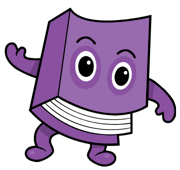
COT 6405
ANLYSIS OF ALGORITHMS

B-Trees

Computer & Electrical Engineering and Computer Science Dept.
Florida Atlantic University

Spring 2017

Advanced Data Structures – B-Trees

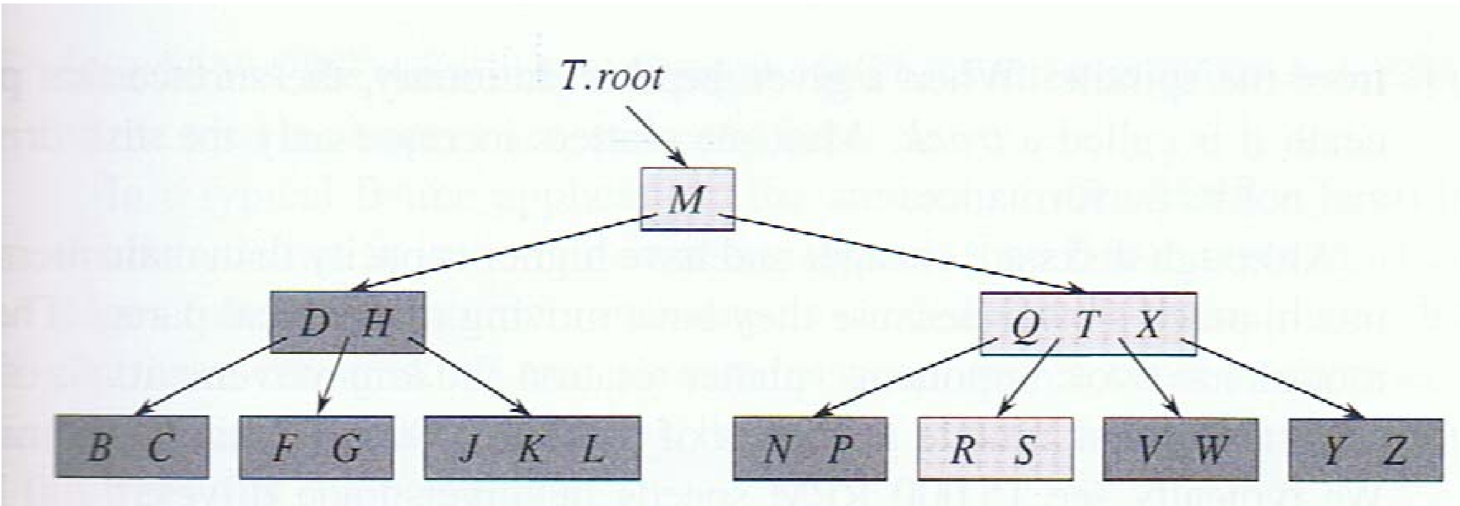


- Reading: CLRS chapter 18

B-trees

- Balanced search trees
- Works well on disks and other direct-access secondary storage devices
- Many database systems use B-trees, or variant of B-trees, to store information
- Efficient in minimizing disk I/O operations
- B-tree nodes may have from a few to thousands children
- B-trees have height O(lg *n*)

B-tree example

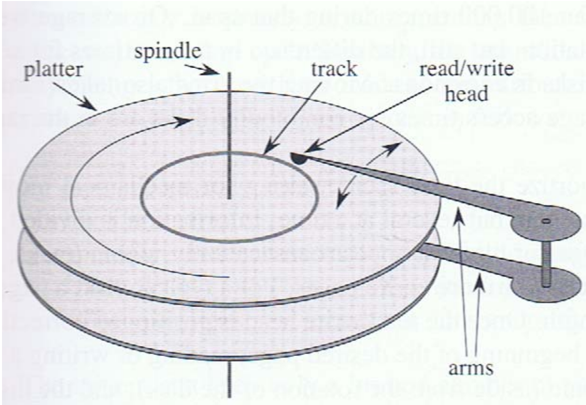


- keys are the consonants of English
- an internal node with x.n keys has x.n + 1 children
- all leaves have the same depth
- lightly shaded nodes are examined in search for the letter R

Primary/Secondary storage

- The **primary memory** (main memory) consists of silicon memory chips
- **Secondary storage**: magnetic storage technology such as tapes or disks
- Disks are *cheaper* and have *higher capacity* than the main memory
- Disks are much *slower* than the main memory because they have moving mechanical parts

Primary/Secondary storage



a typical disk drive

- average access time for commodity disks is ~ 8 – 11 ms
- access time for silicon memory is ~ 50 ns

$\Rightarrow$ access time for disks is over 5 order of magnitude slower !

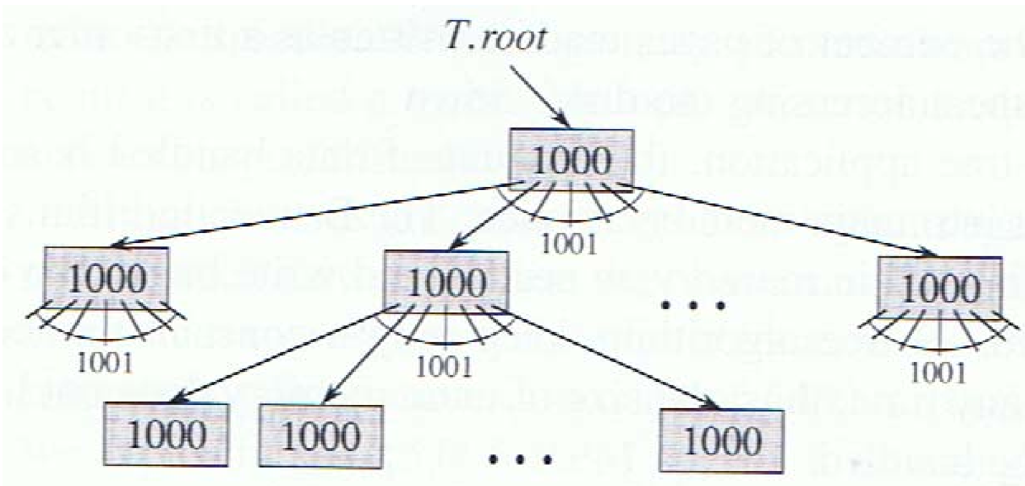
- information divided in **pages** (2^{11} – 2^{14} bytes)
- each disk reads/writes one or more pages

