B-TREES

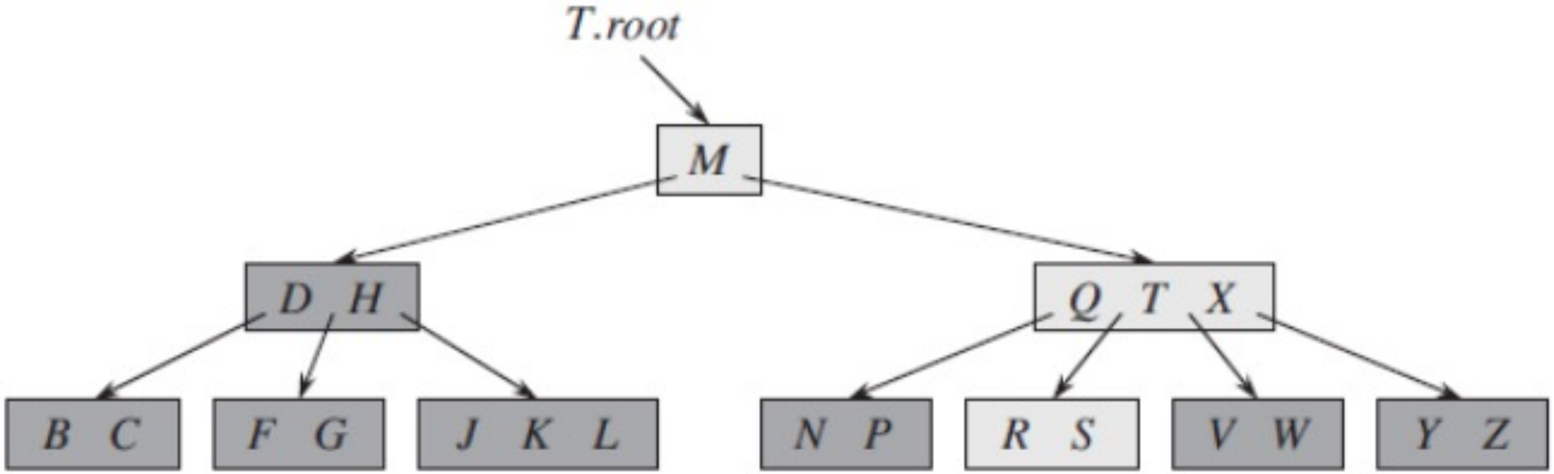
CS340

B-trees

- Balanced search trees designed to work well with old-fashioned spinning hard drives.
- Spinning hard drives are popular
  - They're cheaper than flash drives
  - They're a lot cheaper than putting everything in RAM
- Databases are huge and won't all fit in memory.
- Reading from a disk drive is many orders of magnitude slower than reading from RAM.
- In order to amortize the time spent waiting for mechanical movements, disks access not just one item but several at a time.
- Information is read from a disk one *page* at a time.
- A B-tree node is usually as large as a whole disk page. This gives maximum impact from a disk read operation.

B-Trees

- a node containing n keys has n+1 children
- all leaves are at the same depth in the tree
- Nodes may have many children, from a few to thousands.
- Every n-node B-tree has height $O(\log_t n)$. The branching factor is usually much larger than with binary trees.



