B-Tree

الدكتور
اثير العاني

Multiway search tree

- A tree was defined as either an empty structure or a structure whose children are disjoint tree $t_1, t_2, \ldots t_k$. Each node of this kind of tree can have more than two children. This tree called a multiway tree of order m, or an m-way tree.
- A Multiway search tree of order m, or an m-way search tree, is a Multiway search tree in which
  - Each node has m children and m-1 keys
  - The keys in each node are in ascending order
  - The keys in the first ***i*** children are smaller the ***ith*** key
  - The keys in the ***last m-i*** children are larger than the ***ith*** key
- M-way search tree → m-way tree
- Binary search tree → binary tree

Introduction of B-tree

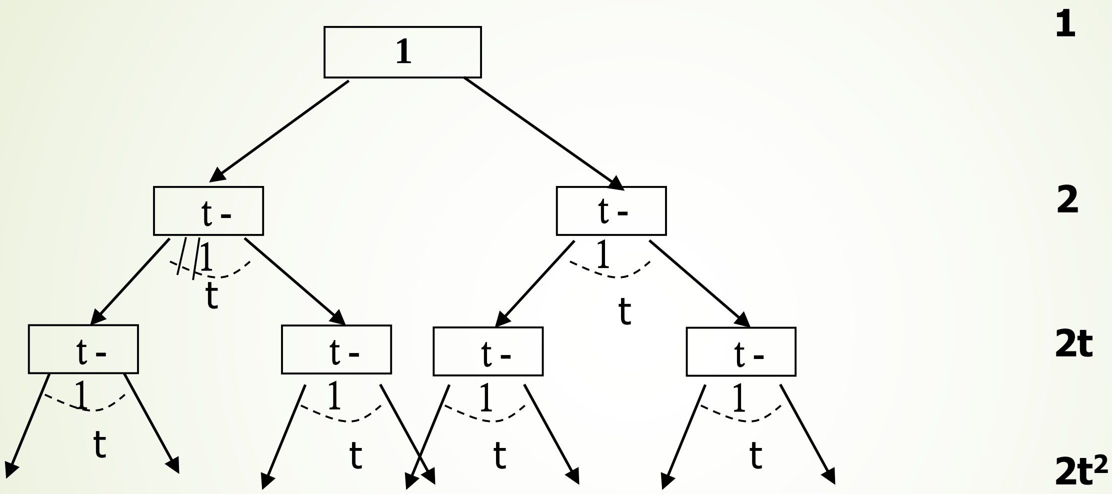
- B-tree: proposed by Bayer and McCreight 1972
- A B-tree operates closely with secondary storage and can be tuned to reduce the impediments imposed by this storage
- One important property of B-trees is the size of each node which can be made as large as the size of the block. (the basic unit of I/O operations associated with a disk is a block)
- a B-tree of order *t* is a *multiway search* tree.

- Thm :

  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 :



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

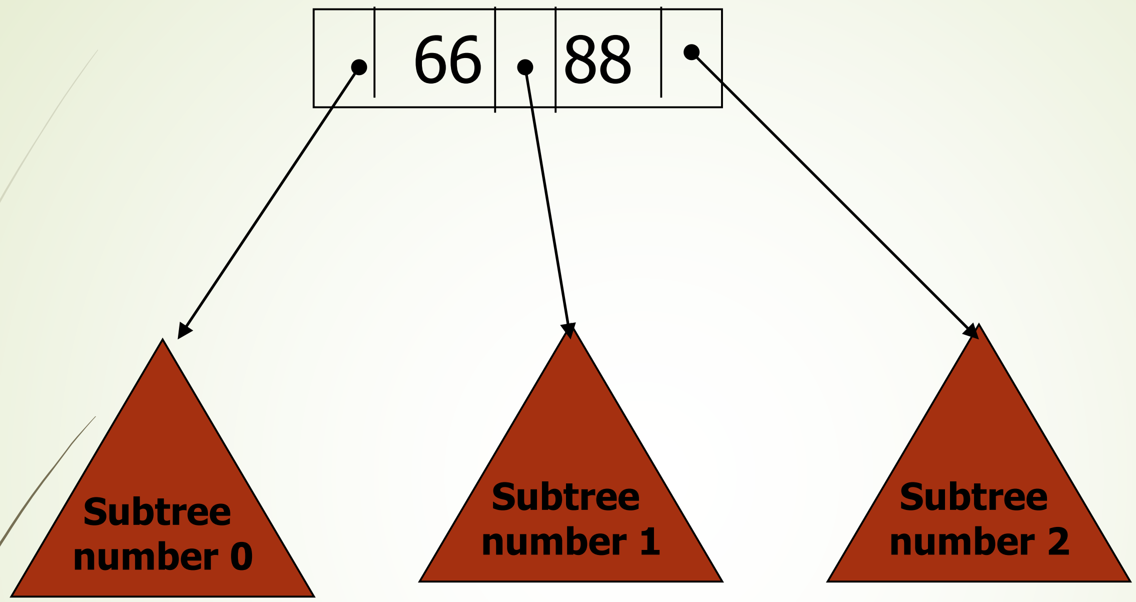
$$= 1 + 2(t+1)\cdot\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1.$$

