Chapter 18 B-tree Operations

Search C455

Document last modified: 03/05/2014 23:03:55

Overview

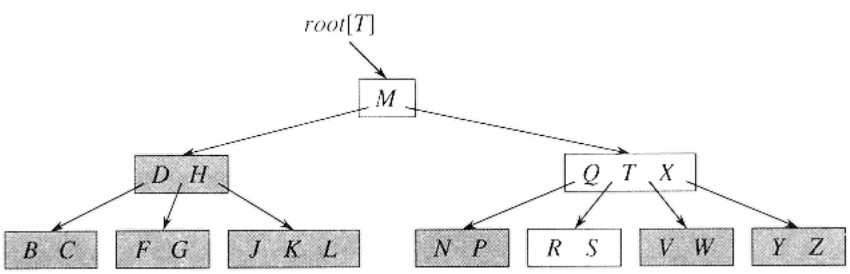
- Dynamic Set Operations: *search, insert* and *delete*
- B-tree operations are one-pass, without backing up

Question 18.7

- What's the big deal about operations that are "one-pass without backing up"?
- The root is normally locked in memory. Why is that reasonable?

Cormen Implementation as C++

```
class Node_Rep;
typedef Node_Rep * Node;

class Node_Rep {
   public:
       Item key[2t - 1];        -- keys
       Node c[ 2t ];             -- child nodes
       Boolean leaf;
       Integer n;                -- number of keys
};

variable Node T;
```



Example

- T.root = Node M
- $T.c_1$ = Node DH
- $T.key_1$ = M
- $T.c_2.key_3$ = X
- $T.c_1$.leaf = false
- $T.c_1.n$ = 2

Question 18.8 For the B-tree T at right that contains the 26 lower-case letter keys, what are:

a. $T.key_1$

b. $T.c_2.key_3$

c. $T.c_1.n$

d. $T.c_1.c_1.n$

e. $T.c_1.c_1.key_3$

f. The

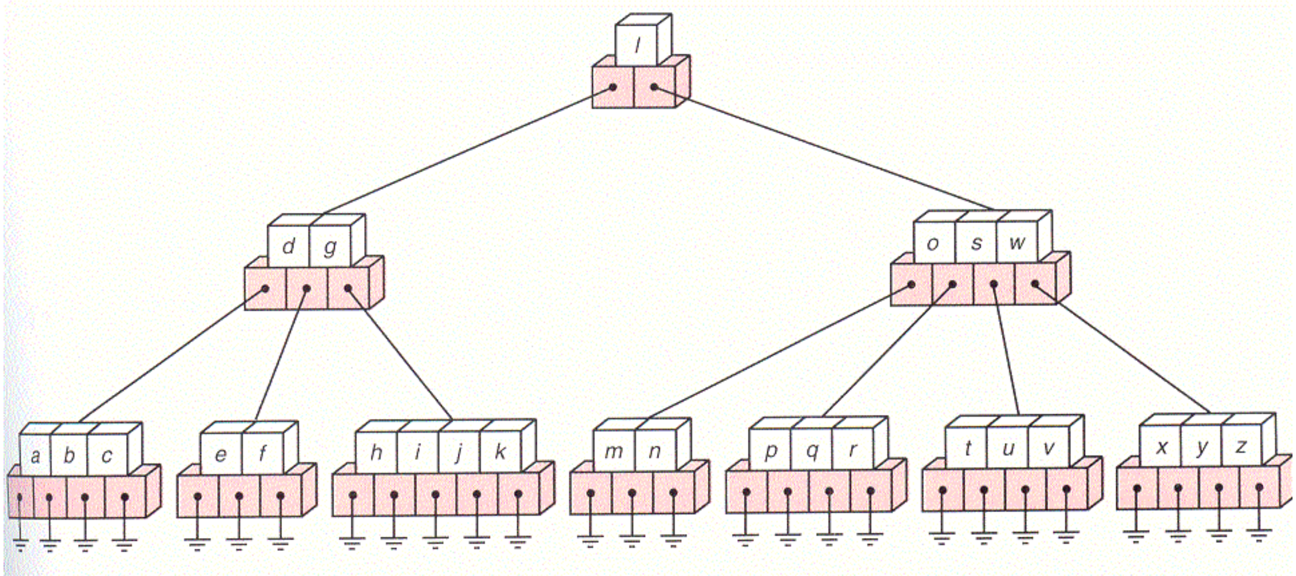


Figure 11.9. A B-tree

minimum degree of the tree is t=3.

- What is the height, h?
- What is *n*?
- Is that consistent with

  $h \le \log_t (n + 1)/2$?

  $\log_3 9=2 \quad \log_3 27=3$

Searching

- Each node holds up to 2t - 1 keys sorted in non-decreasing order.
- Searching node x involves a linear search of the keys in x.
  - Example: At node *hijk*, the search for key *j* must examine h and i.
- If the key is found in node x, k = $x.key_i$, then a two tuple is returned containing a pointer to the node and the index of the exact location of the key k.
  - Example: On finding key *j* returns node *hijk* reference and index *3.*
- If the key is not found in node x, the search of x's keys stopped under two conditions:
  - there are no more keys and no more children to examine, NULL is returned meaning k was not found
    - Example: Search for key *zz* ends after examining key *z.*
  - there are no more keys or k < $x.key_i$, meaning that k may be in the subtree referenced by $x.c_i$ pointer.
    - A recursive call is made to B_Tree_Search is made with $x.c_i$ pointer to the subtree

      Example: Search for key *j* requires recursion on $l.c_1$ then $dg.c_3$

      Example: Search for key *zz* requires recursion on $l.c_1$ then $osw.c_4$

```
(Node, Integer) B_Tree_Search (preserves Node x, preserves Item k)
  i ← 1
  while i ≤ x.n  and  k > x.key_i do              -- linear search
    i ← i + 1

  if i ≤ x.n and k = x.key_i then                 -- key match
    return (x, i)

  if x.leaf then                                  -- failed search
    return NULL
  else
    Disk_Read (x.c_i)                             -- next level
    return B_Tree_Search (x.c_i, k)
```

