree (هرم بیشینه)

درخت ها

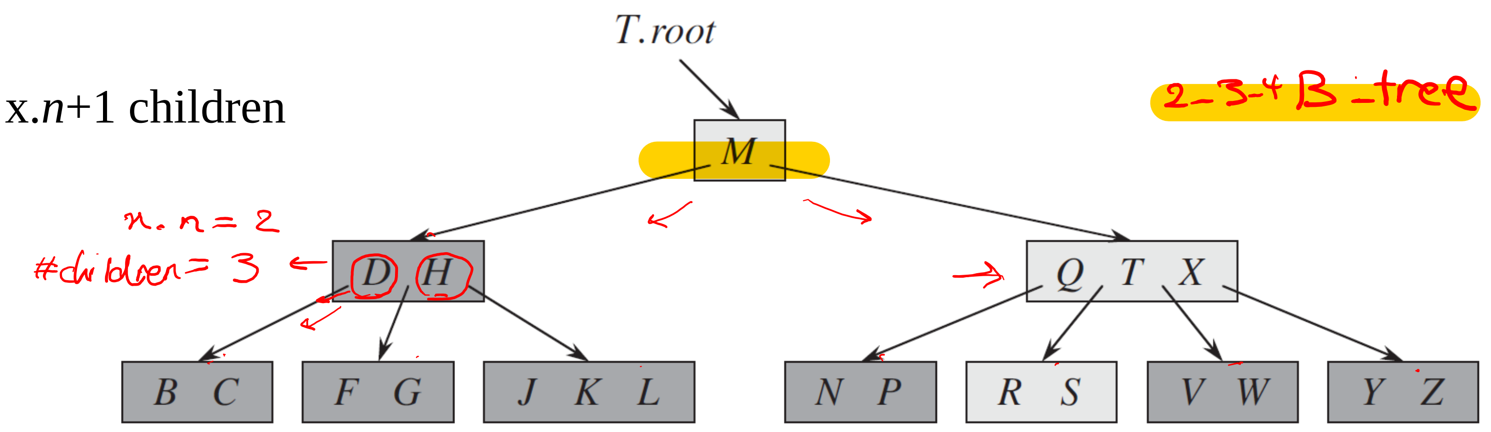
- Red-black tree
- AVL tree
- B-Trees

B-Tree

- balanced search trees designed to work well on disks or other direct access secondary storage devices.
- B-trees differ from red-black and AVL trees in that B-tree nodes may have many children, from a few to thousands.
- B-trees are similar to red-black and AVL trees in that every n-node B-tree has height O(lg n).

An internal node x containing x.*n* keys has x.*n*+1 children

All leaves are at the same depth in the tree.