B-trees

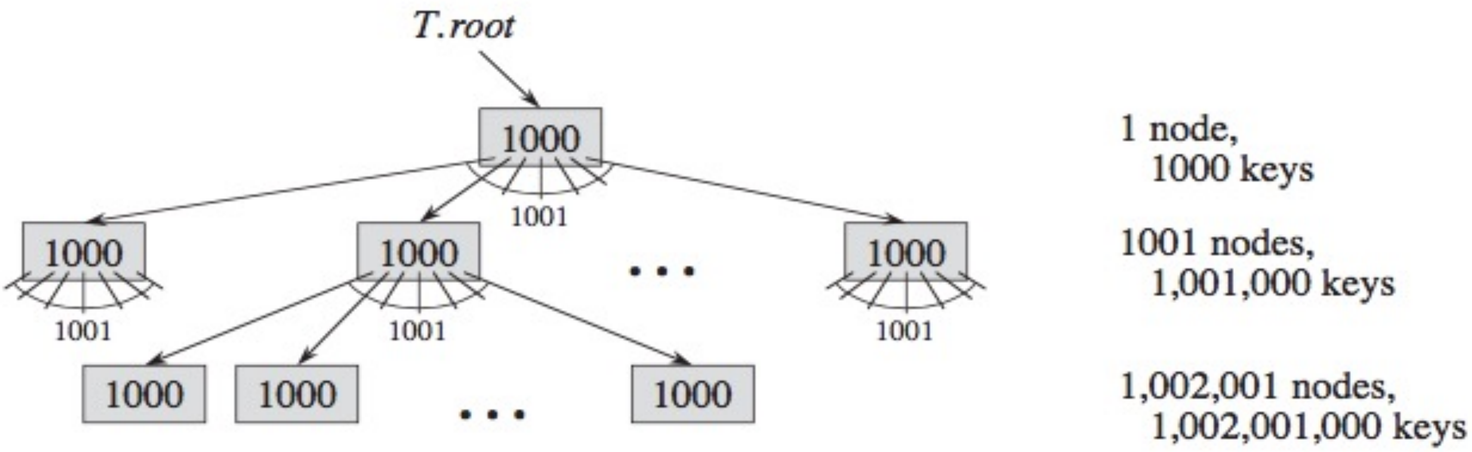


Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x is $x.n$, the number of keys in x. Each internal node and leaf contains 1000 keys. This B-tree has 1001 nodes at depth 1 and over one million leaves at depth 2.

B-tree attributes

- The keys of a node are in increasing order.
- If a node has n keys, then it has n+1 child pointers.
  - These pointers refer to **intervals** between keys.
- All leaves are at the same depth.
- t=minimum degree of a node. Every node other than the root must have at least t children (and t-1 keys). They must also have at most 2t children.
  - This bounds the minimum and maximum branching factors.

B-tree height

Theorem 18.1

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

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

- **Proof:** The root has at least 2 children, and every other node has at least t children. So, at level 1, there are at least 2 nodes. At level 2, there are at least 2t nodes. At level 3, there are at least $2t^2$ nodes, at level 4 $2t^3$ nodes, and so forth. Therefore:

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

The last line can be seen to simplify to the statement of the Theorem.

Basic Operations

- B-Tree-Search:
  - Due to the **ordering** property of the nodes, the only difference with BST search is due to having more than 2 children per node.
  - At a node x, we make x.n+1 branching decisions (where x.n is the number of keys).
  - Time complexity of B-Tree search = **$O(t \log_t n)$**

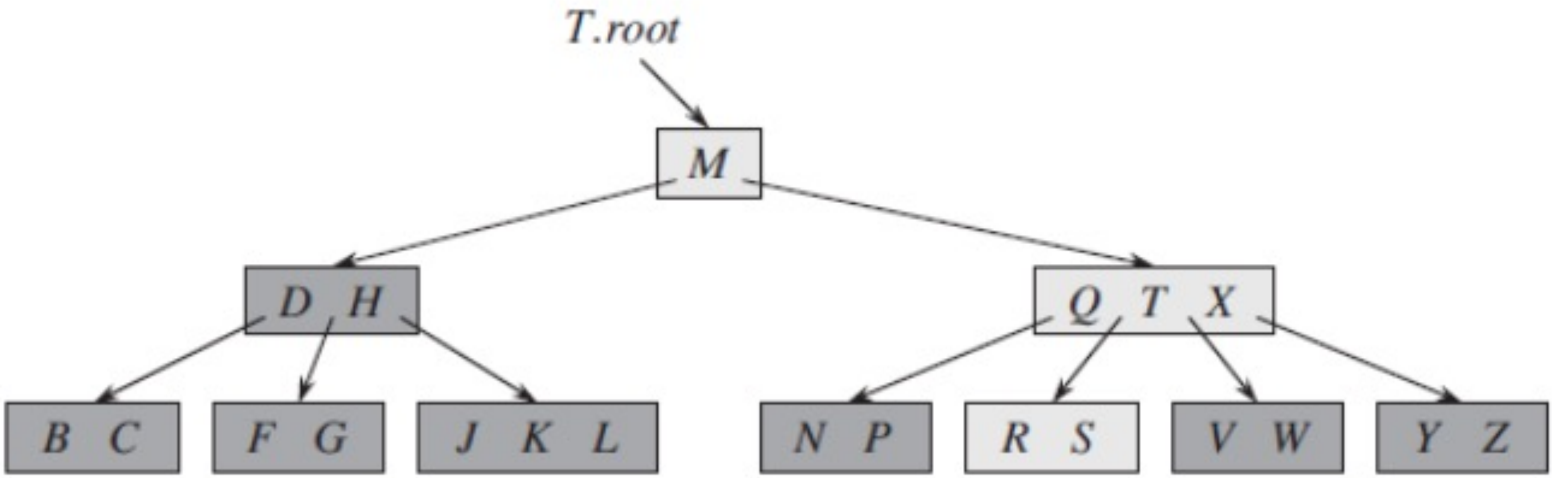
    height of tree = $O(\log_t n)$ where n = keys in b-tree

