Red-black trees, augmenting data structures, B-trees

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Oct 10 & 12, 2017

Red-black trees (Chap 13)

A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute *color*, which is either red or black.

All leaves are empty (nil) and colored black.

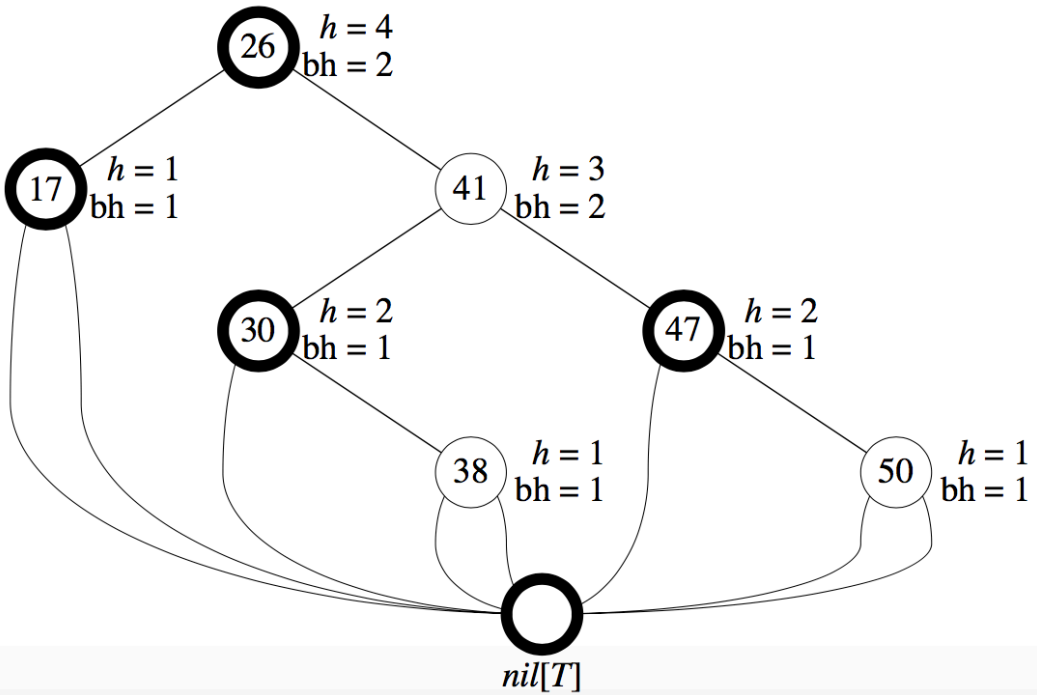
- We use a single sentinel, $nil[T]$, for all the leaves of red-black tree T.
- $color[nil[T]]$ is black.
- The root's parent is also $nil[T]$.

All other attributes of binary search trees are inherited by red-black trees (*key*, *left*, *right*, and p). We don't care about the key in $nil[T]$.

Red-black trees properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($nil[T]$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Example:



Height of red-black trees

- ***Height of a node*** is the number of edges in a longest path to a leaf.
- ***Black-height*** of a node x: $\mathrm{bh}(x)$ is the number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x. By property 5, black-height is well defined.

[Now label the example tree with height h and bh *values.]*

Claim

Any node with height h has black-height $\geq h/2$.

Proof By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. ■ (claim)

Height of red-black trees

Claim

The subtree rooted at any node x contains $\geq 2^{\mathrm{bh}(x)} - 1$ internal nodes.

Proof By induction on height of x.

Basis: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow \mathrm{bh}(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.

Inductive step: Let the height of x be h and $\mathrm{bh}(x) = b$. Any child of x has height $h - 1$ and black-height either b (if the child is red) or $b - 1$ (if the child is black). By the inductive hypothesis, each child has $\geq 2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes. (The $+1$ is for x itself.) ■ (claim)

Lemma

A red-black tree with n internal nodes has height $\leq 2 \lg(n + 1)$.