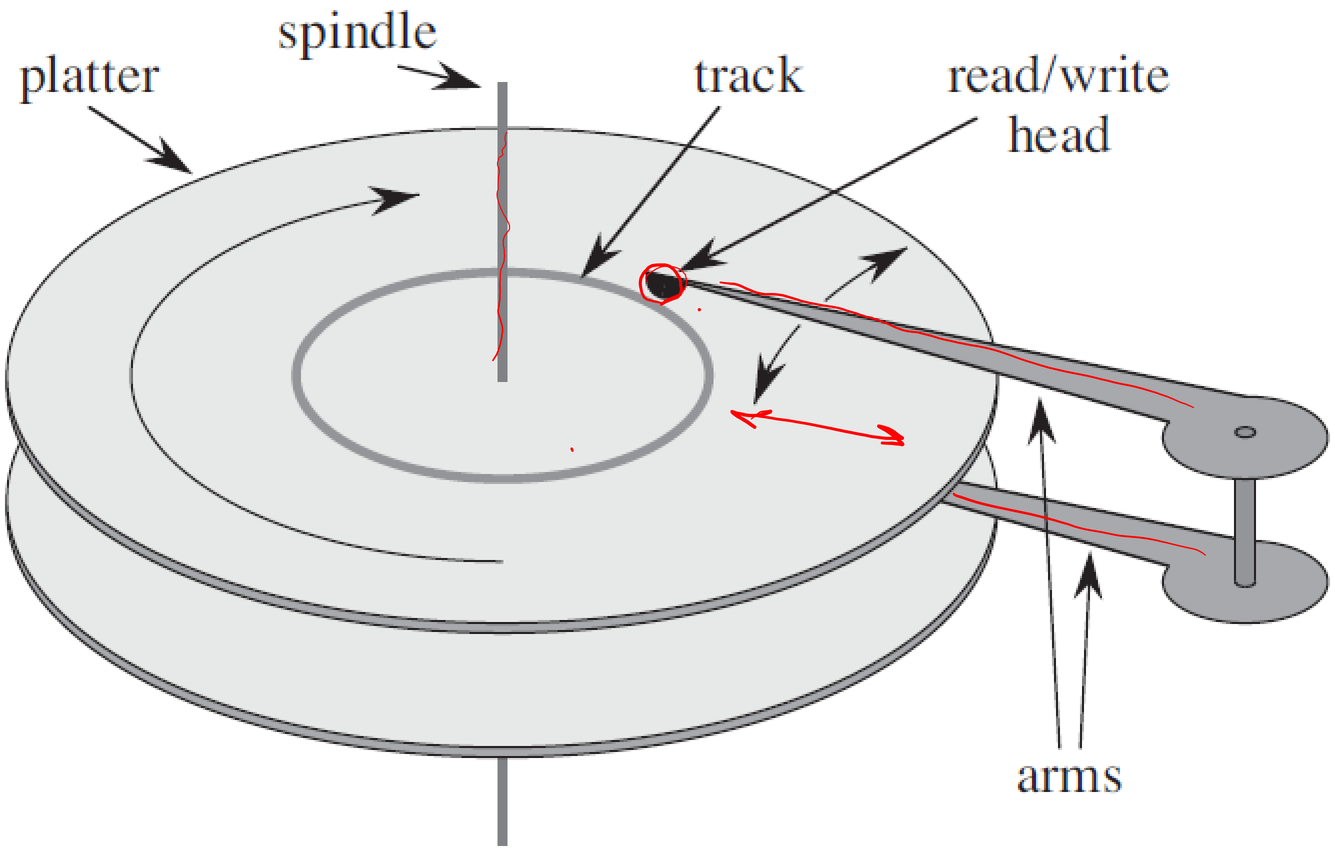
Secondary Storage

- برای سیستم‌هایی که بلاک‌های عظیم اطلاعات را خوانده و می‌نویسند بهینه‌سازی شده‌است.
- معمولاً در پایگاه داده استفاده می شود.
- گره‌های درونی (و نه برگ‌ها) می‌توانند یک شماره متغیر از محدوده‌ای ازپیش‌تعریف‌شده مربوط به گره‌های فرزند را اختیار کنند.
- زمانی که داده‌ها درج شده یا از یک گره حذف می‌شوند، شماره گره‌های فرزند آن‌ها تغییر می‌کند.
- به منظور نگه‌داری محدوده ازپیش تعریف‌شده، ممکن است گره‌های درونی به هم متصل شده یا از هم جدا شوند.
- به دلیل اینکه محدوده‌ای از گره‌های فرزند مجاز هستند، درخت بی، همانند دیگر درخت‌های جستجوی متوازن، نیازی ندارد که به صورت متناوب اقدام به برقراری توازن کند.

Secondary Storage

- اما به دلیل اینکه گره‌ها کاملاً پر نیستند، ممکن است مقداری حافظه هدر رود.
- یک درخت بی با استلزام اینکه همه برگ ها در یک عمق قرار داشته باشند، به صورت متوازن نگه داشته می‌شود.
- درخت‌های بی، هنگامی که زمان دسترسی به گره‌ها به میزان قابل توجهی بیشتر از زمان پیمایش بین دو گره باشد، مزیت‌هایی اساسی بر دیگر انواع پیاده‌سازی دارند.
- این اتفاق معمولاً زمانی رخ می‌دهد که گره‌ها در حافظه‌ای ثانویه مانند دیسک سخت قرار دارند.
- معمولاً تعداد فرزندان گره های داخلی طوری تنظیم می‌شود که هر گره، یک بلاک کامل از دیسک یا مقداری برابر از حافظه ثانویه را اشغال کند.