$$\frac{n+1}{2} \geq t^h. \qquad \log_t \frac{n+1}{2} \geq h.$$

- A B-tree is not a binary tree because B-tree has many more than two children
- B-trees may be formulated to store a *set* of elements or a *bag* of elements. (a given elements can occur many times in a bag but only once in a set)
- A B-tree is *balanced.*
- Every leaf in a B-tree has the *same depth*
- 2-3-4 tree (discussed by Rudolf Bayer): a B-tree of *order 4 (min degree=2)*

The elements in a B-tree node

- Rule 1:the root may have as few as one elements (or even no elements if it also has no children); every other node has at least *minimum* elements
- Rule 2: the *maximum* number of elements in a node is *twice* the value of *minimum*
- Rule 3: the elements of each B-tree node are stored in a partially filled array, sorted from the smallest elements (at index 0) to the largest elements (at the final used position of the array)
- Rule 4: the number of subtrees <u>below</u> a nonleaf node is always *one more than* the number of the elements in the node.
- Rule 5: for any leaf node: (1) an element at index *i* is *greater* than all the elements in subtree number *i* of the node, and (b) an element at index *i* is *less* than all the elements in subtree number *i+1* of the node.



Each element in subtree number 0 is less than 66

Each element in subtree number 1 is between 66 and 88.

Each element in subtree number 2 is greater than 88

- convention :
  - Root of the B-tree is always in main memory.
  - Any nodes that are passed as parameters must already have had a DISK_READ operation performed on them.
- Operations :
  - Searching a B-Tree.
  - Creating an empty B-tree.
  - Splitting a node in a B-tree.
  - Inserting a key into a B-tree.
  - Deleting a key from a B-tree.

- B-Tree-Search(x,k) :
  - Algorithm :

B-Tree-Search(x,k)

$\{ \quad i \leftarrow \mathbf{1}$

$\quad \text{while } i \le n[x] \text{ and } k > key_i[x]$

$\quad\quad \text{do } \quad i \leftarrow i + \mathbf{1}$

$\quad if \quad i \le n[x] \text{ and } k = key_i[x]$

$\quad\quad \text{then return}(x, i)$

$\quad if \quad leaf[x] \text{ then return NULL}$

$\quad\quad\quad \text{else DISK-READ}(C_i[x])$

$\quad\quad\quad\quad \text{return B-Tree-Search}(C_i[x], k)$

$\}$

- Total CPU time :

$$O(th) = O(t \log_t n).$$

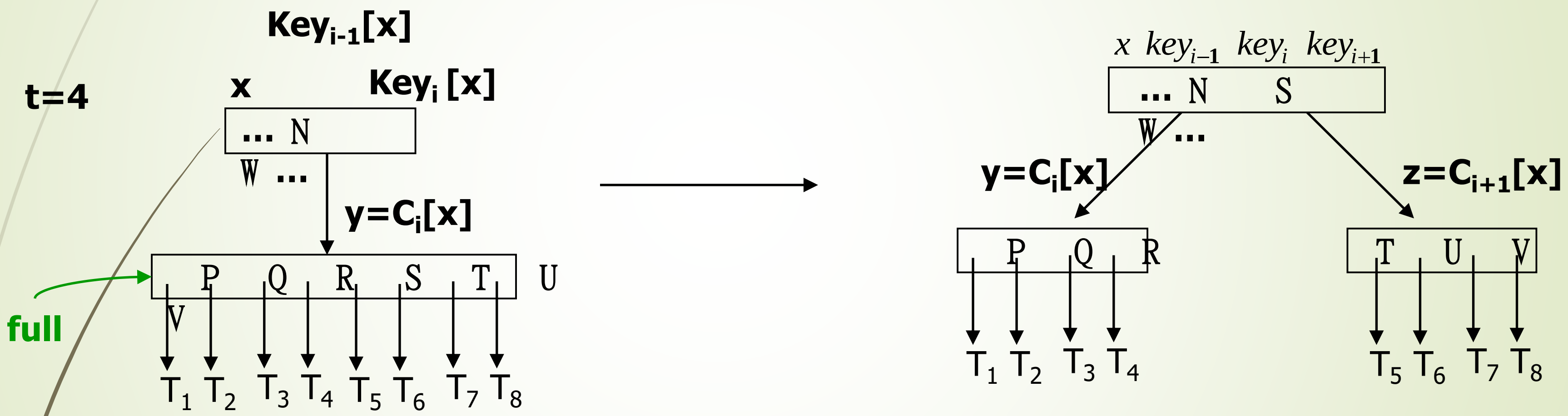
- B-Tree-Created(T) :
  - Algorithm :

