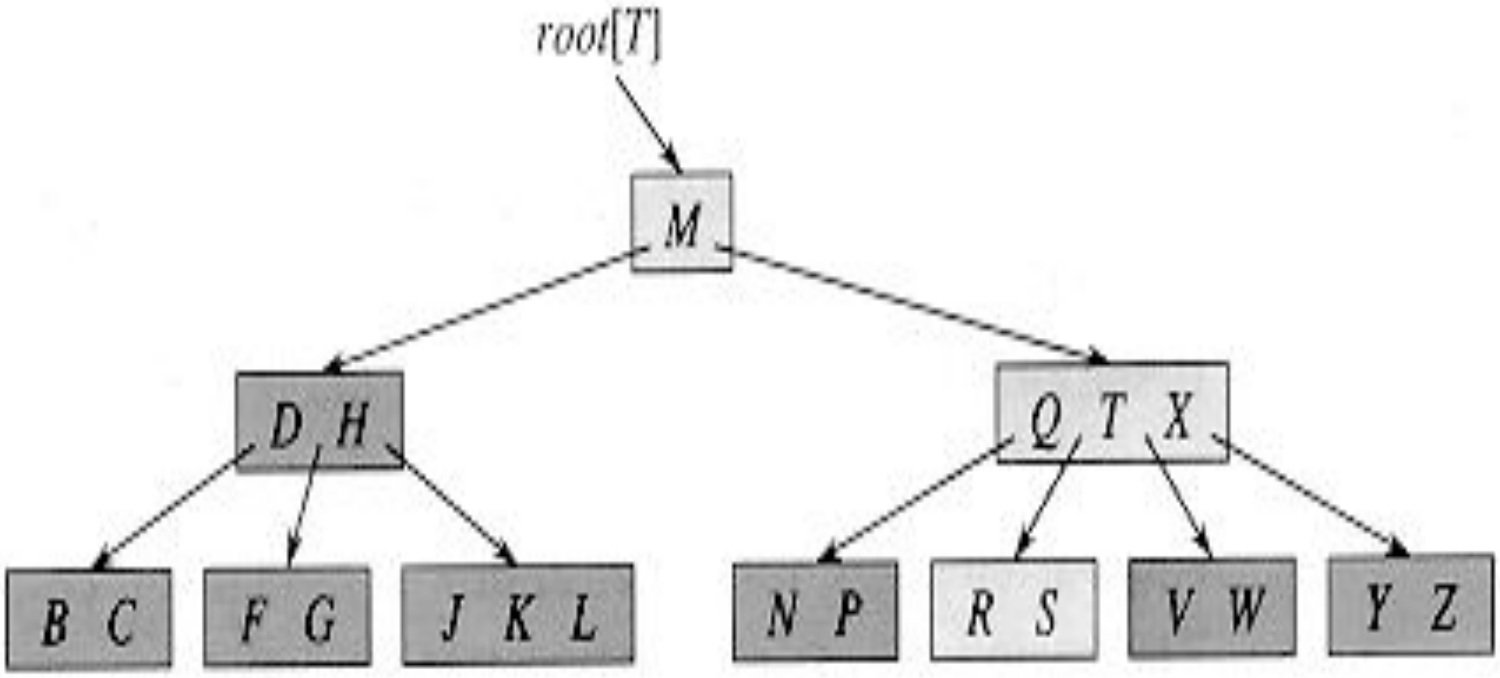
B Tree

Dr. Anant Ram
Associate Professor
Dept. of CEA, GLA University, Mathura

B Tree

B Tree

- B-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices.



Properties of B Tree

A ***B-tree*** *T* is a rooted tree (with root *root*[*T*]) having the following properties.

1. Every node *x* has the following fields:

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

   b. the $n[x]$ keys themselves, stored in nondecreasing order: $key_1[x] <= key_2[x] <=$ ……….$<= key_n[x]$

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

2. If *x* is an internal node, it also contains $n[x] + 1$ pointers c_1 $[x]$, $c_2[x], \ldots, c_n[x]+1[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

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

$k_1 <= key_1[x] <= k_2 <= key_2[x] <=$ --------$<=$ $\text{key}_{n[x]}[x] <= k_n[x]+1$.