Secondary Storage



Although disks are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts

B-Tree

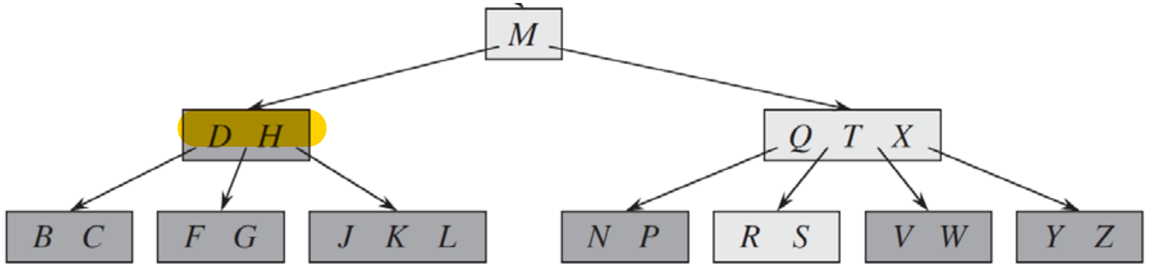
- A B-tree T is a rooted tree having the following properties:

1. Every node x has the following attributes:

   a. $x.n$, the number of keys currently stored in node x,

   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \le x.key_2 \le \cdots \le x.key_{x.n}$,

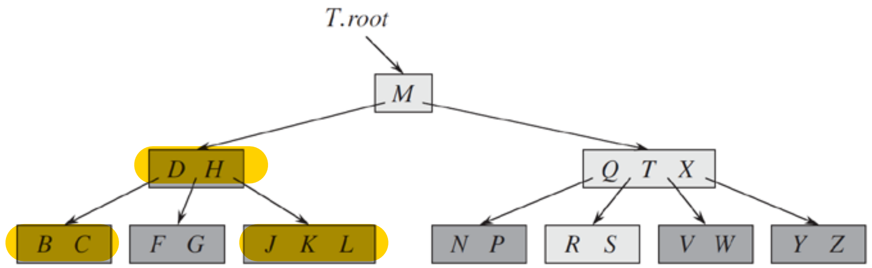
   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.



B-Tree

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

2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

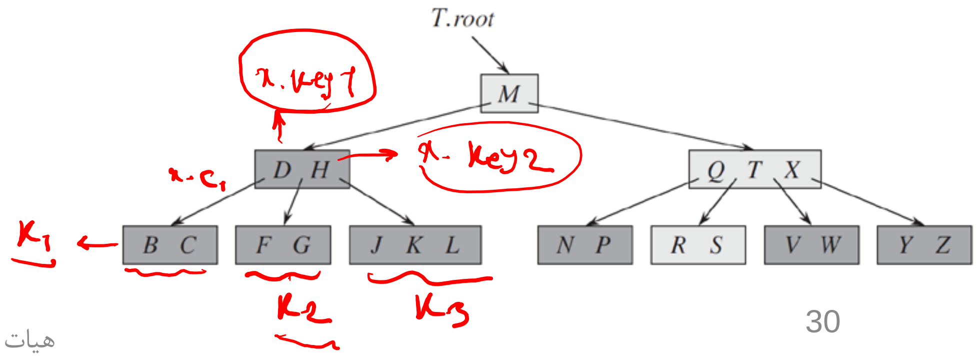


B-Tree

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

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

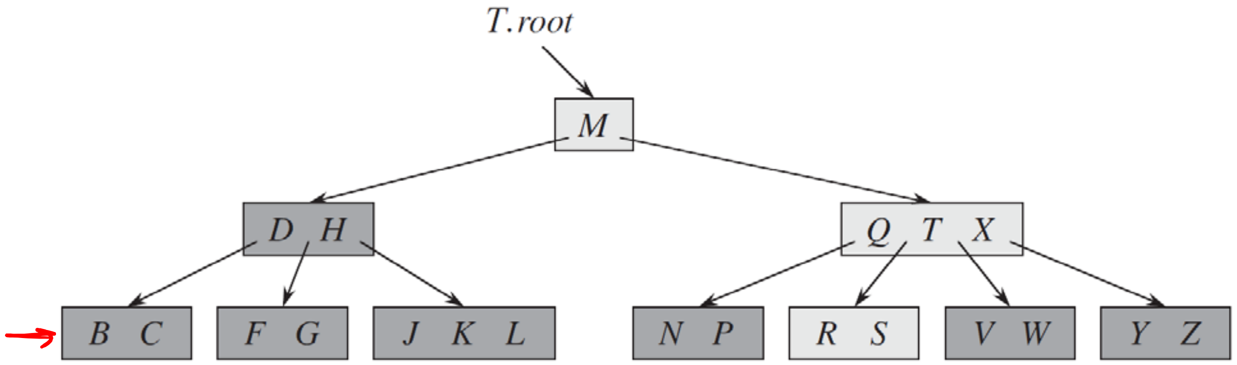
$$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \cdots \le x.key_{x.n} \le k_{x.n+1} .$$