Proof Let h and b be the height and black-height of the root, respectively. By the above two claims,

$$n \geq 2^b - 1 \geq 2^{h/2} - 1 .$$

Adding 1 to both sides and then taking logs gives $\lg(n + 1) \geq h/2$, which implies that $h \leq 2 \lg(n + 1)$. ■ (theorem)

Operations on red-black trees

The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(\text{height})$ time. Thus, they take $O(\lg n)$ time on red-black trees.

Insertion and deletion are not so easy.

If we insert, what color to make the new node?

- Red? Might violate property 4.
- Black? Might violate property 5.

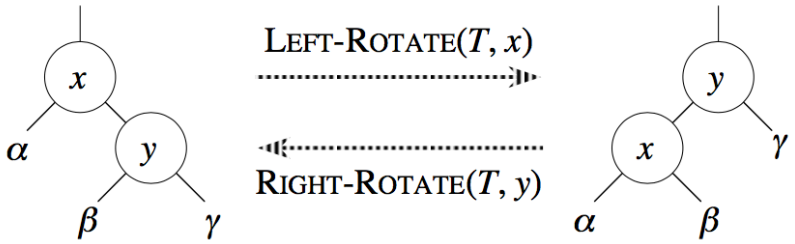
If we delete, thus removing a node, what color was the node that was removed?

- Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
- Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

Rotation

- The basic tree-restructuring operation.
- Needed to maintain red-black trees as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)
- Won't upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.

Left rotation



LEFT-ROTATE(T, x)

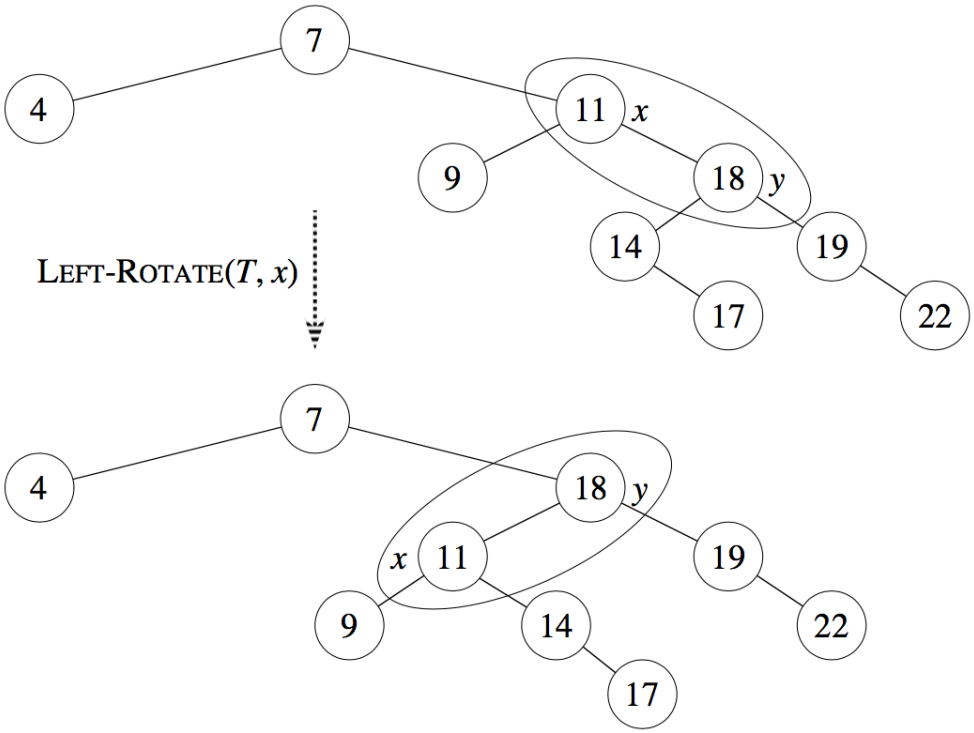
$y \leftarrow right[x]$ ▷ Set y.
$right[x] \leftarrow left[y]$ ▷ Turn y's left subtree into x's right subtree.
if $left[y] \neq nil[T]$
  then $p[left[y]] \leftarrow x$
$p[y] \leftarrow p[x]$ ▷ Link x's parent to y.
if $p[x] = nil[T]$
  then $root[T] \leftarrow y$
  else if $x = left[p[x]]$
    then $left[p[x]] \leftarrow y$
    else $right[p[x]] \leftarrow y$
$left[y] \leftarrow x$ ▷ Put x on y's left.
$p[x] \leftarrow y$