Question 18.9

a. Trace the algorithm for tree at right using

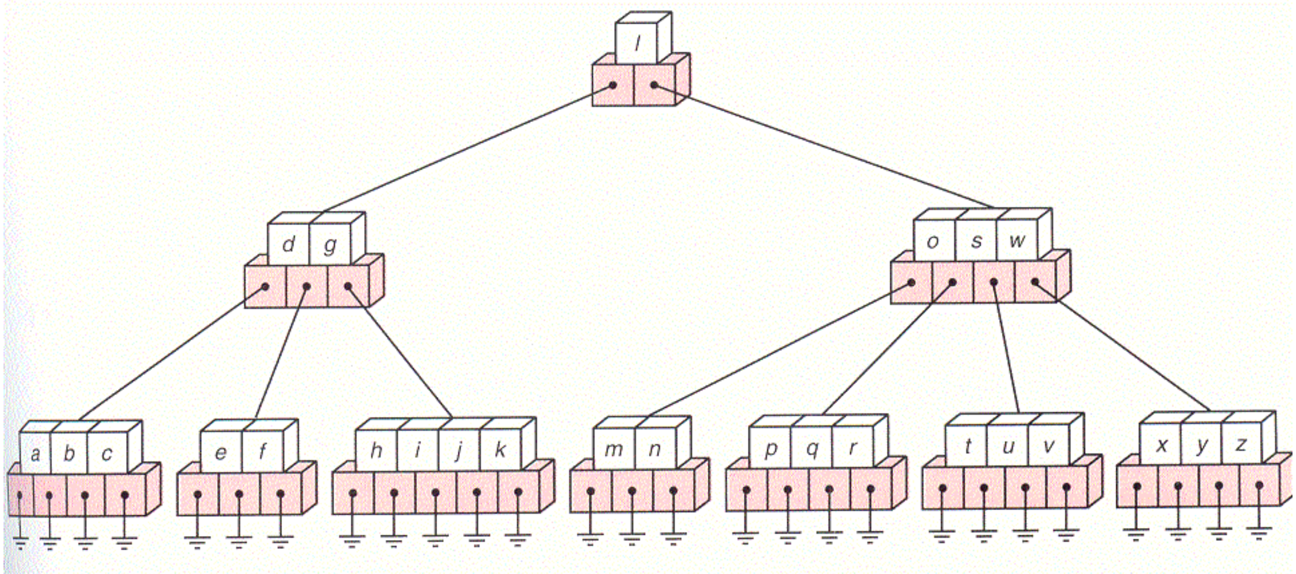


Figure 11.9. A B-tree

B_Tree_Search (T, a).
   - Keep count of the number of comparisons made in the **while.**
   - How many times is *Disk_Read (x.c_i)* executed?

b. Trace the algorithm for tree at right using *B_Tree_Search (T, z).*

   Note x has x.n keys and x.n+1 children.

   - Keep count of the number of comparisons made in the **while.**
   - How many times is *Disk_Read (x.c_i)* executed?

c. Give an upper-bounds estimate of recursions. Use *h.*

d. Give an upper-bounds estimate of comparisons in each **while** using *t.*

e. Give an upper-bounds estimate of comparisons for the **while**s in all recursions using *t* and *h.*

f. Which dominates where it matters - the Disk_Read (x.c_i) or the **while**s? Explain.

g. The greater the value of *t* the _____________ tree depth and the _________ the number of disk accesses.

h. Search is a one-pass operation, what would you estimate the upper-bounds for other one-pass B-tree operations? Use *h.*

i. Explain finding the minimum B-tree key.

j. Explain finding the successor of B-tree keys *l, k, p* and *r*.

   m = successor(l)

   l = successor(k)

   q = successor(p)

   s = successor(r)

k. Explain finding the predecessor of B-tree keys *l, n, p* and *m*.

   k = predecessor(l)

   m = predecessor(n)

   o = predecessor(p)

   l = predecessor(m)

l. Are predecessor and successor one-pass operations?

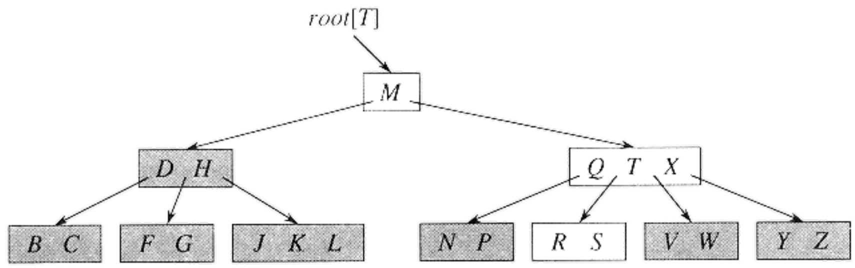
(Node, Integer) **B_Tree_Search** (preserves Node x, preserves Item k)
  i ← 1
  while i ≤ x.n **and** k > *x.key$_i$* **do**   -- linear search

```
    i ← i + 1

  if i ≤ x.n and k = x.key_i then                -- key match
    return (x, i)

  if x.leaf then                                 -- failed search
    return NULL
  else
    Disk_Read (x.c_i)                            -- next level
    return B_Tree_Search (x.c_i, k)
```

Insertion

Assume the B-tree at right has t=2.

a. The new key k must be inserted into an existing leaf node so that the *search tree property* is maintained.

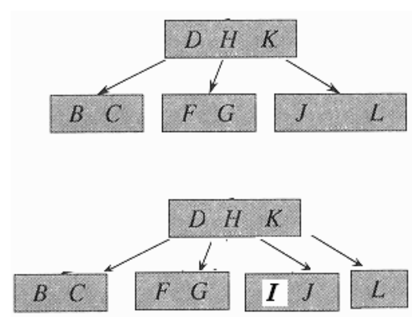


Question 18.10 Where should *O* be inserted?

b. When inserting a new key k, a new leaf cannot be created (as is the case with a binary tree), because this would violate the property that all leaf nodes must be at the same depth.

Question 18.11 Where should *I* be inserted? 2t-1 = 3 is maximum number of keys.

c. Splitting a Full Node:



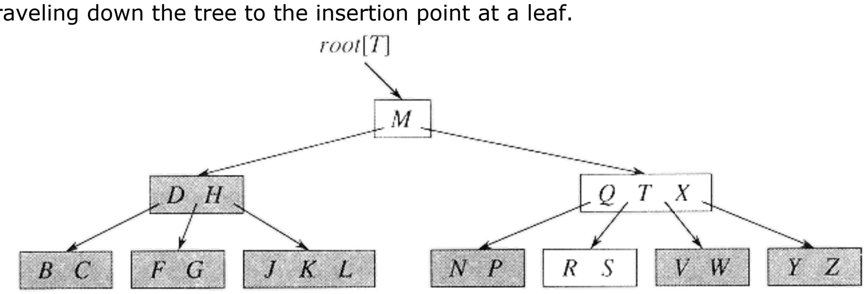
- If the leaf is already full (with *2t-1* keys) then it must be split into two nodes each node having t-1 keys.
- The median value of the original full node moves up to the parent node to become the value that divides the two new child nodes.
  - Note: 2(t-1) + 1 = 2t-1
- When the median value is moved up to the parent, the parent node might also be full, therefore causing it to split.
- When this node splitting happens all the way back up to the root causing the root to split, the B-Tree grows in height by 1 level.
- B-Trees therefore grow (and shrink) at the root.

Example: Insert *I*, because *JKL* is full, requires moving median key (e.g. K) to parent and splitting *JL*.

Question 18.12 Why move K to parent?

d. Aggressive Node Splitting

- Full nodes (with *2t-1* keys) are split while traveling down the tree to the insertion point at a leaf.
- Done to avoid possible propagation of values back up the tree (because of splitting).
- When the leaf is reached where key k must be inserted, if the leaf must be split it is guaranteed that the parent will not also be full.