    where t=minimum degree of node (= how much time the algorithm spends in a node)

B-Tree-Search

```
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)
```

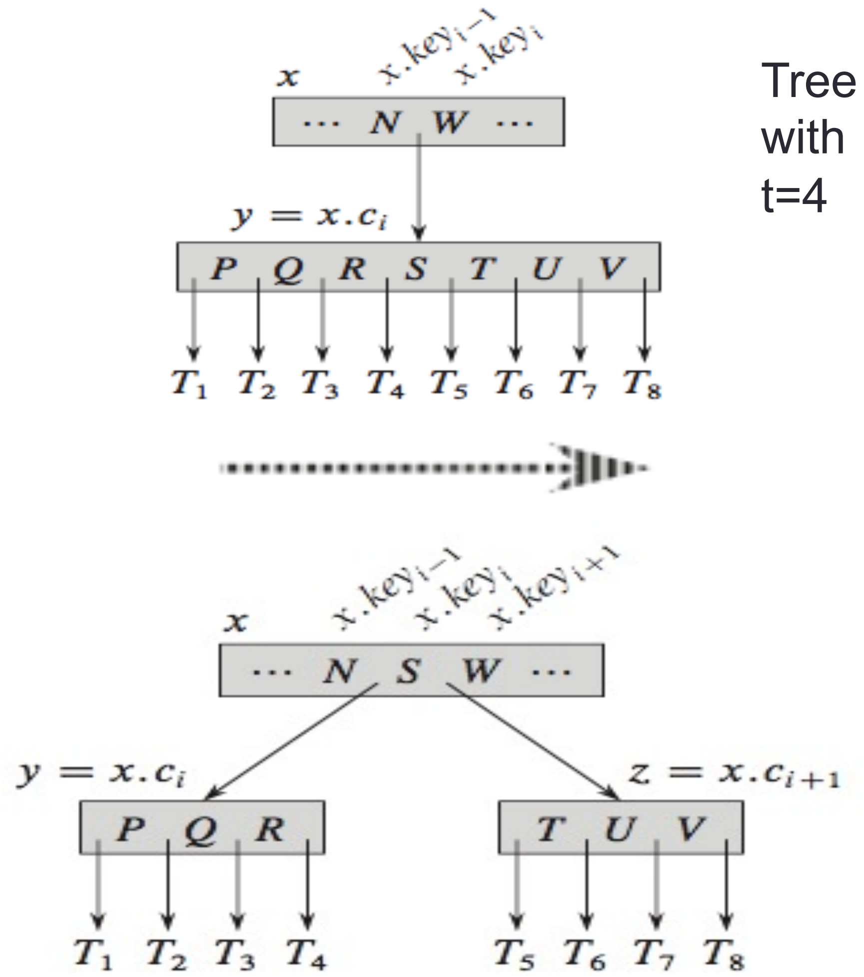
This is pretty straightforward.
But, can you think of how you might speed up the search *within* a node?