The pseudocode for LEFT-ROTATE assumes that

- $right[x] \neq nil[T]$, and
- root's parent is $nil[T]$.

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

Example



- Before rotation: keys of x's left subtree $\leq 11 \leq$ keys of y's left subtree $\leq 18 \leq$ keys of y's right subtree.
- Rotation makes y's left subtree into x's right subtree.
- After rotation: keys of x's left subtree $\leq 11 \leq$ keys of x's right subtree $\leq 18 \leq$ keys of y's right subtree.

Time: $O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.

RB insertion

RB-INSERT(T, z)

```
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
      do y ← x
         if key[z] < key[x]
           then x ← left[x]
           else  x ← right[x]
p[z] ← y
if y = nil[T]
   then root[T] ← z
   else  if key[z] < key[y]
           then left[y] ← z
           else  right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
```

RB-INSERT-FIXUP(T, z)

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

RB insertion fixup

Which property might be violated?

1. OK.
2. If z is the root, then there's a violation. Otherwise, OK.
3. OK.
4. If $p[z]$ is red, there's a violation: both z and $p[z]$ are red.
5. OK.

Remove the violation by calling RB-INSERT-FIXUP:

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
          then y ← right[p[p[z]]]
               if color[y] = RED
                  then color[p[z]] ← BLACK                    ▷ Case 1
                       color[y] ← BLACK                       ▷ Case 1
                       color[p[p[z]]] ← RED                   ▷ Case 1
                       z ← p[p[z]]                            ▷ Case 1
                  else if z = right[p[z]]
                          then z ← p[z]                       ▷ Case 2
                               LEFT-ROTATE(T, z)              ▷ Case 2
                       color[p[z]] ← BLACK                    ▷ Case 3
                       color[p[p[z]]] ← RED                   ▷ Case 3
                       RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
          else (same as then clause
                     with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

Correctness

Loop invariant:

At the start of each iteration of the **while** loop,

a. z is red.

b. There is at most one red-black violation:

- Property 2: z is a red root, or
- Property 4: z and $p[z]$ are both red.

[The book has a third part of the loop invariant, but we omit it for lecture.]

Initialization: We've already seen why the loop invariant holds initially.

Termination: The loop terminates because $p[z]$ is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.