Question 18.13

- Given what we saw in c. where a full leaf was split and the median moved into the parent, what would be effect of *not* performing aggressive node splitting on the way down.
- Would QTX need to be split in order to insert O? Would it be split under aggressive node splitting?
- Using aggressive node splitting, what is the result tree after inserting O? Splitting moves the median key to the parent and divides the remaining keys into two nodes.

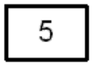
Example of Insertion with Splitting

We will apply the stated rules above first and examine the insertion algorithms later.

Let *t* = 2. A node is then full if it has 2(2)-1 = 3 keys in it, and each node can have up to 4 children.

Insert: 5 9 3 7 1 2 8 6 0 4:

Step 1: Insert 5

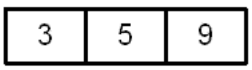


Step 2: Insert 9

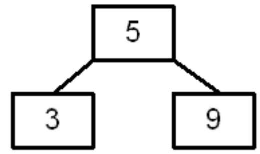
Root node not full.
Put 9 to the right of 5

Step 3: Insert 3

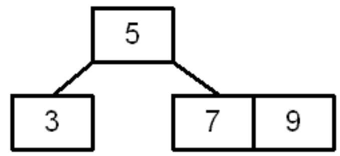
Root node not full.
Put 3 to the left of 5



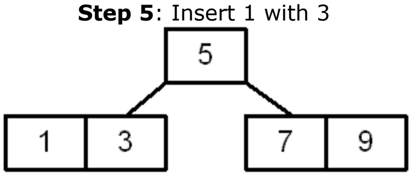
Step 4: Insert 7
Root node is full, do aggressive node splitting.
Allocate a new (empty) node; make it the root,
split pulling median key 5 into the new root:



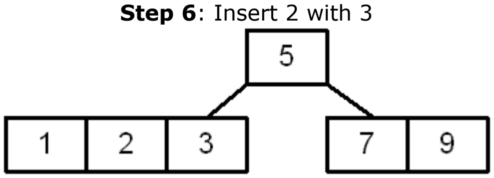
Then insert 7; with 9



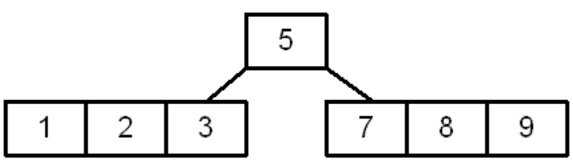
Step 5: Insert 1 with 3



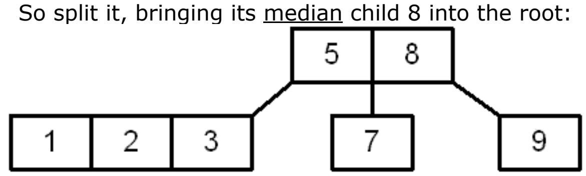
Step 6: Insert 2 with 3



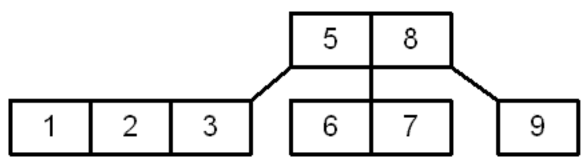
Step 7: Insert 8 with 9



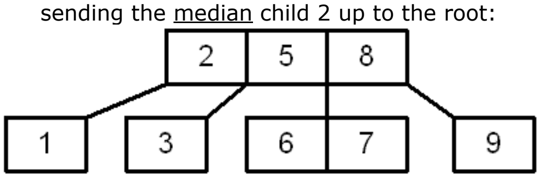
Step 8: Insert 6
It would go in with 7 8 9, but that node is full.
So split it, bringing its median child 8 into the root:



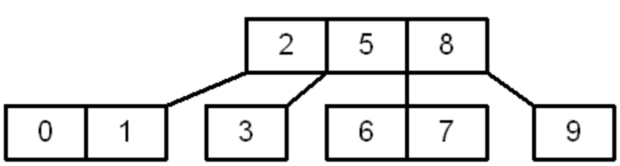
Then insert 6 with 7:



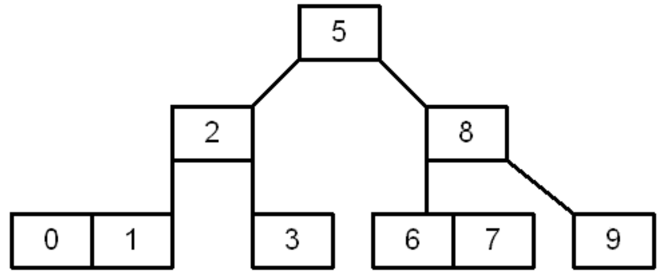
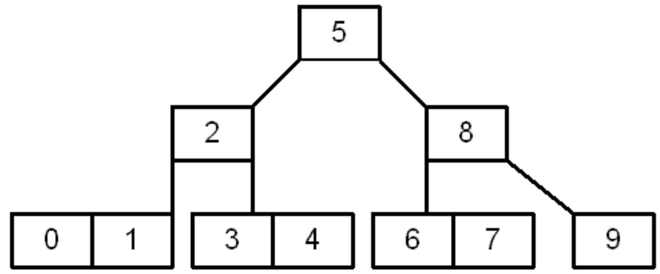
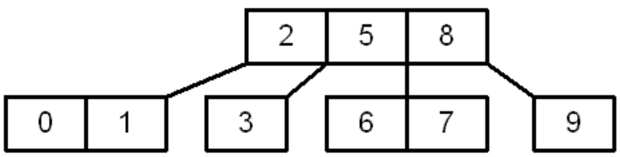
Step 9: Insert 0
0 would go in with 1 2 3, which is full, so split it,
sending the median child 2 up to the root:



Now put 0 in with 1



Step 10: Insert 4
It would be nice to just stick 4 in with 3,
but the B-Tree algorithm requires splitting the full root
(or any full node encountered - Aggressive Node Splitting).
If we don't and one of the leaves becomes full,
nowhere for median key of split node since root would be full,
aggressive splitting prevents backing the median key up more
than one level:

<table>
<tr>
<td></td>
<td>
Assured room for the median key in case of a lower-level split, we can insert 4:
</td>
</tr>
<tr>
<td>Question 18.14

a. Try inserting a -1 and -2 without splitting the root node, t=2.
b. Repeat, inserting a -1 and -2 with Aggressive Node Splitting.
c. What purpose is served by Aggressive Node Splitting in Step 10?</td>
<td></td>
</tr>
</table>

Splitting a Full Node

- Splitting a Full Node:
  - If the leaf is already full (with 2t - 1 keys) then it must be split into two nodes each node having t - 1 keys.
  - The median value of the original full node moves up to the parent node to become the value that divides the two new child nodes.
  - When the median value is moved up to the parent, the parent node might also be full, therefore causing it to split.
  - When this node splitting happens all the way back up to the root causing the root to split, the B-Tree grows in height by 1 level.
  - B-Trees height therefore grow (and shrink) at the root.
- Aggressive Node Splitting
  - Full nodes are split while traveling down the tree to the insertion point.
  - This is done in order to avoid this possible propagation of values back up the tree (because of splitting).
  - So, when the leaf is reached where key k must be inserted, then if the leaf must be split, it is guaranteed that the parent will not also be full.