B-Tree-Create(T)

$$
\begin{aligned}
\{ \quad & x \leftarrow \mathrm{Allocate-Node}() \\
& \mathrm{Leaf}[x] \leftarrow \mathrm{TRUE} \\
& \mathrm{n[x]} \leftarrow 0 \\
& \mathrm{DISK\text{-}WRITE(x)} \\
& \mathrm{root[T]} \leftarrow \mathrm{x} \\
\} &
\end{aligned}
$$

  - time : $O(\mathbf{1})$

- B-Tree-Split-Child(x,i,y) :
  - Splitting a node in a B-Tree :



Splitting a full node y (have 2t-1 keys) around its median key key$_t$[y] into 2 nodes having (t-1) keys each.

- Algorithm :

B-Tree-Split-Child(x,I,y)

{ $z \leftarrow \text{Allocate - Node}()$

$\text{leaf}[z] \leftarrow \text{leaf}[y]$

$n[z] \leftarrow t - \mathbf{1}$

$\textit{for}\ \ j \leftarrow \mathbf{1}\ \ \textit{to}\ \ \text{t - 1}\ \ \textit{do}\ \ \text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$

$\textit{if}\ \text{not}\ \text{leaf}[y]\ \textit{then}$

$\quad \textit{for}\ \ j \leftarrow \mathbf{1}\ \ \textit{to}\ \ \text{t}\ \ \textit{do}\ \ C_j[z] \leftarrow C_{j+t}[y]$

$n[y] \leftarrow t - \mathbf{1}$

$\textit{for}\ \ j \leftarrow n[x] + \mathbf{1}\ \ \textit{downto}\ \ i+1\ \ \textit{do}\ \ C_{j+1}[x] \leftarrow C_j[x]$

$C_{j+1}[x] \leftarrow z$

$\textit{for}\ \ j \leftarrow n[x]\ \ \textit{downto}\ \ i\ \ \textit{do}\ \ \text{Key}_{j+1}[x] \leftarrow \textit{Key}_j[x]$