B-Tree

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

4. All leaves have the same depth, which is the tree's height *h*.



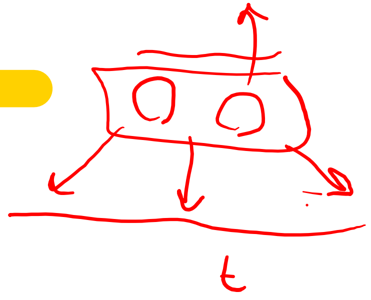
B-Tree

$\rightarrow t =$ تعداد فرزندان min در B-tree
$\rightarrow 2t =$ 〃 〃 〃 max

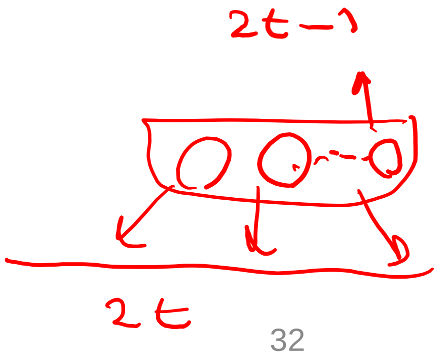
- A B-tree T is a rooted tree having the following properties:

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:

$t-1 =$ تعداد کلیدها



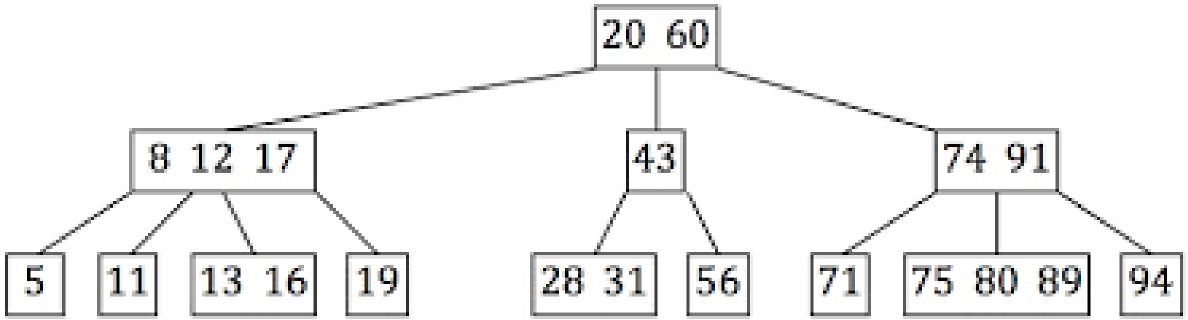
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.



   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.[2]

B-Tree

- The simplest B-tree occurs when t = 2.
- Every internal node then has either 2, 3, or 4 children.
- We have a 2-3-4 tree.