Properties of B Tree

4. Every leaf has the same depth, which is the tree's height h.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer t >= 2 called the **minimum degree** of the B-tree:

   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 can contain at most 2t - 1 keys. Therefore, an internal node can have at most 2t children. We say that a node is **full** if it contains exactly 2t - 1 keys.

.

B Tree

B Tree

B Tree Search

```
B-TREE-SEARCH(x, k)
i = 1
while i <=n[x] and k< key_i[x]
    do i= i + 1
if i<= n[x] and k = key_i[x]
then return (x, i)
if leaf [x]
then return NIL
else DISK-READ(c_i[x])
return B-TREE-SEARCH(c_i[x], k)
```

Creating an empty B-tree

B-TREE-CREATE(T)

1 x = ALLOCATE-NODE()

2 *leaf*[x] = TRUE

3 $n[x] = 0$

4 DISK-WRITE(x)

5 *root*[T] = x

Splitting a node in a B-tree

B-TREE-SPLIT-CHILD(x,i,y)

z = ALLOCATE-NODE()

l*eaf*[*z*] = leaf*[*y*]*

$n[z]$ = t - *1*

for j = 1 **to** t - 1

  do $key_j[z] = key_{j+t}[y]$

if not *leaf* [y]

  then for j = 1 **to** t

  do $c_j[z] = c_{j+t}[y]$

$n[y]$ = t - *1*

for j = n*[*x*]* + *1* ***downto*** i + *1*

  do $c_{j+1}[x] = c_j[x]$

Splitting a node in a B-tree

$c_{i+1}[x] = z$

for $j = n[x]$ **downto** i

do $key_{j+1}[x] = key_j[x]$

$key_i[x] = key_t[y]$

$n[x] = n[x] + 1$

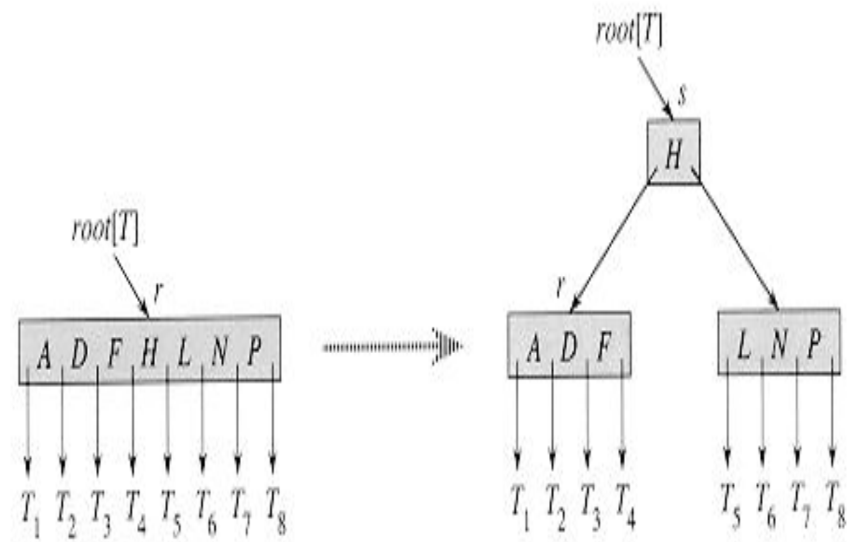
DISK-WRITE(y)

DISK-WRITE(z)

DISK-WRITE(x)

Inserting a key into a B-tree

```
B-TREE-INSERT(T,k)
r = root[T]
if n[r] = 2t - 1
    then s = ALLOCATE-NODE()
    root[T] = s
    leaf[s] = FALSE
    n[s] = 0
    c_1[s] = r
    B-TREE-SPLIT-CHILD(s,1,r)
    B-TREE-INSERT-NONFULL(s,k)
else B-TREE-INSERT-NONFULL(r,k)
```



Inserting a key into a B-tree

```
B-TREE-INSERT-NONFULL(x,k)
i = n[x]
if leaf[x]
   then while i =1 and k < key_i[x]
              do key_{i+1} [x] = key_i[x]
                 i = i - 1
   key_{i+1}[x] = k
   n[x] = n[x] + 1
   DISK-WRITE(x)
else while i >= 1 and k < key_i[x]
              do i = i - 1
      i = i + 1
      DISK-READ(c_i[x])
     if n[c_i[x]] = 2t - 1
          then B-TREE-SPLIT-CHILD(x,i,c_i[x])
          if k > key_i[x]
              then i = i + 1
      B-TREE-INSERT-NONFULL(c_i[x],k)
```

Deleting a key from a B-tree

1. If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

b. Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

c. Otherwise, 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.