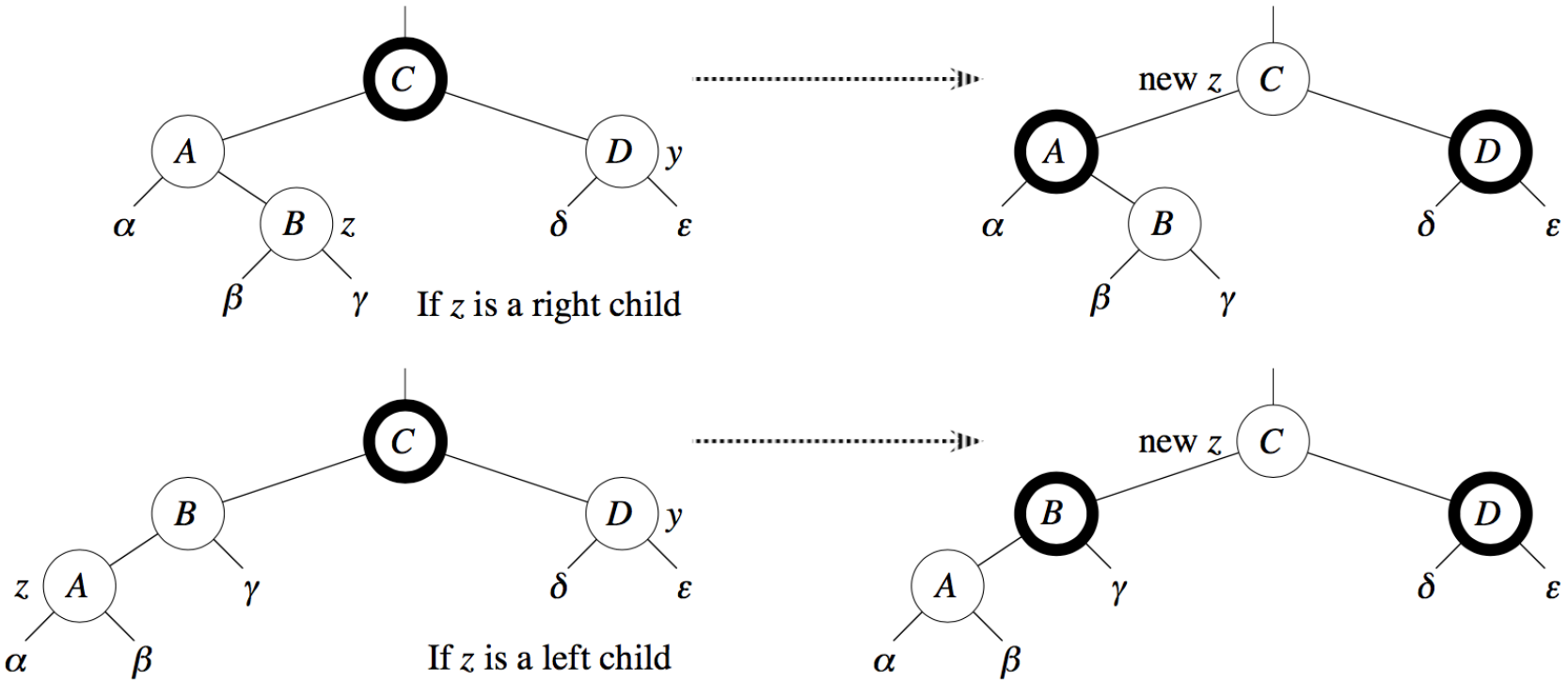
Maintenance: We drop out when z is the root (since then $p[z]$ is the sentinel $nil[T]$, which is black). When we start the loop body, the only violation is of property 4.

There are 6 cases, 3 of which are symmetric to the other 3. The cases are not mutually exclusive. We'll consider cases in which $p[z]$ is a left child.

Let y be z's uncle ($p[z]$'s sibling).

Case 1

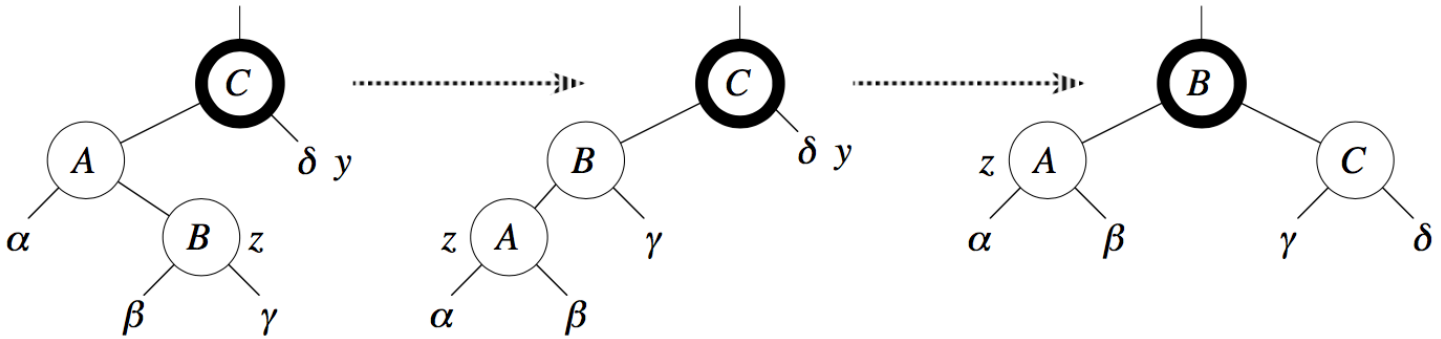
Case 1: y is red



- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black $\Rightarrow$ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Rightarrow$ restores property 5.
- The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).