$\text{Key}_i[x] \leftarrow \text{Key}_t[y]$

$n[x] \leftarrow n[x] + \mathbf{1}$

$\text{DISK - WRITE}(y)$

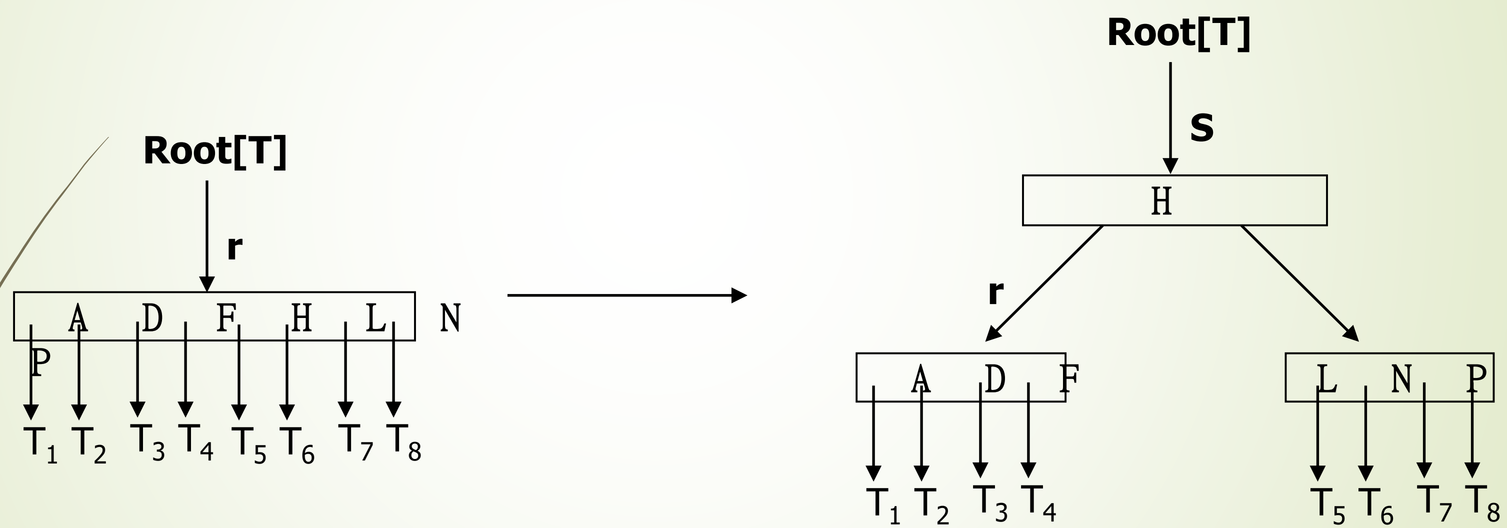
$\text{DISK - WRITE}(z)$

$\text{DISK - WRITE}(x)$

}

- B-Tree-Insert(T,k) :
  - Insert a key in a B-Tree :

t=4



- Algorithm :

B-Tree-Insert(T,k)

{ $r \leftarrow root[T]$

$\textit{if}\ n[r] = 2t - 1\ \textit{then}$

{ $S \leftarrow Allocate - Node()$

$root[T] \leftarrow S$

$leaf[S] \leftarrow FALSE$

$n[S] \leftarrow \mathbf{0}$

$C_i[S] \leftarrow r$

B-Tree-Split-Child(S,l,r)

B-Tree-Insert-Nonfull(S,k)

}

$\textit{else}\ B - Tree - Insert - Nonfull(r,k)$

}

- B-Tree-Insert-Nonfull(x,k) :
  - Algorithm :

B-Tree-Insert-Nonfull(x,k)

{ $\quad \mathrm{i} \leftarrow \mathrm{n}[x]$

$\quad$ *if* leaf[x] *then*

$\quad$ { *while* $\mathrm{i} \geq 1$ and $\mathrm{k} < \mathrm{key}_{\mathrm{i}}[x]$

$\quad\quad$ *do* { $\mathrm{key}_{\mathrm{i+1}}[x] \leftarrow \mathrm{key}_{i}[x]$

$\quad\quad\quad \mathrm{i} \leftarrow i - \mathbf{1}$ }

$\quad\quad \mathrm{key}_{\mathrm{i+1}}[x] \leftarrow k$

$\quad\quad \mathrm{n}[x] \leftarrow \mathrm{n}[\mathrm{x}] + \mathbf{1}$

$\quad\quad$ DISK - WRITE(x) }

$\quad$ *else*

$\quad$ { *while* $\mathrm{i} \geq 1$ and $\mathrm{k} < \mathrm{key}_{\mathrm{i}}[x]$

$\quad\quad$ *do* $\mathrm{i} \leftarrow \mathrm{i} - 1$

$\quad\quad \mathrm{i} \leftarrow i + \mathbf{1}$

$\quad\quad$ DISK - READ(C_{i}[x])

$\quad\quad$ *if* $\mathrm{n}[\mathrm{C}_{\mathrm{i}}[\mathrm{x}]] = 2\mathrm{t} - 1$

$\quad\quad\quad$ *then* B - Tree - Split - Child(x,i,C_{i}[x])

$\quad\quad\quad\quad$ *if* $\mathrm{k} > \mathrm{key}_{\mathrm{i}}[\mathrm{x}]$ *then* $\mathrm{i} \leftarrow \mathrm{i} + 1$