B-trees

- in the typical applications, the whole B-tree does not fit in the main memory
- copy pages from disk → main memory, then write back onto the disk the pages that have changed
- usually, B-tree algorithms keep only a constant number of pages in the main memory

```
x = a pointer to some object
DISK-READ(x)   // read object x in MM; no-op if
               //       x already in the MM
operations that access/modify attributes of x
DISK-WRITE(x)  // omitted if no attributes of x changed
other ops that access but not modify x
```

B-tree example: branching factor=1001, height=2



1 node,
1000 keys

1001 nodes,
1,001,000 keys

1,002,001 nodes,
1,002,001,000 keys

- B-trees stored on disks, often have branching factors 50 … 2000
- keep the root node permanently in the MM $\Rightarrow$ find any key with at most two disk accesses

B-tree definition

A B-tree T is a rooted tree (where T.root is the root) with the following properties:

1. every node x has the following attributes
   a. $x.n$ – the number of keys currently stored in x
   b. the keys $x.key_1$, $x.key_2$, …, $x.key_{x.n}$ so that

   $$x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$$

   c. $x.leaf$ – a boolean value which is TRUE if x is a leaf and FALSE if x is an internal node
2. each internal node x has $x.n+1$ pointers $x.c_1$, $x.c_2$, …, $x.c_{x.n+1}$ to its children; if x is a leaf then its c_i attributes are undefined
3. if k_i is any key stored in the subtree with root $x.c_i$ then:

   $$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$

B-tree definition, cont.

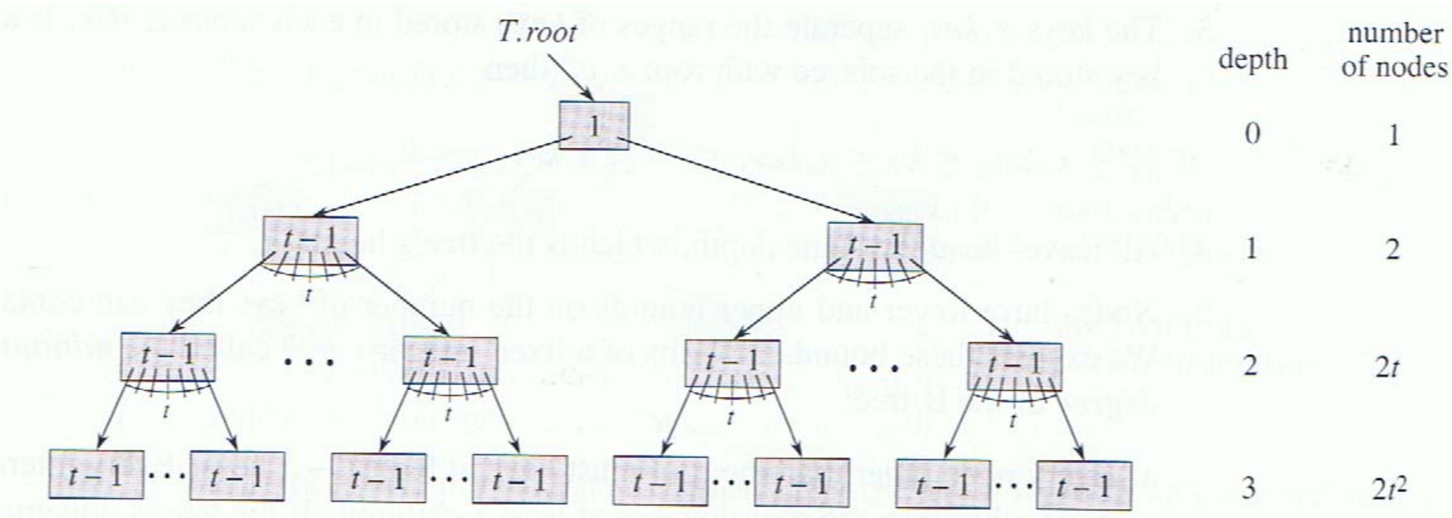
4. All leaves have the same depth, which is the tree high *h*.
5. the B-tree has a **minimum degree** *t* (*t* is an integer $t \geq 2$):
   - Every node other than the root must have $\geq t - 1$ keys and $\geq t$ children; if B-tree is nonempty, then the root has at least one key
   - Every node has $\leq 2t - 1$ keys and $\leq 2t$ children

   A node is full is it has $2t - 1$ keys.

The height of a B-tree

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

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



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

The height of a B-tree, cont.