Figure 18.5 for example of node splitting, t=4.

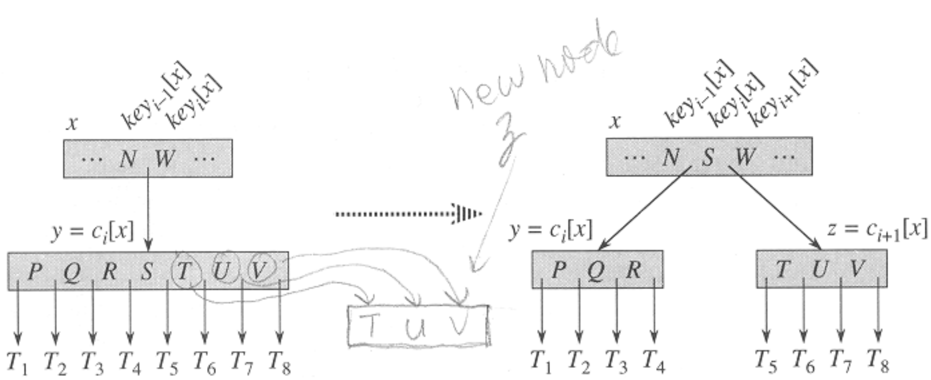
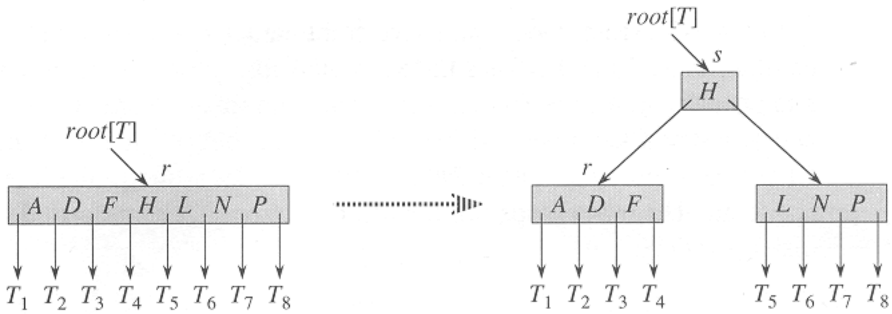
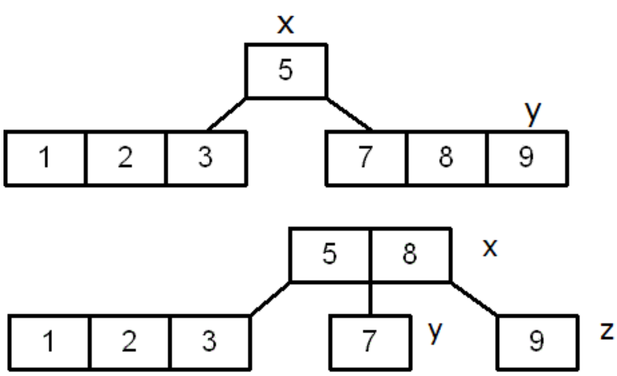


Figure 18.6 example of *root* node splitting, t=4



Question 18.15 What is the key difference between splitting the root and any other node?



t = 2

i = 2

y.n = 2t-1 = 3

```
void B_Tree_Split_Child (Node x, Integer i)
-- pre: x.n < (2t - 1)  true given that x is the parent
--      and subjected to aggressive node splitting

-- i index where parent should receive median key from y
-- x pointer to parent node of y
-- y pointer to node to be split, y = x.c_i
-- z new node receiving "1/2" y node's keys and children
-- t is the index of the median node in a full node

   z ← Allocate_Node ()                  -- initialize new node z
   y ← x.c_i
   z.leaf ← y.leaf
   z.n ← (t - 1)

   for j ← 1 to (t - 1) do               -- copy 1/2 keys from y (node being split) to new z node
     z.key_j ← y.key_(j + t)             -- 2t-1 keys in y, so y and z will have t-1 keys

   if not y.leaf then                    -- y (node being split) is not a leaf, copy children to z
     for j ← 1 to t do
       z.c_j ← y.c_(j + t)

   y.n ← (t - 1)                         -- In parent node x, make room for child pointer to z node
   for j ← (x.n + 1) downto (i + 1) do
     x.c_(j + 1) ← x.c_j                 -- i is index where new child goes, slide existing children up
```

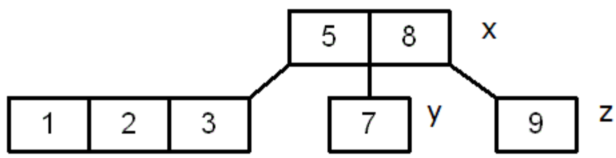
$x.c_{(i+1)} \leftarrow z$
for j ← x.n **downto** i **do** -- In parent node x, make room for the median key from y node
  $x.key_{(j+1)} \leftarrow x.key_j$ -- i is index where new key must go, slide existing keys up

$x.key_i \leftarrow y.key_t$ -- t is index of median node of full node y, copy to y's parent
x.n ← x.n + 1 -- one more key in x

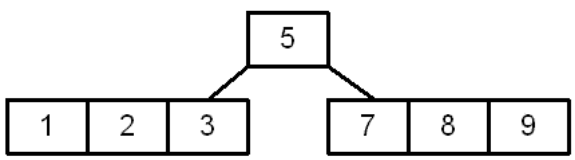
Disk_Write (y)
Disk_Write (z)
Disk_Write (x)