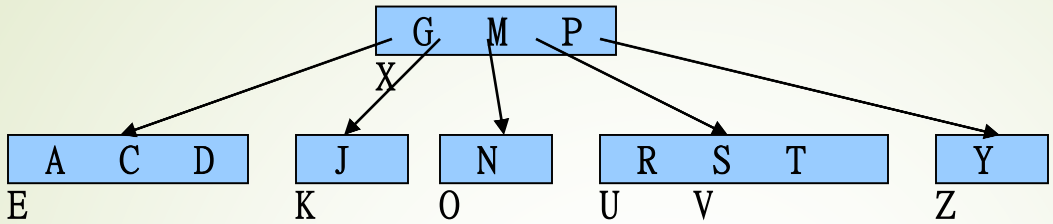
$\quad\quad$ B-Tree-Insert-Nonfull(C_{i}[x],k) }

}

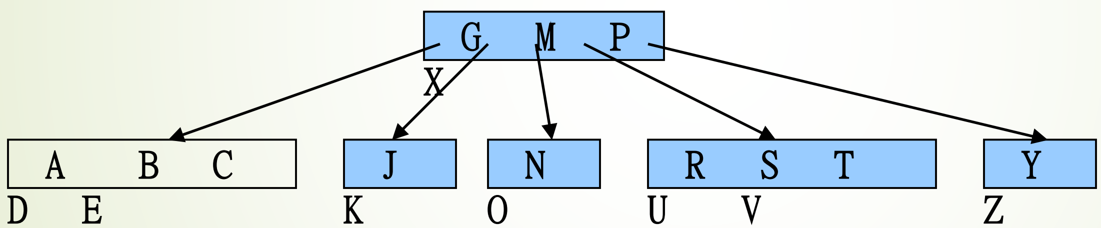
- Example : Inserting keys into a B-Tree.