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

$$= 1 + 2(t-1)\frac{t^h - 1}{t-1} = 2t^h - 1$$

$$t^h \leq \frac{n+1}{2}$$

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

h = O($\log_t n$)

$\boxed{\text{h} = O(\lg n)}$

Basic operations on B-trees

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

Conventions

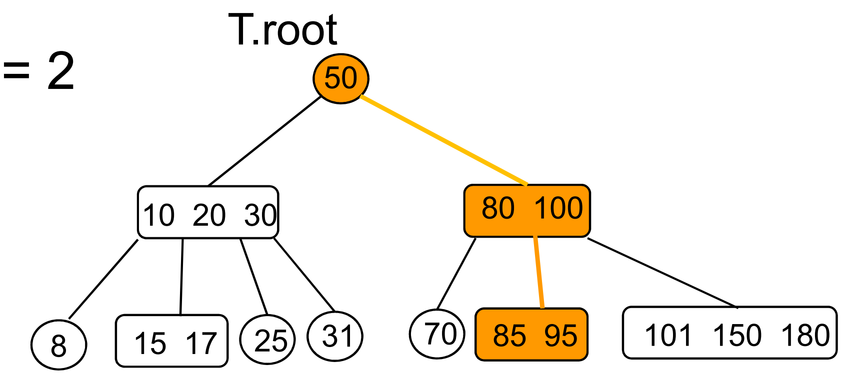
- The root of the B-tree is always in the main memory
  - NO need to call DISK-READ for the root
  - need to call DISK-WRITE when the root is changed
- All nodes passed as parameters must have already had a DISK-READ operation performed on them
- The procedures are **“one-pass” algorithms** that proceed downward from the root, w/o having to back up

Searching a B-tree

- At each node make a (x.n +1) – way branching decision

B-TREE-SEARCH(T.root, 85)

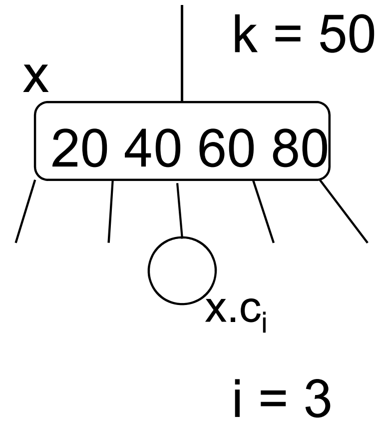
t = 2



Search operation

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 == TRUE
        return NIL
else DISK-READ(x.c_i)
        return B-TREE-SEARCH(x.c_i, k)
```



Initial call: B-TREE-SEARCH(T.root, k)

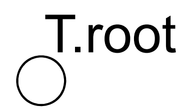
RT = $O(t \cdot \log_t n)$

- while loop takes $O(t)$
- number of recursive calls is $O(h) = O(\log_t n)$

Creating an empty B-tree

- Creates an empty root node

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```



RT = O(1)

Insert operation - main functions

- B-TREE-SPLIT-CHILD(x,i)
- B-TREE-INSERT(T,k)
- B-TREE-INSERT-NONFULL(x,k)

Insert operation

- Search for a leaf where to insert the new key
- Insert into an existing leaf node
  - Cannot create a new leaf
- If the leaf node is full, then split around the median key

Insert operation

- Goal: insert the key in the B-tree in a single pass from the root to a leaf
  - As the algorithm travels down the tree, it splits each full node along the way, including the leaf

Splitting a node in a B-tree

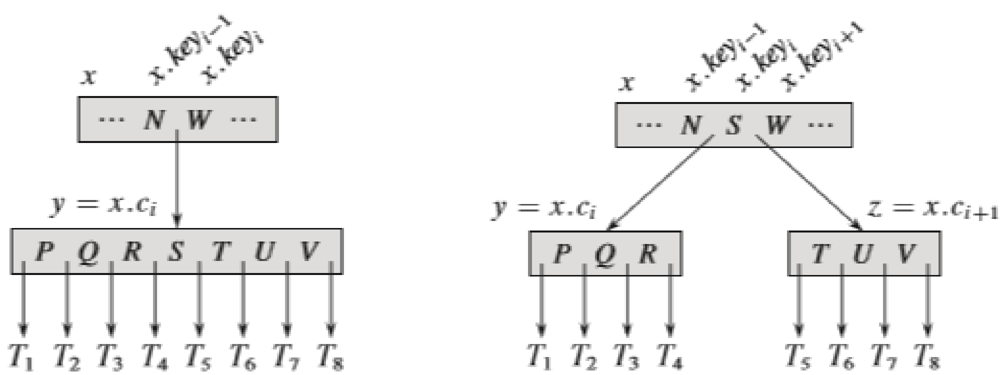
B-TREE-SPLIT-CHILD(x,i)

- Input:

  x – nonfull internal node (in the main memory)

  index i s.t. $\begin{cases} x.c_i \text{ is a full child of } x \\ x.c_i \text{ is in the main memory} \end{cases}$

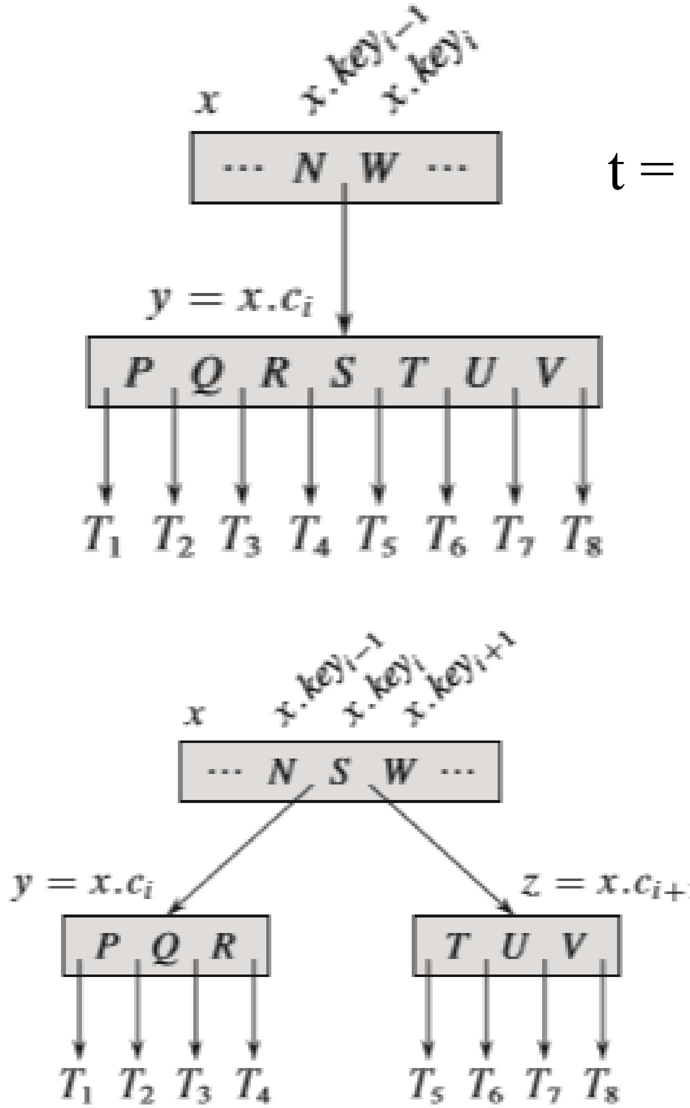
- Output: split the node $x.c_i$ around its median key $x.key_t$



t = 4

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)
```