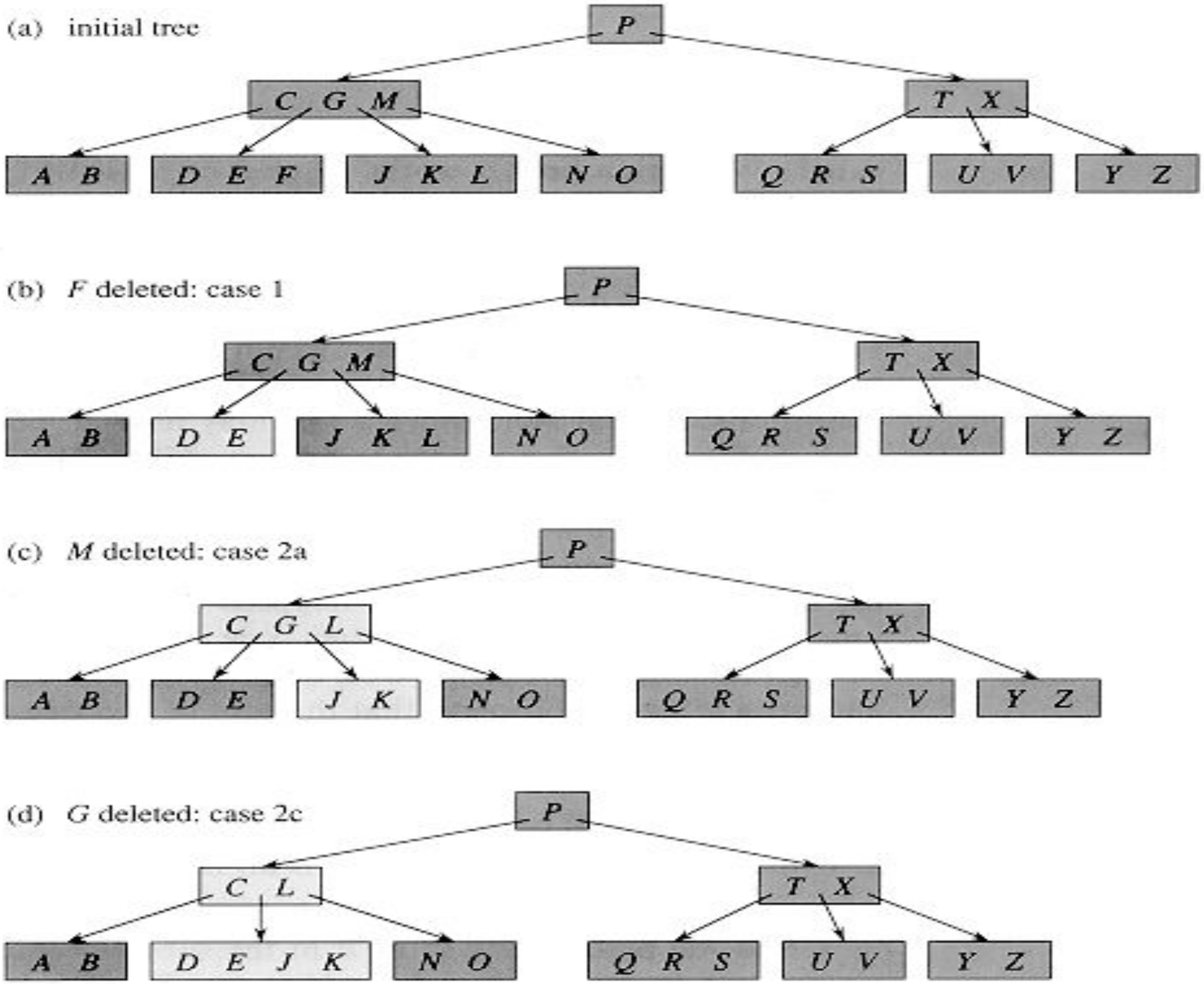
Deleting a key from a B-tree

3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate sub tree that must contain k, if k is in the tree at all. If $c_i[x]$ 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 recursing on the appropriate child of x.

a. If $c_i[x]$ has only t - 1 keys but has a sibling with t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child from the sibling into $c_i[x]$.

b. If $c_i[x]$ and all of $c_i[x]$'s siblings have t - 1 keys, merge 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.