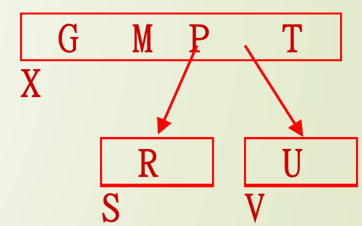
t=3

(a) Initial tree

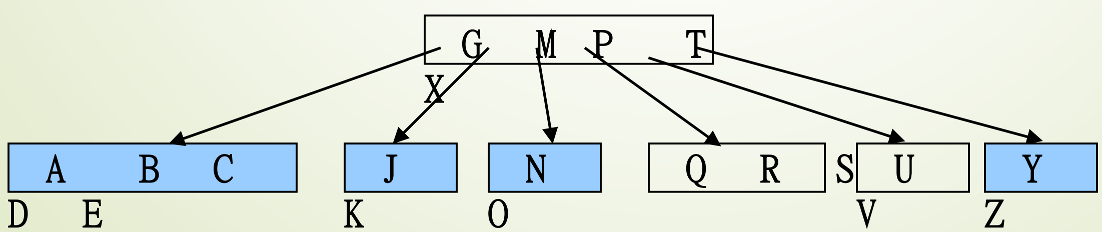


(b) ***B*** inserted

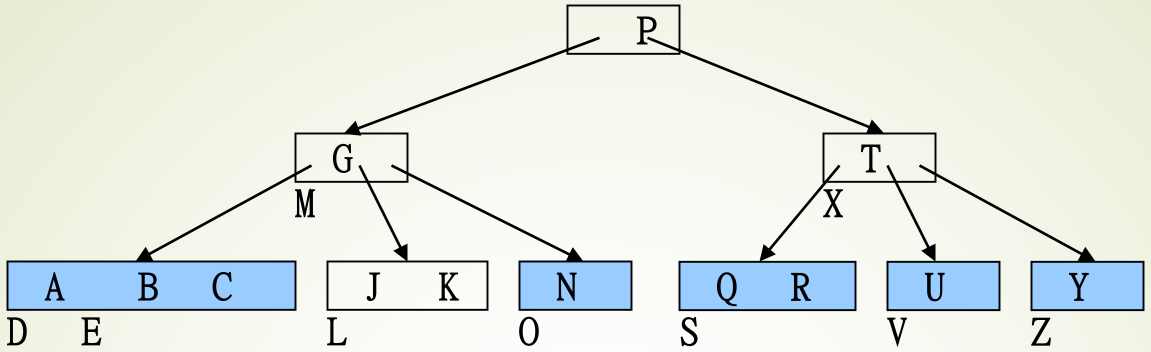




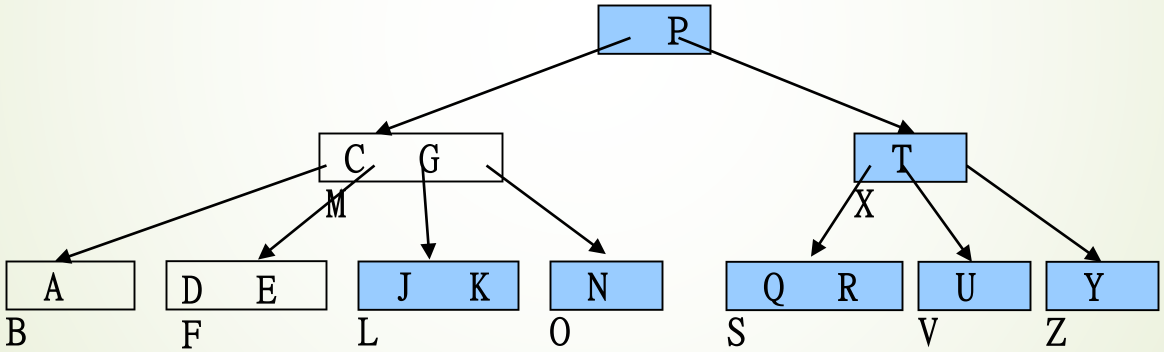
(c) ***Q*** inserted



(d) **L** insert



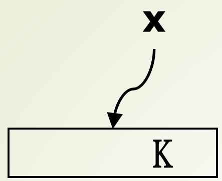
(e) **F** insert



- Deleting a key from a B-Tree :

(x has ≥ t keys)

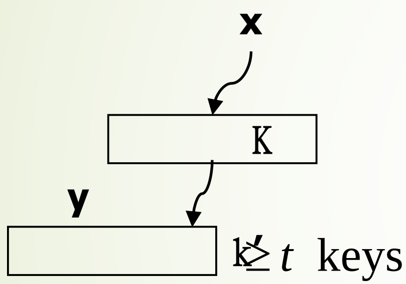
1. K is in x and x is a leaf :



delete k from x.

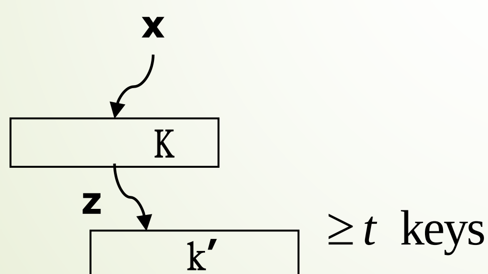
2. K is in x and x is an internal node :

a.



Recursively delete k′ and replace k by k′ in x.

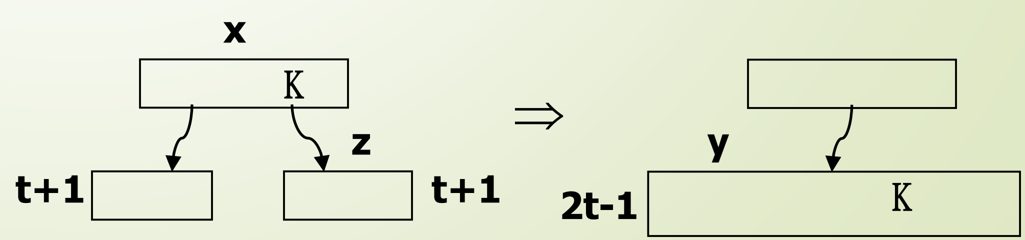
b.



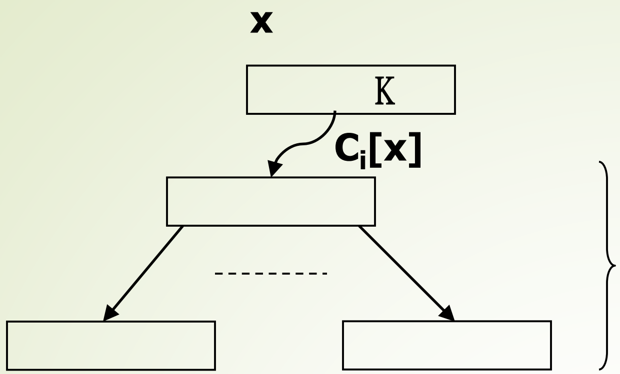
c. if both y,z has ≤ t-1 keys.

Merge y,z and k into y.

Recursively delete k from y.