t = 4

B-TREE-SPLIT-CHILD

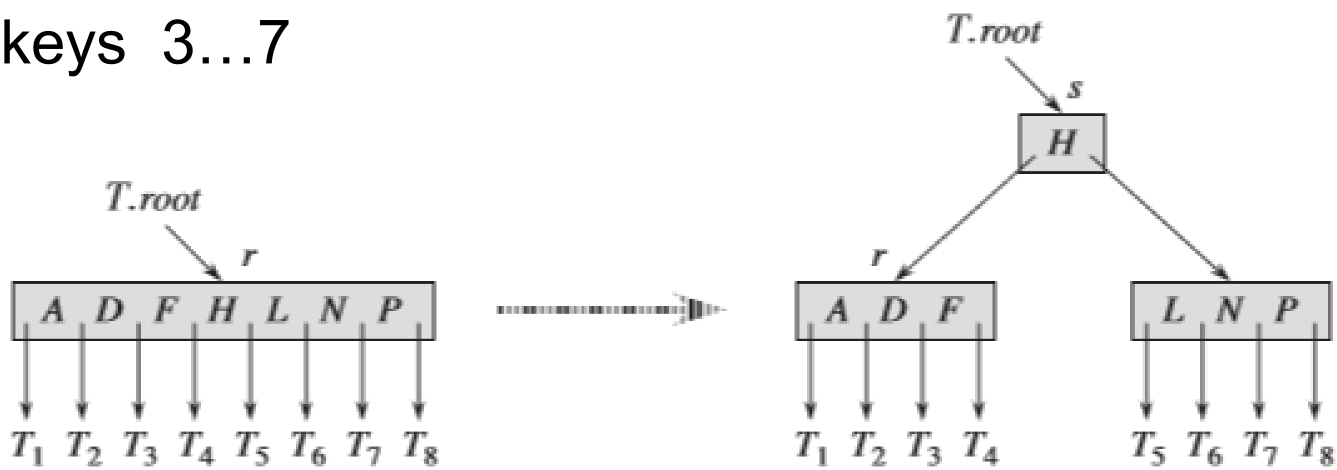
RT = $\Theta(t)$

$\Theta(1)$ disk operations

B-TREE-INSERT()

t = 4

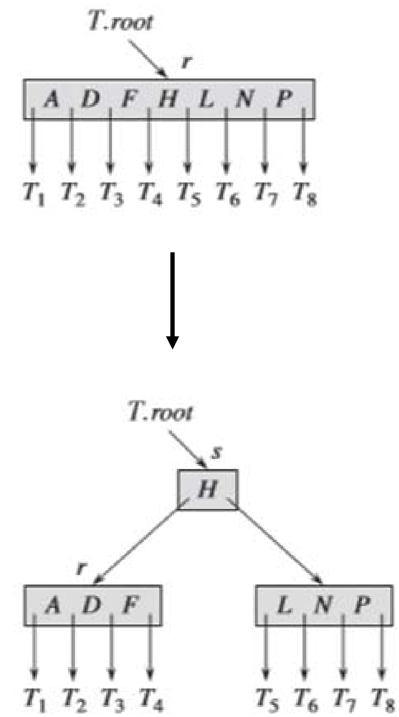
keys 3…7



- If the root r is full, then split r and a new node s becomes the root

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

$t = 4$
\# keys
3…7

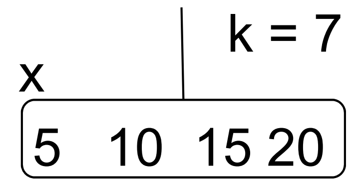


B-TREE-INSERT-NONFULL(x,k) – inserts key k into the subtree rooted at the **nonfull** node x

- node x must be nonfull when the procedure is called !

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

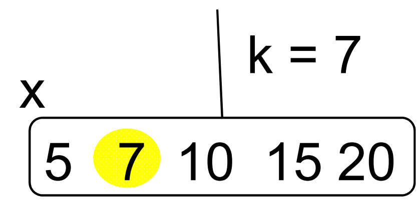
<u>x is a leaf:</u>



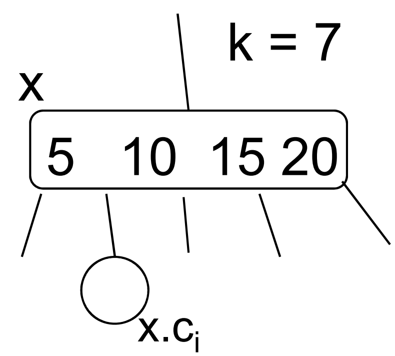
i = 4

….