Question 18.16 t=2

1. Explain why *t=index of median key* in node to split?
2. Why the 3 Disk_Writes?



3. Perform split necessary to insert 6.



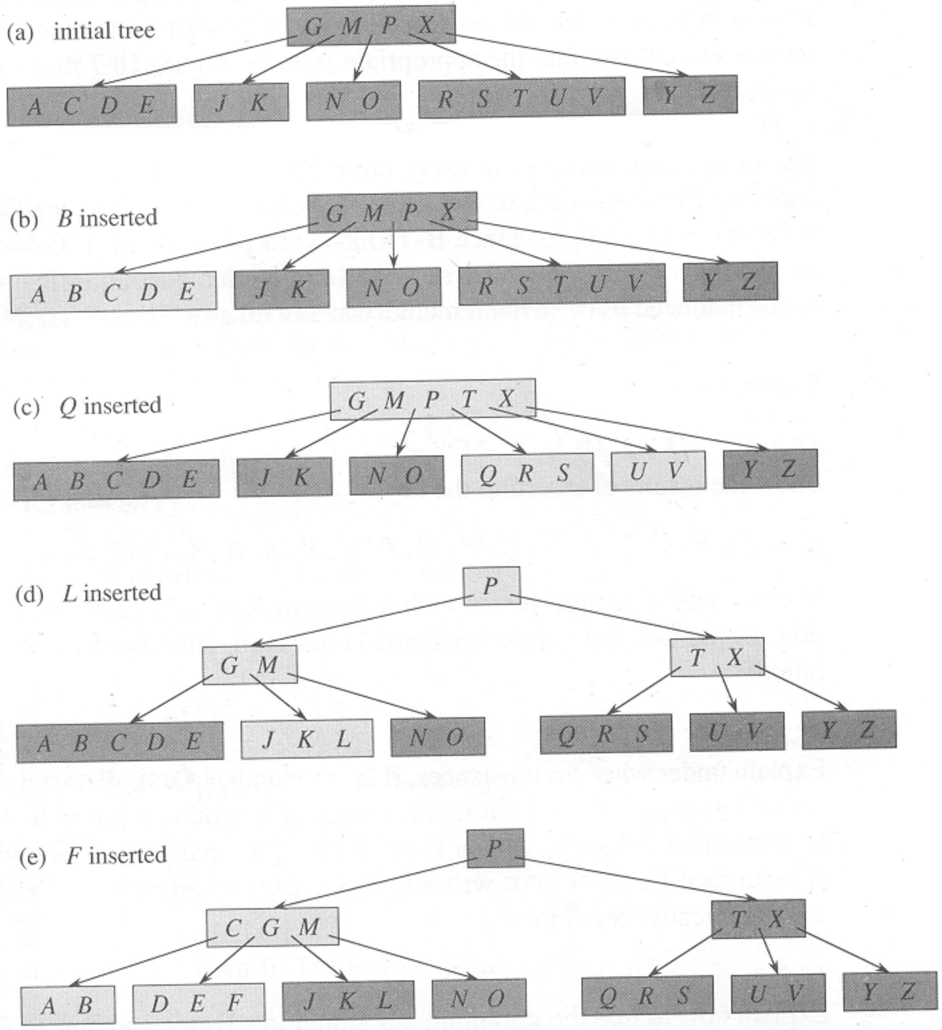
4. Perform the additional split necessary to insert 4.

Inserting a key in a single pass down the tree

- The new key k must be inserted into an existing leaf node so that the search tree property is maintained.
- Non-leaf nodes add keys only through splitting, not insertion.
- When inserting a new key k, a new leaf cannot be created (as is the case with a binary tree), because this would violate the property that all leaf nodes must be at the same depth.

  Figure 18.7 Inserting keys

  t = 3, the minimum degree, maximum of 5 keys

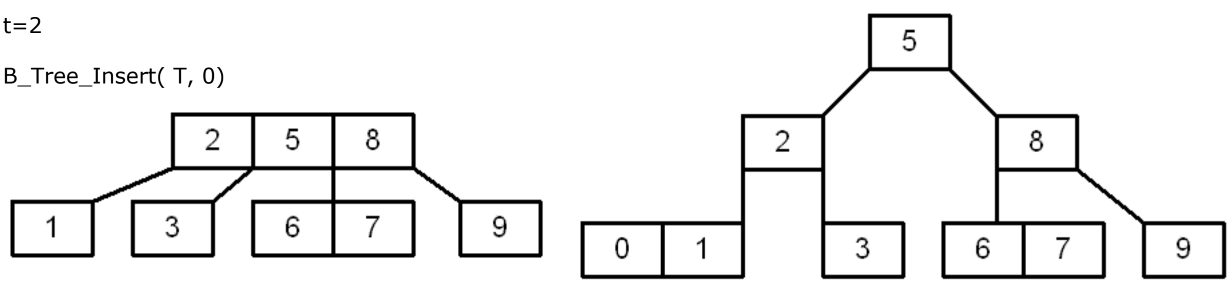


Question 18.17

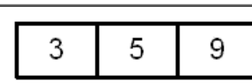
- Explain what occurs at (c), (d) and (e).
- How many Disk_Writes are required in (b) and (d)?

t=2

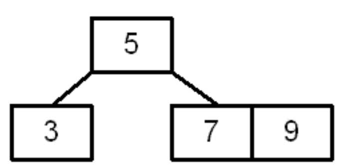
B_Tree_Insert(T, 0)




```
void B_Tree_Insert (Tree T, Item k)
   r ← T.root
   if r.n = (2t - 1) then                  -- root is full, create new root,
      s ← Allocate_Node ()                   -- split old root into 2 leaves
      T.root ← s
      s.leaf ← FALSE
      s.n ← 0
      s.c1 ← r                               -- old root is now the child to split
      B_Tree_Split_Child (s, 1)            -- new root gets median key of old root
      B_Tree_Insert_Nonfull (s, k)
   else
      B_Tree_Insert_Nonfull (r, k)
```



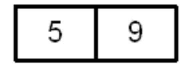
B_Tree_Insert(T, 7)

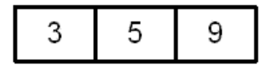



```
void B_Tree_Insert_Nonfull (Node x, Item k)
```

<table>
<tr>
<td>
i ← x.n

if x.leaf then -- at a non-full leaf, insert k

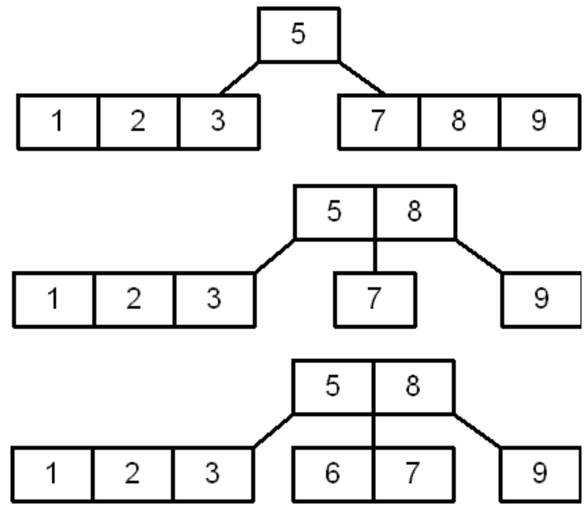
  while i ≥ 1 and k < $x.key_i$ do -- search for k's insertion point,
    $x.key_{(i+1)} \leftarrow x.key_i$ -- and slide existing keys up
    i ← i - 1
  $x.key_{(i+1)} \leftarrow$ k -- insert k
  x.n ← x.n + 1 -- increment key count
  Disk_Write (x)
</td>
<td>


B_Tree_InsertNonfull(T, 3)

</td>
</tr>
<tr>
<td>
else -- not a leaf
  while i ≥ 1 and k < $x.key_i$ do -- find index to correct child subtree key
    i ← i - 1

  i ← i + 1
  Disk_Read ($x.c_i$) -- read child node into memory

  if $x.c_i$.n = (2t - 1) then -- if full child node
    B_Tree_Split_Child (x, i) -- split before inserting

    if k > $x.key_i$ then -- $x.key_i$ holds median key from child $x.c_i$
      i ← i + 1 -- insert key k in last child

  B_Tree_Insert_Nonfull ($x.c_i$, k) -- assured that child i is not full, insert k
</td>
<td>
B_Tree_InsertNonfull(T, 6)

</td>
</tr>
</table>

Question 18.18

1. When are Disk_Reads performed?
2. When are Disk_Writes performed?
3. Where is Aggressive Node Splitting performed?
4. For Disk_Reads and Disk_Writes, what is the best-case? the worst-case?