B-Tree Insertion

- Starts from root, proceeds downwards until correct location is found for insertion.
- Issue is maintaining B-Tree properties
  - Cannot insert into a “full” node with 2t children
- Therefore, might “Split” nodes as we move down.
  - The child of a node is checked to see if it is full and needs splitting.

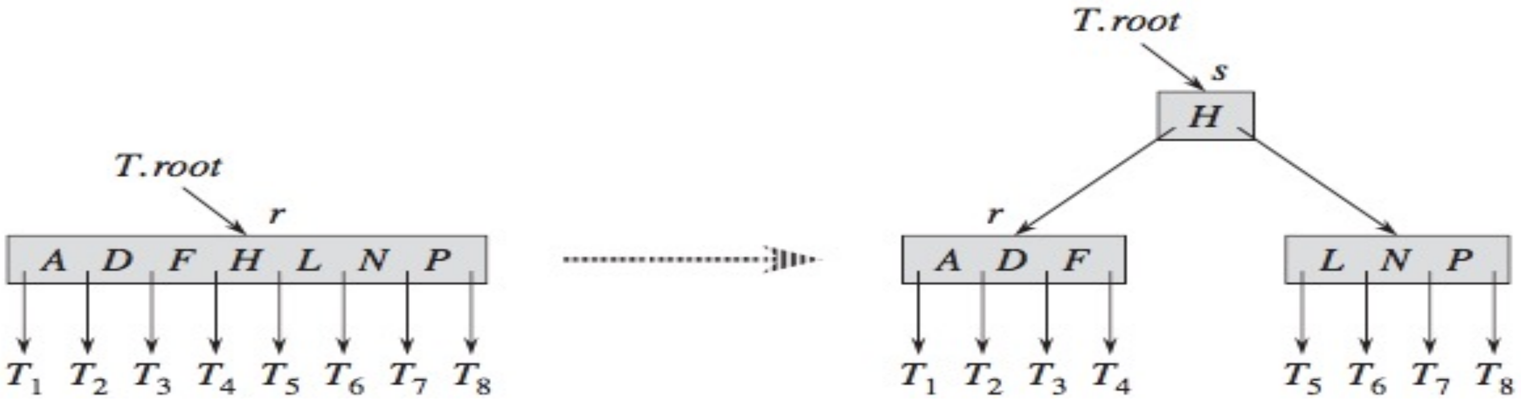
```
B-TREE-SPLIT-CHILD(x, i)
z = ALLOCATE-NODE()
y = x.c_i
z.leaf = y.leaf
z.n = t - 1
for j = 1 to t - 1
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t
        z.c_j = y.c_{j+t}
y.n = t - 1
for j = x.n + 1 downto i + 1
    x.c_{j+1} = x.c_j
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```



Tree with t=4

In case of a full root

- Remember that the root is the only node that is allowed to have only two children as minimum.



```
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.c_1 = r
8       B-TREE-SPLIT-CHILD(s, 1)
9       B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

Lines 2-9 handle the case of the full root.

B-Tree-Insert-Nonfull is the recursive procedure that continues downward, splitting children as needed.