i = 1



<u>x is NOT a leaf:</u>



RT for the insert operation

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

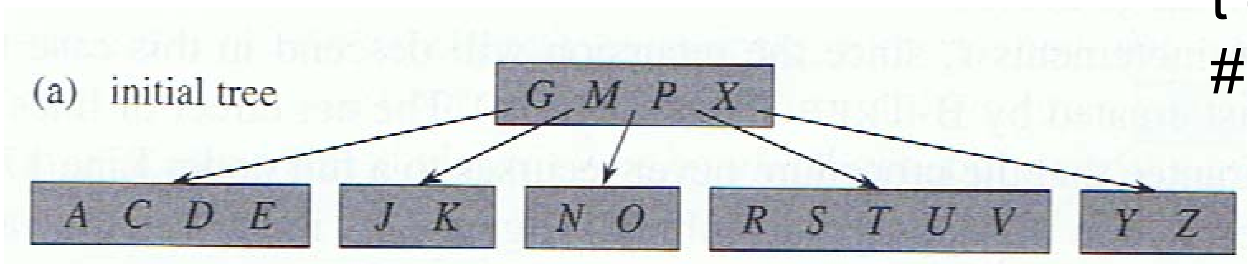
$O(h)$ disk access operations

Example

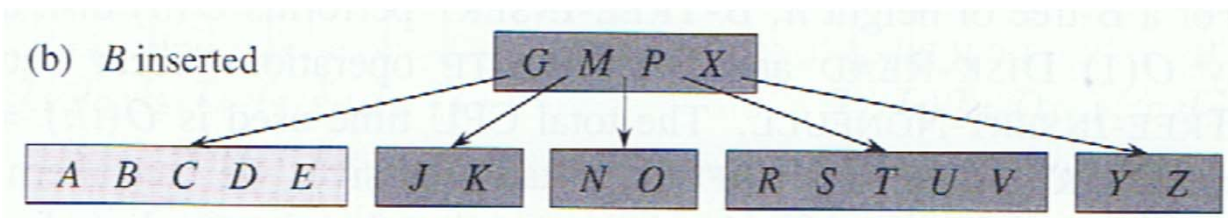
Initial Tree:

t = 3

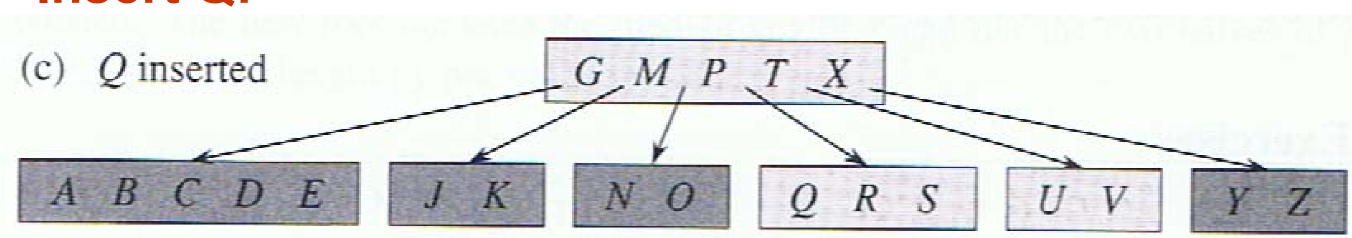
\# keys 2…5



Insert B:



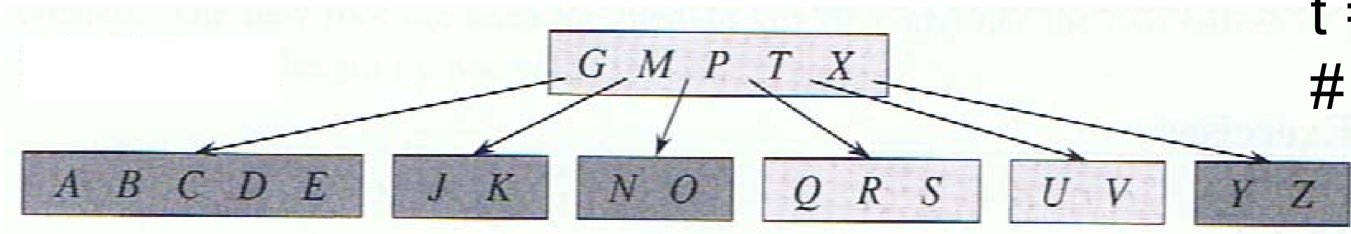
Insert Q:



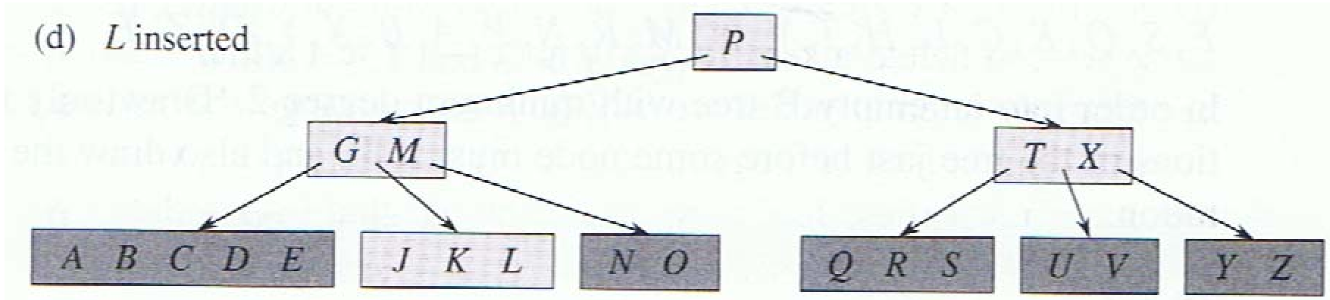
Example, cont.