Case 2 and Case 3

Case 2: y is black, z is a right child



Case 2 Case 3

- Left rotate around $p[z] \Rightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Case 3: y is black, z is a left child

- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

Time analysis

Analysis

$O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

Within RB-INSERT-FIXUP:

- Each iteration takes $O(1)$ time.
- Each iteration is either the last one or it moves z up 2 levels.
- $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

RB deletion

RB-DELETE(T, z)

if $left[z] = nil[T]$ or $right[z] = nil[T]$
  then $y \leftarrow z$
  else $y \leftarrow$ TREE-SUCCESSOR(z)
if $left[y] \neq nil[T]$
  then $x \leftarrow left[y]$
  else $x \leftarrow right[y]$
$p[x] \leftarrow p[y]$
if $p[y] = nil[T]$
  then $root[T] \leftarrow x$
  else if $y = left[p[y]]$
    then $left[p[y]] \leftarrow x$
    else $right[p[y]] \leftarrow x$
if $y \neq z$
  then $key[z] \leftarrow key[y]$
    copy y's satellite data into z
if $color[y] =$ BLACK
  then RB-DELETE-FIXUP(T, x)
return y

- y is the node that was actually spliced out.
- x is either
  - y's sole non-sentinel child before y was spliced out, or
  - the sentinel, if y had no children.

RB deletion

If y is black, we could have violations of red-black properties:

1. OK.
2. If y is the root and x is red, then the root has become red.
3. OK.
4. Violation if $p[y]$ and x are both red.
5. Any path containing y now has 1 fewer black node.

- Correct by giving x an "extra black."
- Add 1 to count of black nodes on paths containing x.
- Now property 5 is OK, but property 1 is not.
- x is either ***doubly black*** (if $color[x] = \text{BLACK}$) or ***red & black*** (if $color[x] = \text{RED}$).
- The attribute $color[x]$ is still either RED or BLACK. No new values for $color$ attribute.
- In other words, the extra blackness on a node is by virtue of x pointing to the node.