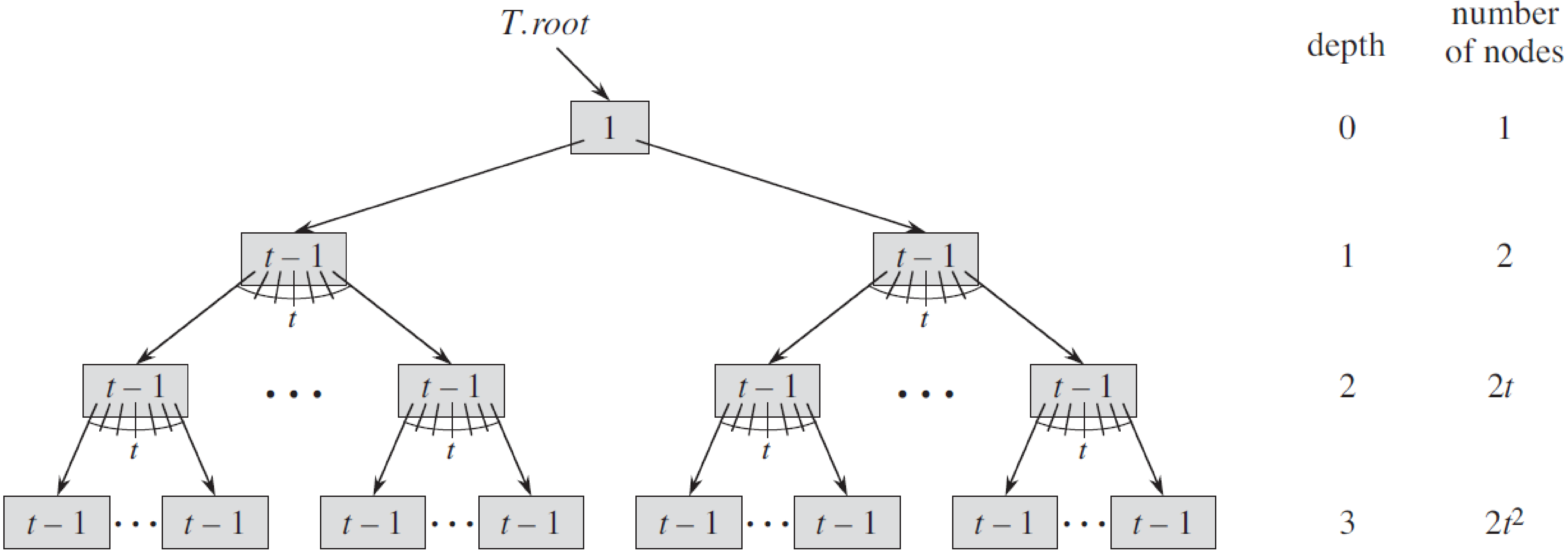
The height of a B-tree

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

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

The height of a B-tree

- The root of a B-tree T contains at least one key, and all other nodes contain at least t-1 keys.



The height of a B-tree

- The root of a B-tree T contains at least one key, and all other nodes contain at least t-1 keys.

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 . \end{aligned}$$

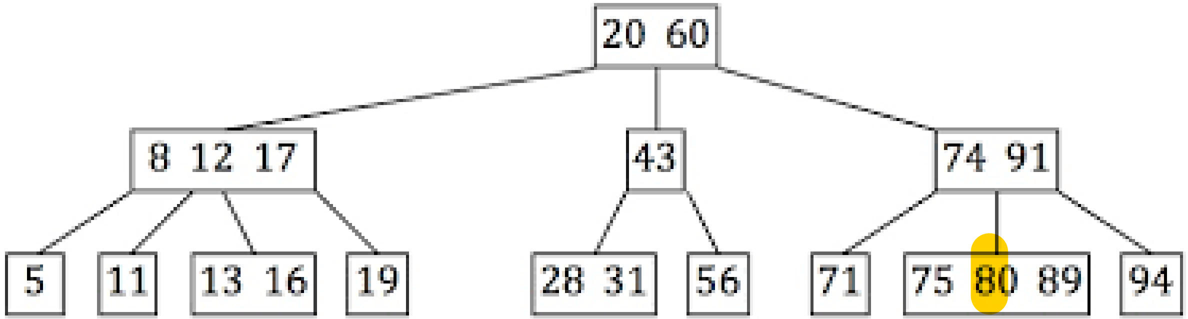
$$t^h \leq (n+1)/2 \longrightarrow h \leq \log_t \frac{n+1}{2} .$$

Basic operations on B-trees

- B-TREE-SEARCH
- B-TREE-CREATE
- B-TREE-INSERT
- B-TREE-DELETE

Searching a B-tree

- Binary search tree: “two-way,” branching decision
- B-tree: multiway branching decision according to the number of the node’s children.
- At each internal node x, we make an $(x.n+\ 1)$ −way branching decision



Searching a B-tree

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

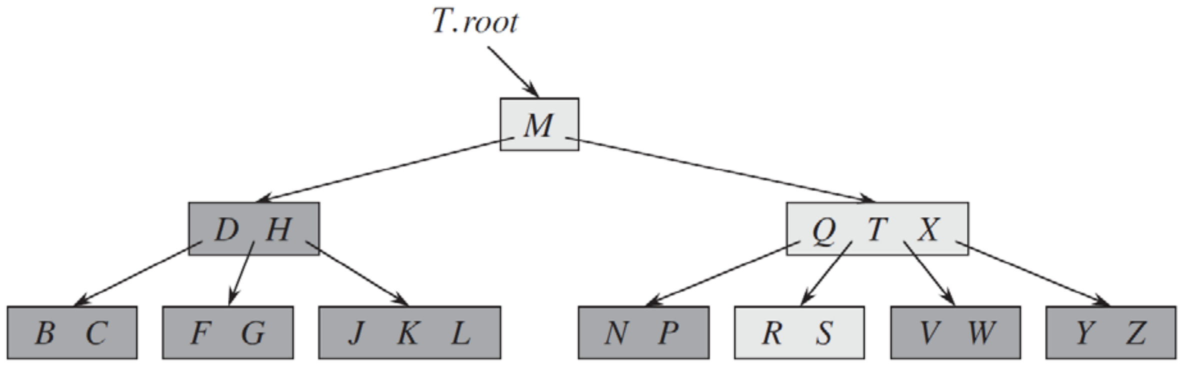
B-TREE-SEARCH($T.root$, k).

pair (y, i)

$y.key_i = k$

Searching a B-tree

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```



Basic operations on B-trees

- The B-TREE-SEARCH procedure accesses $O(h) = O(log_t^n)$ disk pages.
- h is the height of the B-tree and n is the number of keys in the B-tree.