Deletion

- Key may be deleted from any node, not just leaf
- Must not allow a deletion to result in node (other than the root) with less than t-1 keys
- Deletes in one pass down tree without backing up (except for one case and that does not involve a duplicate read/write)
- Deleting from an internal node is recursive,

  meaning that child nodes must be adjusted down to a leaf node since there is one less key.

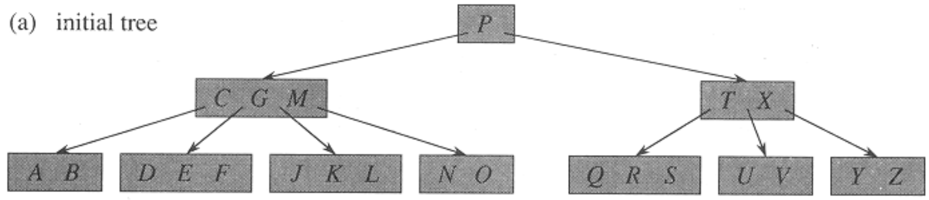
Example

Figure 18.8 Deleting keys

t = 3, the minimum degree

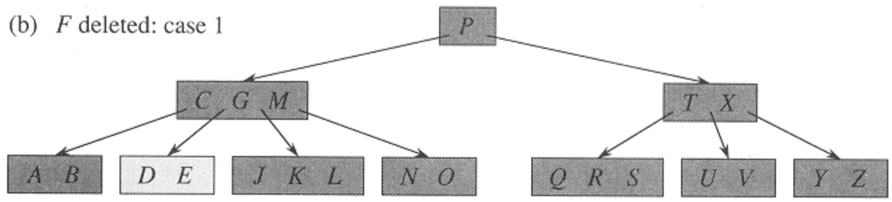
minimum 2 keys

maximum of 5 keys



Case 1: If key k is in node x and x is a leaf, delete the key k from x. Assumes has at least *t* keys in leaf x.

k=F   x = DEF



Question 18.19a - Draw the tree (b) after deleting K.

Case 2a: If key k is an internal node x

and 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.

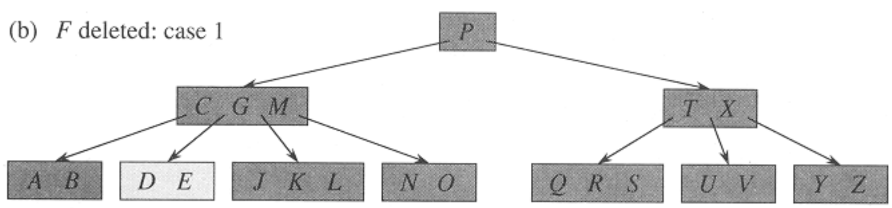
Recursively delete k' and replace k by k' in x.

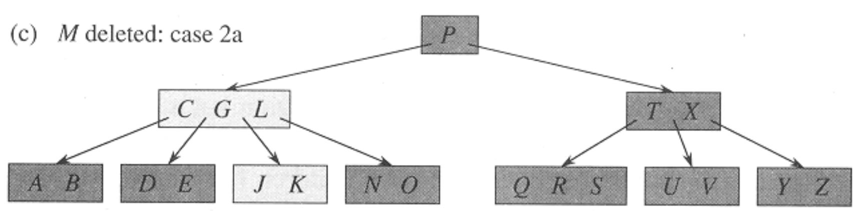
Necessary since one less key in predecessor node.

Stop at leaf.

- Finding k' and deleting in a single pass.

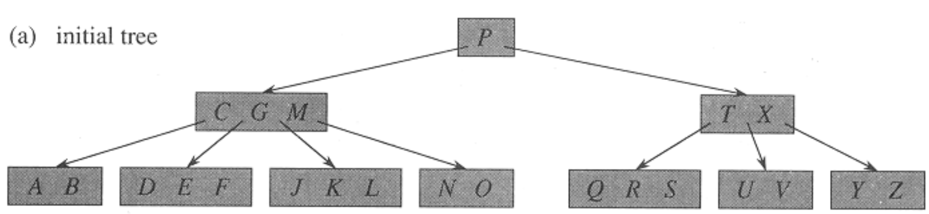
k=M   x=CGM   y=JKL   k'=L predecessor of M





Question 18.19.b

Draw the tree (a) after deleting G.



Question 18.19.c

Is *search tree property* preserved?

Case 2b: Use symmetry of predecessor of 2a with *successor*.

If the key k is an internal node x

and the child y that follows k in node x has at least *t* keys

then

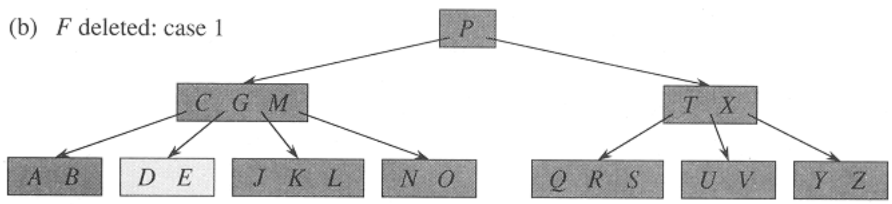
find the *successor* k' of k in the subtree rooted at y.

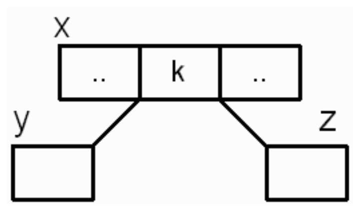
Recursively delete k' and replace k by k' in x.

Finding k' and deleting in a single pass.

Stop at leaf.

Question 18.19.d - Draw the tree (b) after deleting G.





Case 2c:

If both y and z have only t-1 keys

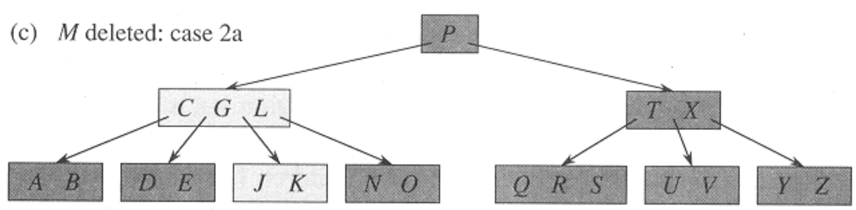
merge k and all of z into y,

so x loses both k and the pointer to z, and y now contains 2t-1 keys.

free z and recursively delete k from y.