Consider the B-tree:

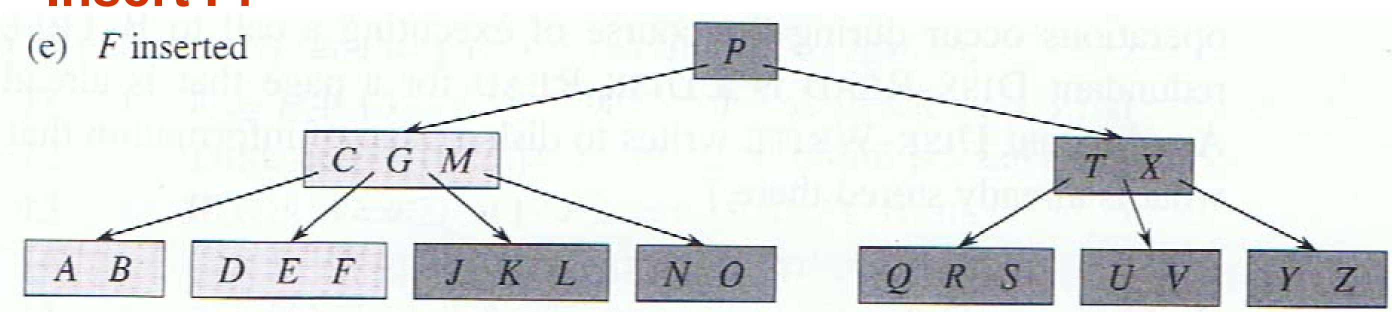
t = 3
# keys 2…5



Insert L:



Insert F:



Delete operation

Aspects to consider:

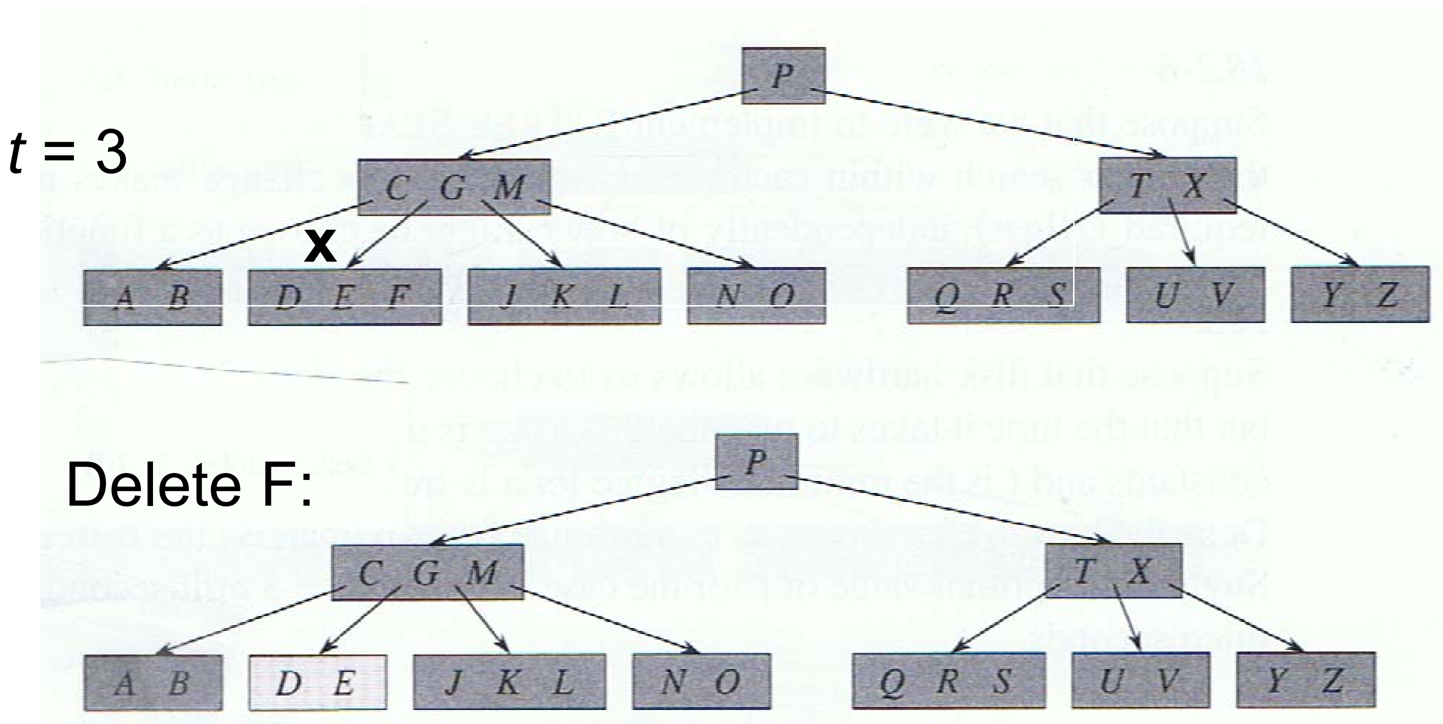
- When deleting a key from an internal node, rearrange the node's children
- Any node (except the root) cannot have fewer than $t - 1$ keys
- B-TREE-DELETE(x,k) – deletes key k from the subtree rooted at x
- Idea : when calling delete on a node x, guarantee that the number of keys in x is $\geq$ t
  - sometimes a key has to be moved to a child before recursion descends to that child

Delete operation

- Goal: delete a key from the tree in one downward pass w/o having to "back-up"
- If the root x becomes an internal node with no keys, then delete x and $x.c_1$ (the only child !) becomes the new root of the tree
  - decrease the height of the tree by 1
- Next, we discuss the rules for deleting keys from a B-tree