Creating an empty B-tree

B-TREE-CREATE(T)

x = ALLOCATE-NODE()
$x.leaf$ = TRUE
$x.n = 0$
DISK-WRITE(x)
$T.root = x$

$O(1)$ disk operations

Inserting a key into a B-tree

- with binary search trees, we search for the leaf position at which to insert the new key.
- With a B-tree, however, we cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree

Inserting a key into a B-tree

- تمامی درج‌ها در برگ‌ها انجام می‌شود.

1. از طریق جستجوی درخت، برگی که عنصر جدید باید در آنجا اضافه گردد را می‌یابیم.
2. اگر این برگ، کمتر از بیشینه تعداد قابل قبول، عنصر داشت، برای یکی بیشتر فضا وجود دارد. با مرتب نگه‌داشتن عناصر گره، عنصر جدید را در این برگ درج می‌کنیم.
3. در غیراین‌صورت، برگ مورد نظر را به دو گره تقسیم می‌کنیم.
   1. میانه منحصربه‌فرد از بین عناصر برگ و عنصر جدید انتخاب می‌شود.
   2. میانه به عنوان مقدار جدایی عمل می‌کند و مقادیر کمتر از میانه در گره چپ جدید و مقادیر بزرگتر از میانه در گره راست جدید قرار داده می‌شوند.
   3. این مقدارِ جدایی به گره پدر اضافه می‌شود، که این کار ممکن است باعث تقسیم شدن پدر شود، و این روند به همین ترتیب ادامه می‌یابد.