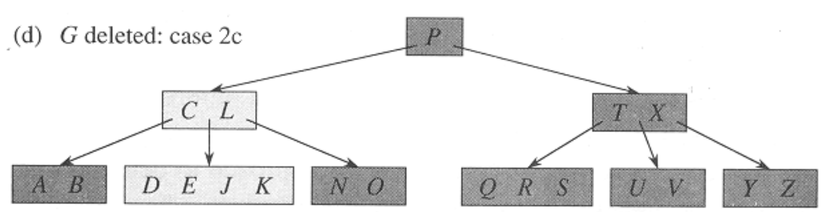
k=G x=CGL y=DE z=JK

Before deleting G



Combine DE G JK

Delete G



Question 18.20.a - Draw the tree (c) after deleting L.

How many children of DE and JK? DEGJK? DEJK?

What has to happen to CL?

Question 18.20.b - With t=3, can X be deleted this way?

Case 3:

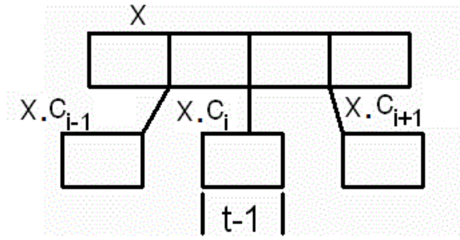
If the key k is not present in internal node x,

determine root $x.c_i$ of the appropriate child subtree that must contain k, if k is in the tree at all.

If $x.c_i$ has only t-1 keys,

execute steps 3a or 3b as necessary to guarantee descent to a node with at least t keys.

Finish by recursing on the appropriate x.c



Question 18.21

Why is it necessary to only descend into nodes that have at least *t* keys? Consider Case 2c.

Case 3b: Merge.

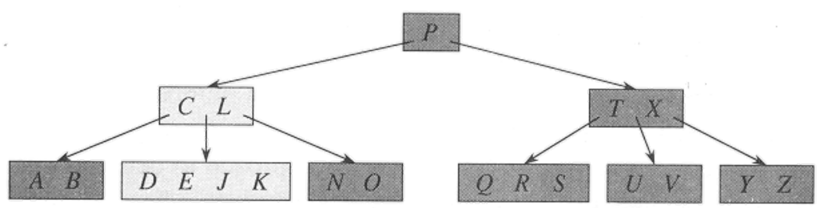
If $x.c_i$ and both of $x.c_i$'s immediate siblings have t-1 keys

merge $x.c_i$ with one sibling

which involves moving either the predecessor or successor key in x down

into the new merge node to become the median key for that node.

k=D x=P



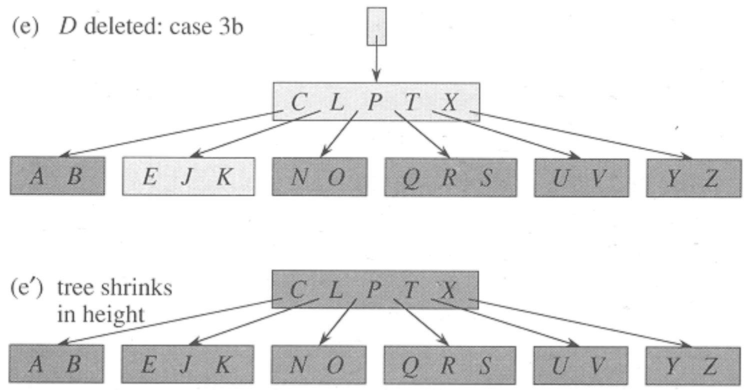
Recursion cannot descend into CL with only t-1 keys.

CL's immediate siblings, TX, also has t-1 keys.

- Merge CL with one sibling, TX, creating node with 2t-2 keys.
- Move key from x that was between the two siblings, P, to the median of the new node; now with 2t-1 keys or full.
- New node: x=CL**P**TX
- Recursively descend into node, DEJK.

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

k=D x=DEJK



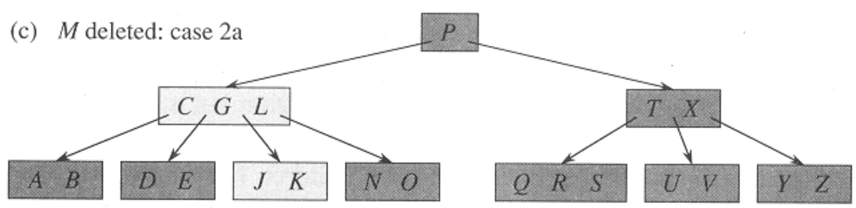
Question 18.22

a. How do B-trees shrink?

b. Suppose there were 2 keys in the root, would the tree shrink?

c. Can we be certain deleting D from DEJK does not violate B-tree requirement: x.n **≥** t-1?

d. Draw the tree (c) after deleting N, t=3.



Case 3a: Borrow.

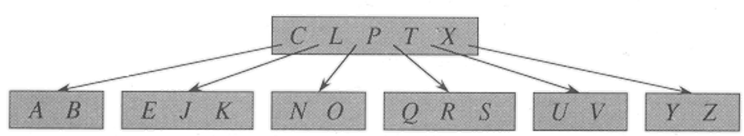
If $x.c_i$ has only t-1 keys but has an immediate sibling with at least t keys

give $x.c_i$ an extra key by moving a key from x down into $x.c_i$

moving a key from $x.c_i$ immediate left or right sibling up into x and

moving the appropriate child pointer from the sibling into $x.c_i$.

k=B x=CLPTX CLPTX.c_1=AB



Move key C (not the child pointers) into AB to give ABC, now has t keys, delete B.

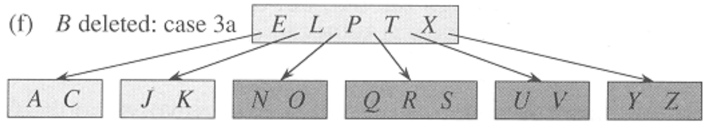
Deleting B from ABC losses one key.

Use other cases described here recursively to adjust B's children appropriately.

Move key E from EJK (with t keys) to x giving ELPTX.

Note that child pointer ELPTX.c_1 remains to AC.
ELPTX.c_1=AC

Move child pointer of EJK to AC.
The old pointer from the left of E is now the pointer from the right of C.
$AC.c_3 = EJK.c_1$



Question 18.23

a. Does it make any difference whether borrow is from left or right sibling of $x.c_i$ in general. That is, do we ever need to change child
b. Why is the last step necessary, in the example: $AC.c_3 = EJK.c_1$?
c. Draw the tree (f) after deleting U.

Case 3a: Borrow.