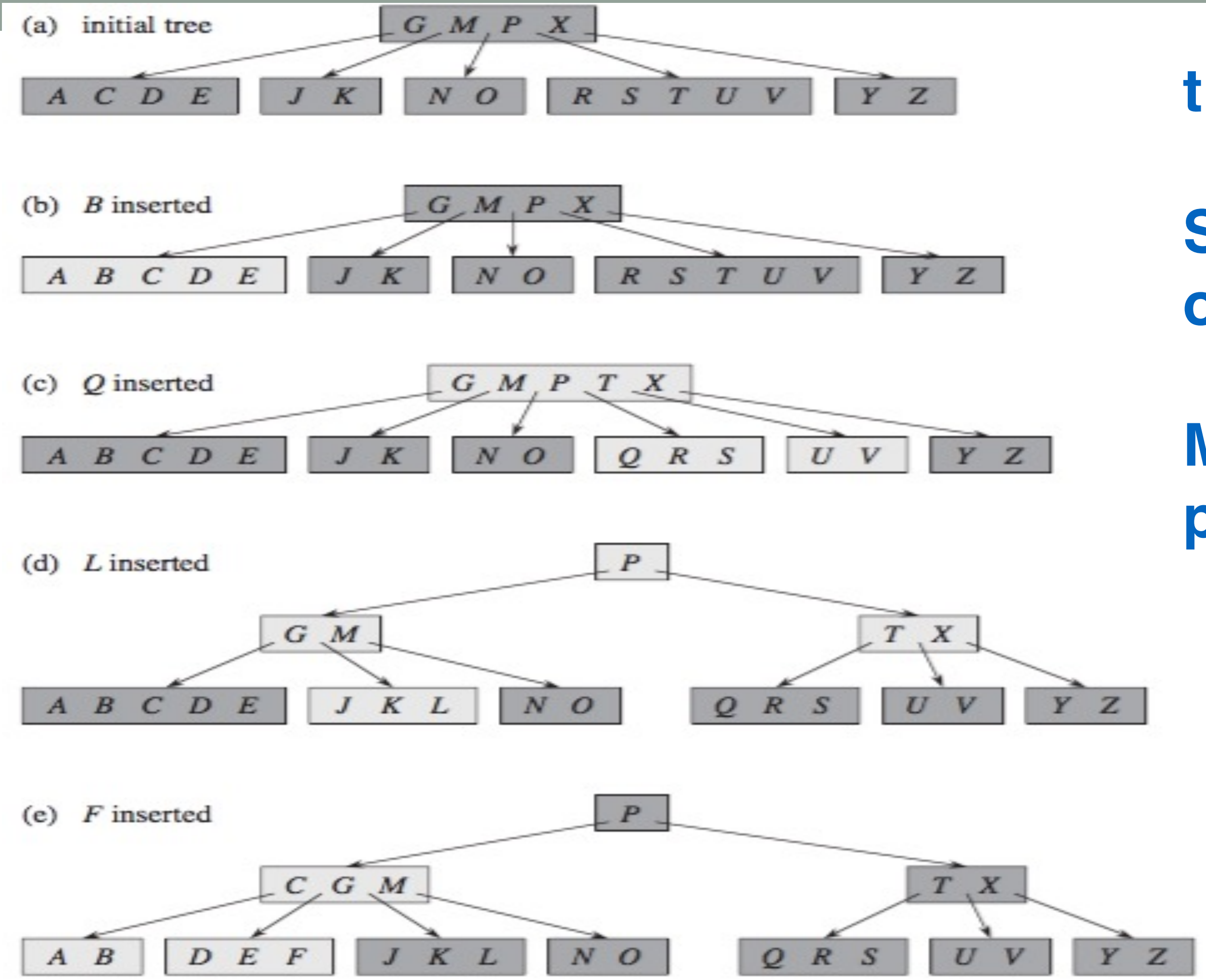
```
B-TREE-INSERT-NONFULL(x, k)
1  i = x.n
2  if x.leaf
3      while i ≥ 1 and k < x.key_i
4          x.key_{i+1} = x.key_i
5          i = i − 1
6      x.key_{i+1} = k
7      x.n = x.n + 1
8      DISK-WRITE(x)
9  else while i ≥ 1 and k < x.key_i
10         i = i − 1
11     i = i + 1
12     DISK-READ(x.c_i)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

The leaf case is straightforward as it has no children needing splitting, and the recursion ends too.

Lines 9-17 are the recursive case.

Line 13 checks if the child is full and splits accordingly.

Line 17 continues recursing downward in any case.



t is 3 in this example.

So, max number of child pointers is 6.

Max number of keys per node is 5.

In what case is the root split?

In what case is a leaf split?

B-Tree

- We are using a B-Tree to index a large amount of data stored on a hard drive. The hard drive is read in blocks of 4096 bits. The keys are 32-bit integers, and pointers to child nodes are 64-bits.

- What is the maximum number of keys a node can hold?