Splitting a node in B-tree

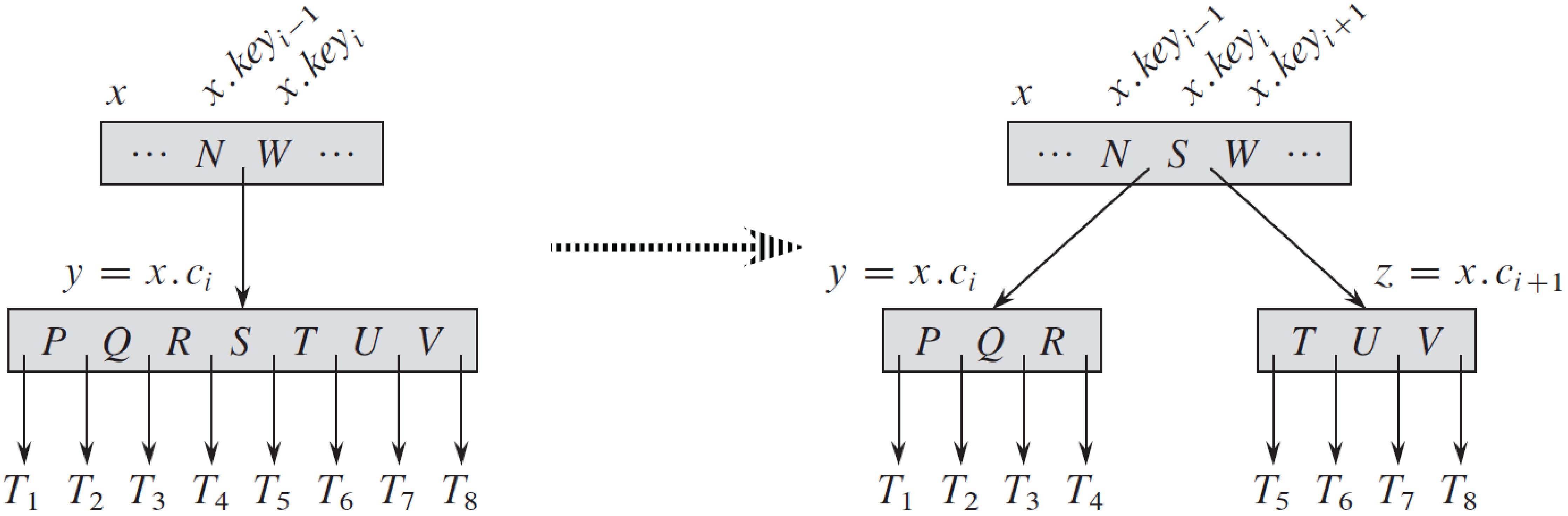


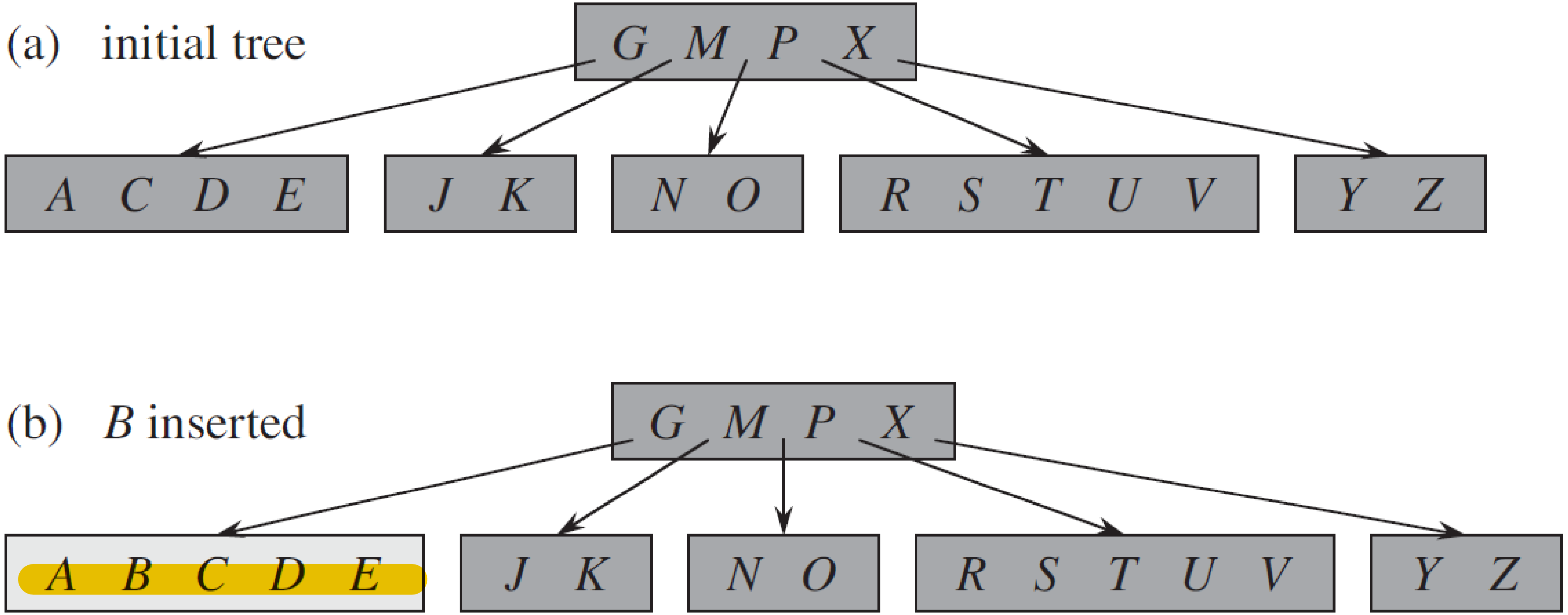
Figure 18.5 Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent.

Inserting a key into a B-tree

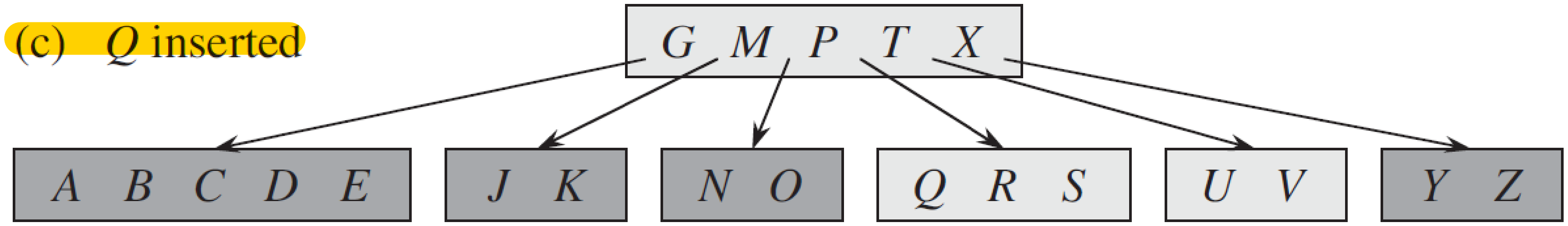
- we do not wait to find out whether we will actually need to split a full node in order to do the insertion.
- Instead, as we travel down the tree searching for the position where the new key belongs, we split each full node we come to along the way (including the leaf itself).
- Thus whenever we want to split a full node y, we are assured that its parent is not full.