Rule 1

- If the key $k \in$ to the LEAF node x, then delete the key k from x



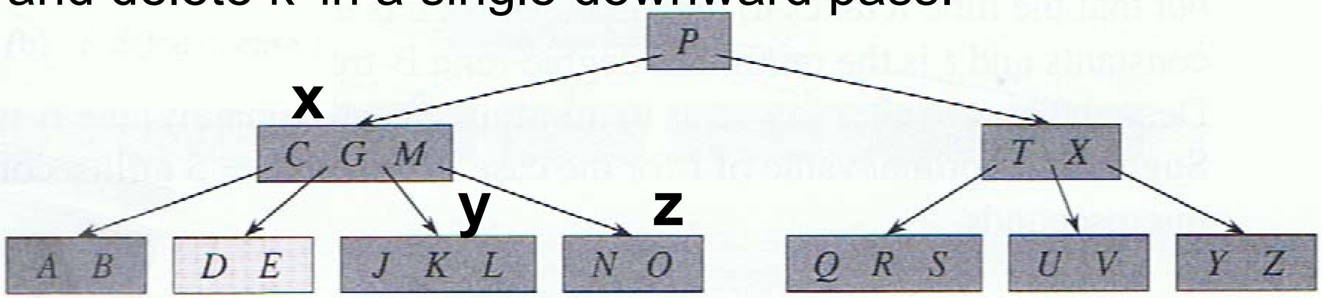
Rule 2

- If the key $k \in$ to the internal node x :

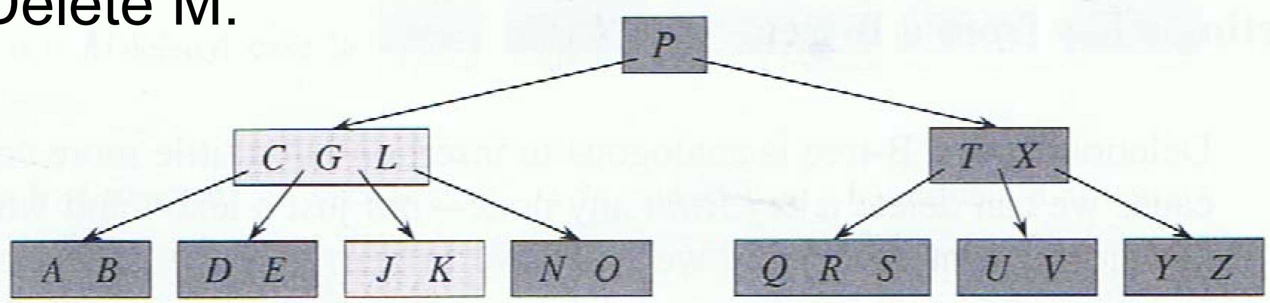
a. if the child y that precedes k in a node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k' and replace k by k' in x.

Find and delete k' in a single downward pass.

$t = 3$



Delete M:



Rule 2, cont.

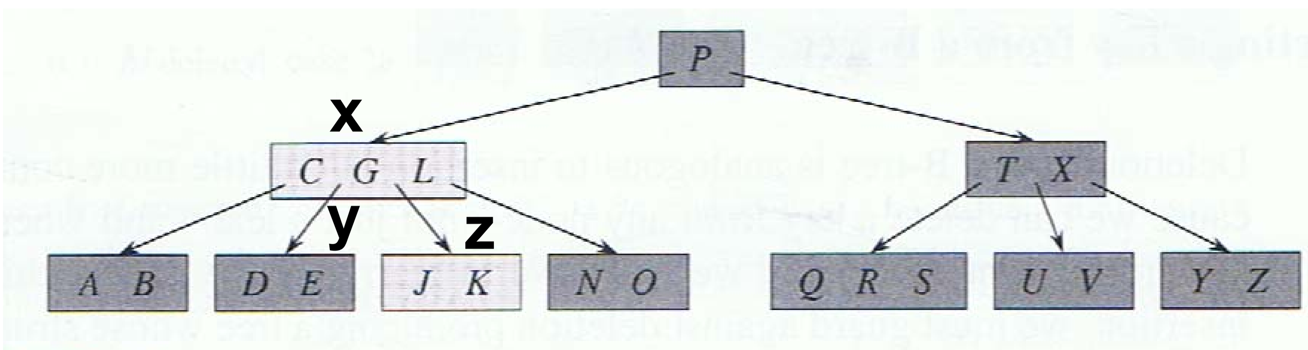
b. if y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k' and replace k by k' in x.

   Find and delete k' in a single downward pass.

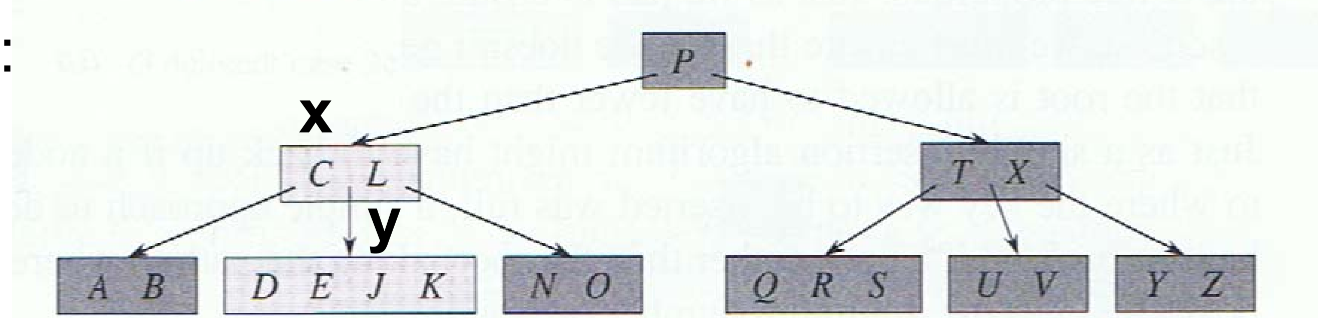
Rule 2, cont.