B Tree

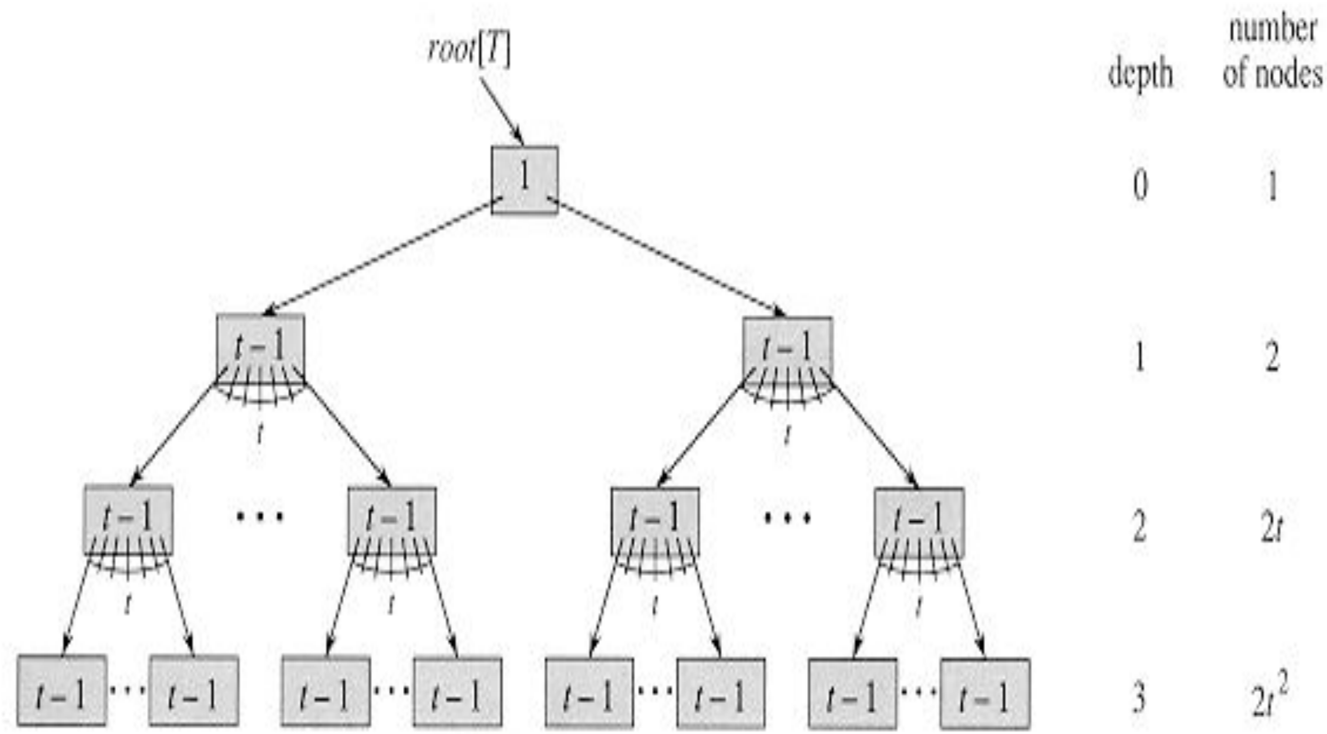


(a) initial tree
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z
(b) F deleted: case 1
P
C G M
T X
A B
D E
J K L
N O
Q R S
U V
Y Z
(c) M deleted: case 2a
P
C G L
T X
A B
D E
J K
N O
Q R S
U V
Y Z
(d) G deleted: case 2c
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z

B Tree

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

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



B-tree

Proof If a B-tree has height h, the number of its nodes is minimized when the root contains one key and all other nodes contain t - 1 keys. In this case, there are 2 nodes at depth 1, $2t$ nodes at depth 2, $2t^2$ nodes at depth 3, and so on, until at depth h there are $2t^{h-1}$ nodes.

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

B Tree