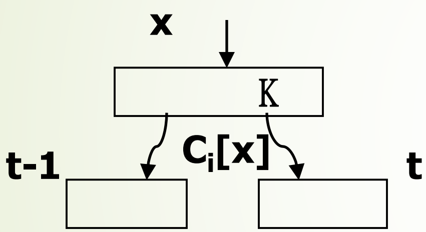


3. If K is not in internal node x :



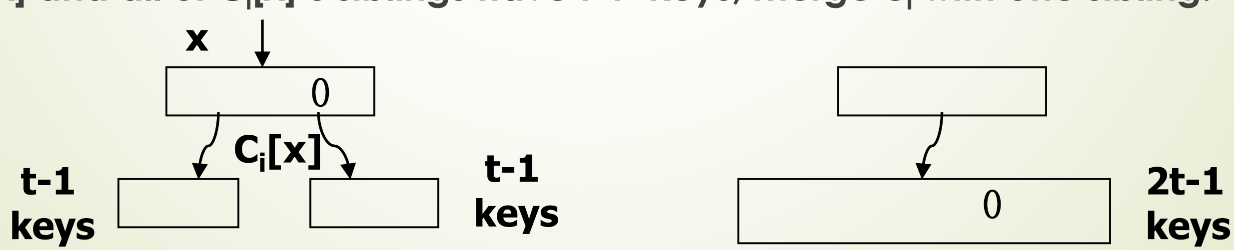
k is in this subtree.

a. If $C_i[x]$ has only t-1 keys but has a sibling with t keys



- **Move a key from x down to $C_i[x]$.**
- **Move a key from $C_i[x]$'s sibling to x.**
- **Move an appropiate child to $C_i[x]$ from its sibling.**

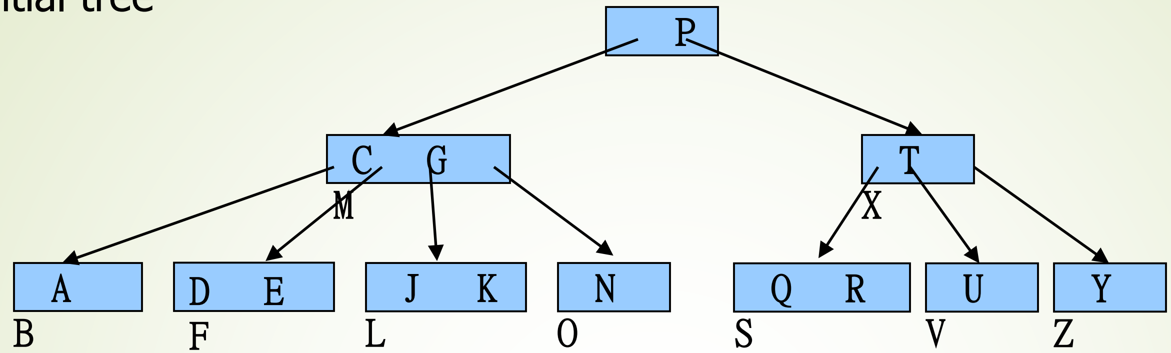
b. If $C_i[x]$ and all of $C_i[x]$'s siblings have t-1 keys, merge c_i with one sibling.



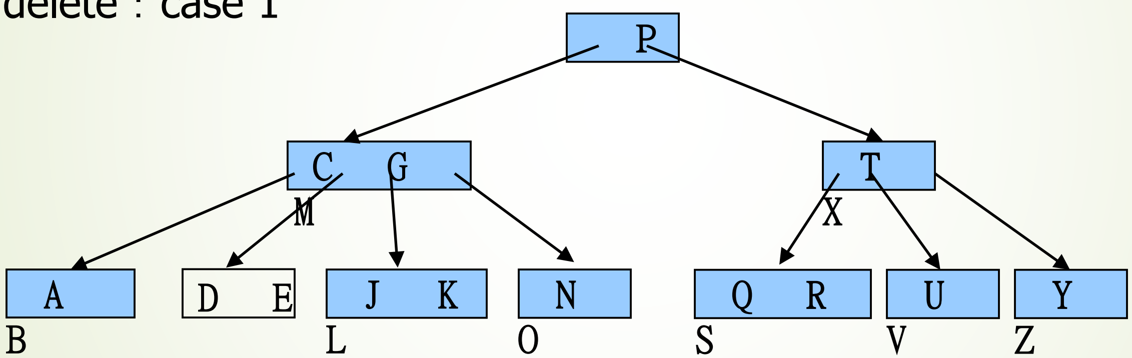
Example：Deleting a key from a B-Tree.