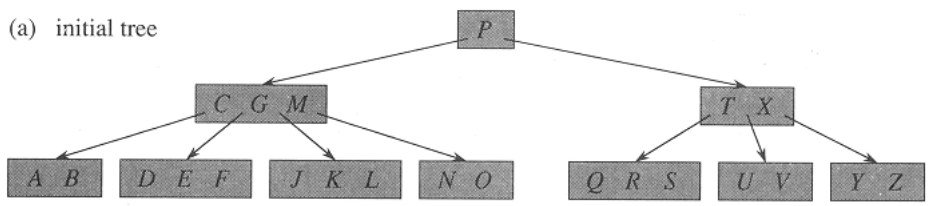
If $x.c_i$ has only t-1 keys but has an immediate sibling with at least t keys

give $x.c_i$ an extra key by moving a key from x down into $x.c_i$

moving a key from $x.c_i$ immediate left or right sibling up into x and

moving the appropriate child pointer from the sibling into $x.c_i$.

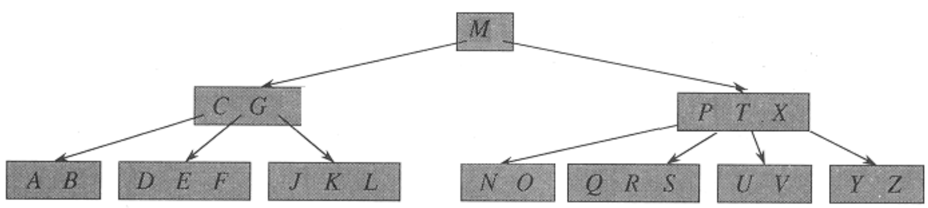
k=R x=P P.c_2=TX



Give $P.c_2$ an extra key by moving a key from P down (key P) into $P.c_2$

moving a key from $P.c_2$ immediate left sibling (CGM) up into P, now M, and

moving the appropriate child pointer from the sibling CG.c_4 to sibling PTX.c_1



(b)

Can now descend into PTX and QRS, to delete R.

Question 18.23 - Draw the tree (b) after deleting L.

Analysis

- Disk accesses for B-tree of height h is O(h).

  By observing that:

  deletion proceeds from root toward leafs,

  read/write at most 3 nodes (e.g. merge) at any level
- CPU time is O(th) = O(t $\log_t$n)

  Question 18.24 What does the t describe?

Delete Implementation

The text does not provide an algorithm for deletion, below is one presented to provide some insight to delete operation.

```
void B_Tree_Delete ( Node root, Item k )
-- pre: root node of B-tree
-- post: if k in B-tree, remove corresponding node

  remove( root, k )

  if root ≠ NIL and root.n = 0 then
    root ← root.c0
    Disk_Write ( root )
```

```
void remove (Node x, Item k)
-- pre: x points to root node of subtree
-- post: if k in subtree, remove corresponding node from subtree

  i ← x.n
  while i ≥ 1  and  k < x.key_i do    -- search for k
      i ← i - 1

  if i ≥ 1 and k ≠ x.key_i then           -- not found, check child
    Disk_Read ( x.c_i)
    remove( x.c_i, k )
  else                                     -- found

    if x.leaf then                       -- at a leaf, delete k
      while i < x.n do                   -- copy over k
        x.key_i ← x.key_i+1
        i ← i + 1
      x.n ← x.n - 1
    else                                  -- not at a leaf, find pointer to correct
      copy_predecessor( x, i)          -- subtree and recurse
      Disk_Read ( x.c_i )
      remove( x.c_i , x.key_i )

  if not x.leaf and x.c_i.n < t - 1 then
      restore( x, i )                    -- child < minimum degree, restore

  Disk_Write (x)
```

```
void copy_predecessor (Node x, int i)
-- pre: x points to non-leaf node with entry at i
-- post: if k in subtree, remove corresponding node from subtree

  leaf ← x.c_i                          -- x.c_i is node to the left of x

  while leaf.c_leaf.n ≠ NIL do    -- Go as far right to a leaf
      leaf ← leaf.c_leaf.n
      Disk_Read ( leaf )

  x.key_i ← leaf.key_leaf.n-1       -- Copy predecessor key

  Disk_Write (x)
```

```
void restore (Node x, int i)
-- pre: x points to non-leaf node with entry at x.c_i has one too few entries
-- post: An entry is taken from elsewhere to restore minimum number
--        of entries in the node to which x.c_i points

  if i = x.n then                          -- rightmost child
     if x.c_i-1.n > t - 1 then
       move_right( x, i - 1)
     else
       combine( x, i )
  else
       if i = 1 then                       -- leftmost child
          if x.c2.n > t - 1 then
            move_left( x, 2 )
          else
            combine( x, 2 )
       else                                -- remaining cases: intermediate branches
          if x.c_i-1.n > t - 1 then
            move_right( x, i-1 )
```

```
            else
                if x.c_{i+1}.n > t - 1 then
                    move_left(x, i+1)
                else
                    combine( x, i )
```

```
void move_left (Node x, int i)
-- pre: x points to node with more than minimum number of entries in child i
--        and one too few in child i-1
-- post: leftmost entry of child i is moved to x, which has sent an entry to child i-1

   left ← x.c_{i-1}
   right ← x.c_i

   left.key_{left.n} ← x.key_{i-1}            -- take parent entry

   left.n ←  left.n + 1

   left.c_{left.n} ← right.c_1

   left.key_{i-1} ← right.key_1             -- add right to parent

   right.n ← right.n - 1

   for j ← 1 to right.n-1 do            -- move right entries to fill hole
         right.key_j ← right.key_{j+1}
         right.c_j ← right.c_{j+1}

    right.c_{right.n} ← right.c_{right.n+1}

    Disk_Write( left )
    Disk_Write( right )
```

```
void move_right (Node x, int i)
-- pre: x points to node with more than minimum number of entries in child i
--        and one too few in child i+1
-- post: rightmost entry of child i is moved to x, which has sent an entry to child i+1

   left ← x.c_i
   right ← x.c_{i+1}

   right.c_{right.n+1} ← right.c_{right.n}

   for j ← right.n downto 1 do          -- make room for new entry
         right.key_j ← right.key_{j-1}
         right.c_j ← right.c_{j-1}

   right.n ← right.n + 1

   right.c_1 ← left.c_{left.n}

    left.n ←  left.n - 1

   x.key_i ← left.key_{left.n}

    Disk_Write( x )
    Disk_Write( left )
    Disk_Write( right )
```

```
void combine (Node x, int i)
-- pre: x points to a node with child i and i-1 with too few entries to move
-- post: nodes at i and i-1 have been combined into one node

   left ← x.c_{i-1}
   right ← x.c_i

   left.key_{left.n} ← x.key_{i-1}

    left.n ← left.n + 1

   left.c_{left.n} ← right.c_1

   for j ← 1 to right.n-1 do
         left.key_{left.n} ← right.key_j
```

```
      left.n ←  left.n + 1
      left.c_{left.n} ← right.c_{j+1}

x.n ← x.n - 1

for j ← i-1 to x.n-1 do
      x.key_j ← x.key_{j+1}
      x.c_{i+1} ← x.c_{i+2}

 right = NIL

 Disk_Write( x )
 Disk_Write( left )
 Disk_Write( right )
```