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 2*t*-1 keys. Then free *z* and recursively delete *k* from *y*.

$t = 3$



Delete G:



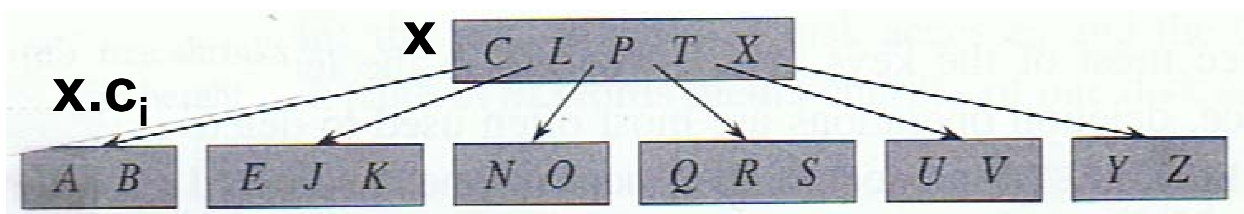
Rule 3

- If $k \notin$ to the internal node x, take $x.c_i$ the root of the subtree that must contain k (if k is in the tree). If $x.c_i$ has only $t\text{-}1$ keys, then use 3a or 3b to guarantee we descend to a node with $\geq t$ keys

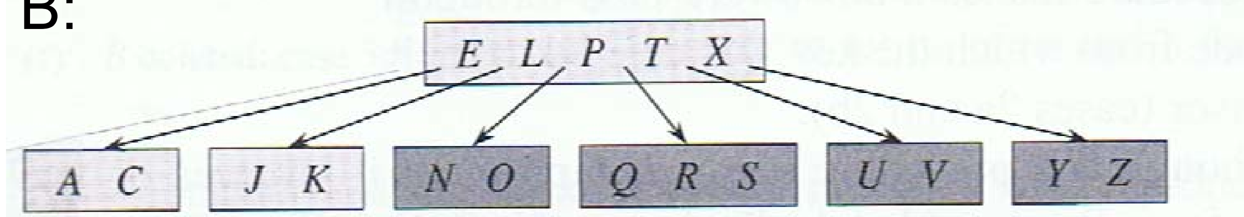
a. If $x.c_i$ has an immediate sibling with $\geq t$ keys, then give $x.c_i$ an extra key by:

- moving a key from x to $x.c_i$,
- moving a key from $x.c_i$'s immediate left or right sibling up to x,
- moving the appropriate child pointer from the sibling into $x.c_i$

$t = 3$



Delete B:

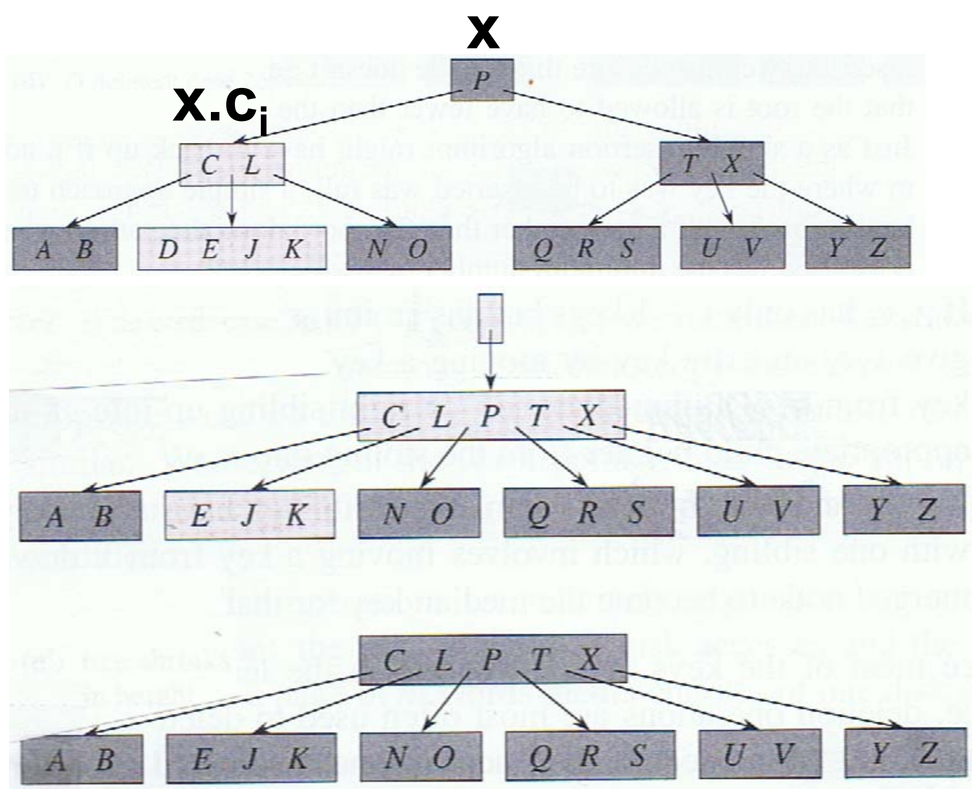


Rule 3, cont.

b. If both $x.c_i$'s immediate siblings have t-1 keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median for that node

Delete D:

The tree shrinks in height



Delete operation, RT analysis

- Most of the keys are in the leaves
  - In practice, most often delete keys from the leaves
- One downward pass through the tree, w/o having to back up
  - Cases 2a & 2b: make a downward pass through the tree. Return to the node where the key was deleted to replace it with the predecessor/successor key.
- O(h) disk operations
  - O(1) calls to DISK-READ and DISK-WRITE between recursive invocations of the procedure.
- RT is $O(th) = O(t \log_t n)$