t=3

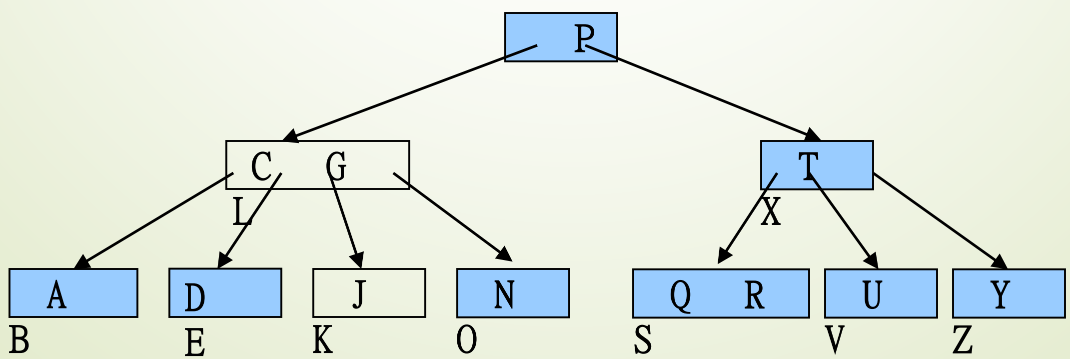
(a) Initial tree



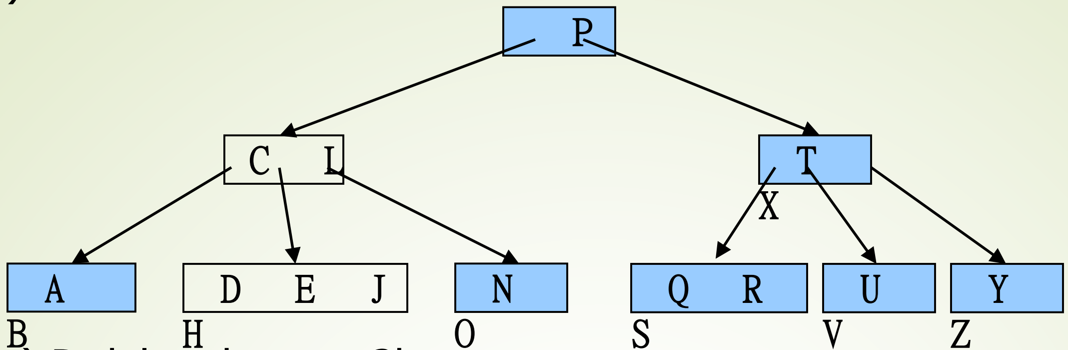
(b) F delete：case 1



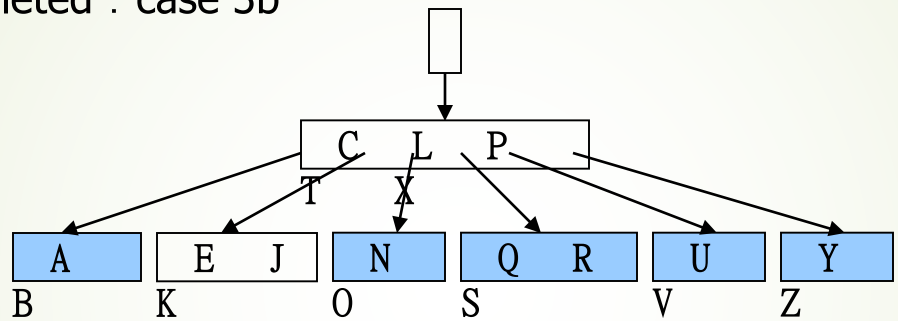
(c) M delete：case 2a



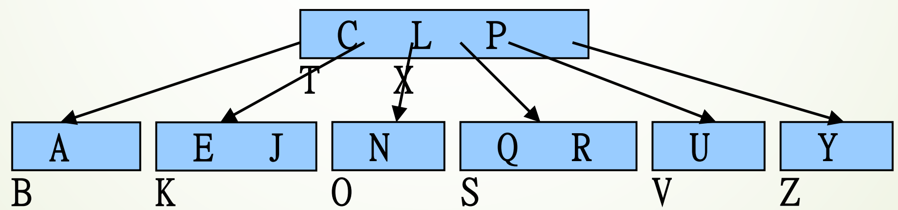
(d) G deleted : case 2c



(e) D deleted : case 3b



(e′) tree shrinks in height



(f) B delete : case 3a