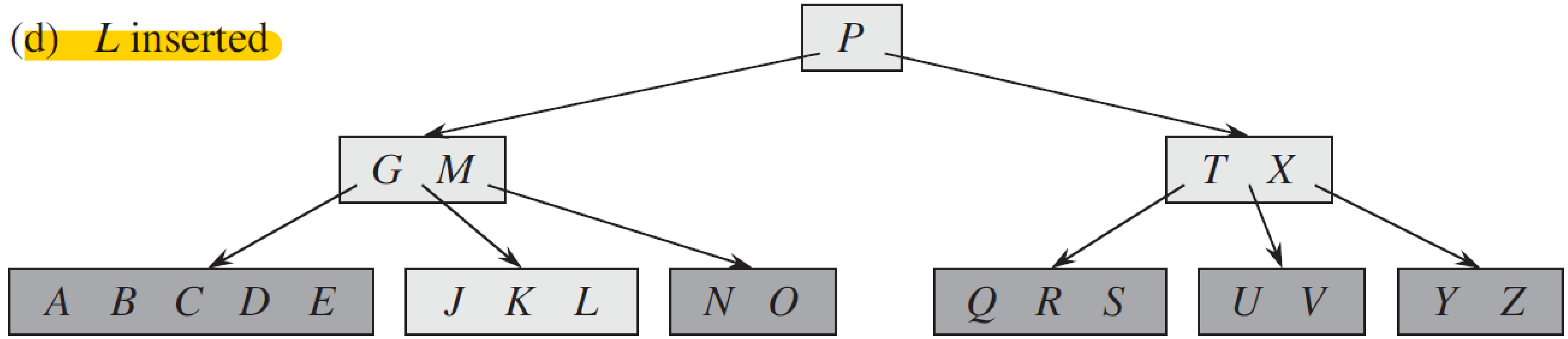
Inserting a key into a B-tree

The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys.



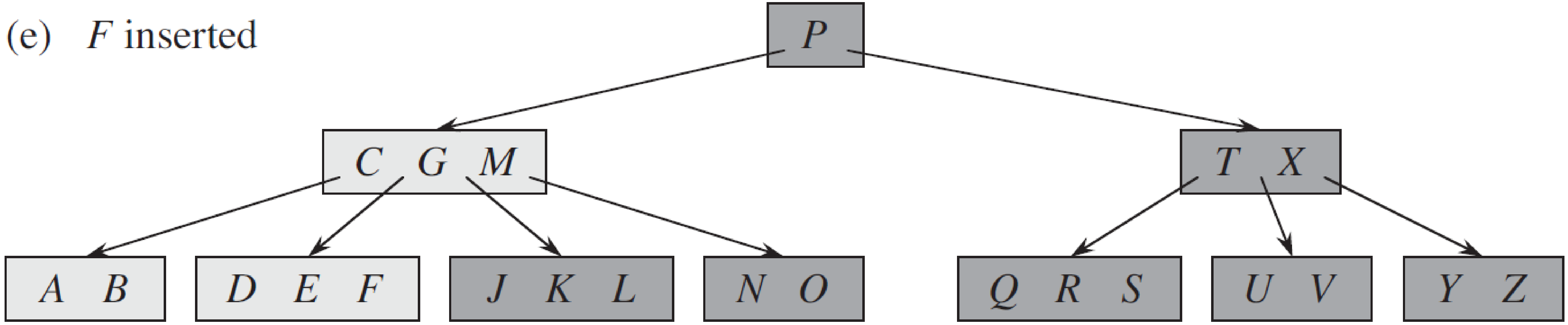
Inserting a key into a B-tree





Inserting a key into a B-tree

The minimum degree for this B-tree is t = 3, so a node (other than the root) cannot have fewer than 2 keys



B-TREE-INSERT performs $O(h)$ disk accesses.