RB deletion fixup

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w ← right[p[x]]
               if color[w] = RED
                 then color[w] ← BLACK                          ▷ Case 1
                      color[p[x]] ← RED                         ▷ Case 1
                      LEFT-ROTATE(T, p[x])                      ▷ Case 1
                      w ← right[p[x]]                           ▷ Case 1
               if color[left[w]] = BLACK and color[right[w]] = BLACK
                 then color[w] ← RED                            ▷ Case 2
                      x ← p[x]                                  ▷ Case 2
                 else if color[right[w]] = BLACK
                        then color[left[w]] ← BLACK             ▷ Case 3
                             color[w] ← RED                     ▷ Case 3
                             RIGHT-ROTATE(T, w)                 ▷ Case 3
                             w ← right[p[x]]                    ▷ Case 3
                      color[w] ← color[p[x]]                    ▷ Case 4
                      color[p[x]] ← BLACK                       ▷ Case 4
                      color[right[w]] ← BLACK                   ▷ Case 4
                      LEFT-ROTATE(T, p[x])                      ▷ Case 4
                      x ← root[T]                               ▷ Case 4
          else (same as then clause with "right" and "left" exchanged)
color[x] ← BLACK
```

RB deletion fixup

Idea: Move the extra black up the tree until

- x points to a red & black node $\Rightarrow$ turn it into a black node,
- x points to the root $\Rightarrow$ just remove the extra black, or
- we can do certain rotations and recolorings and finish.

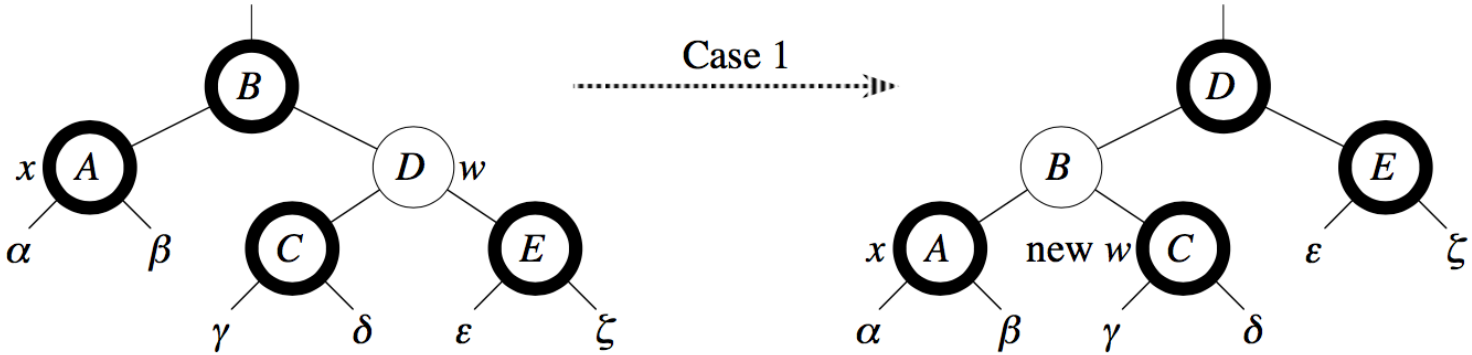
Within the **while** loop:

- x always points to a nonroot doubly black node.
- w is x's sibling.
- w cannot be $nil[T]$, since that would violate property 5 at $p[x]$.

There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which x is a left child.

Case 1

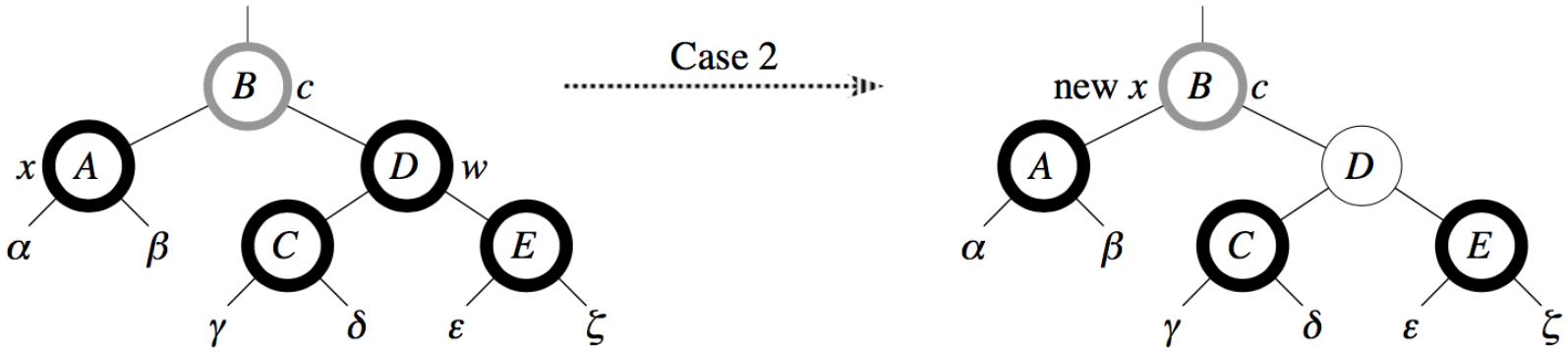
Case 1: w is red



- w must have black children.
- Make w black and $p[x]$ red.
- Then left rotate on $p[x]$.
- New sibling of x was a child of w before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

Case 2

Case 2: w is black and both of w's children are black

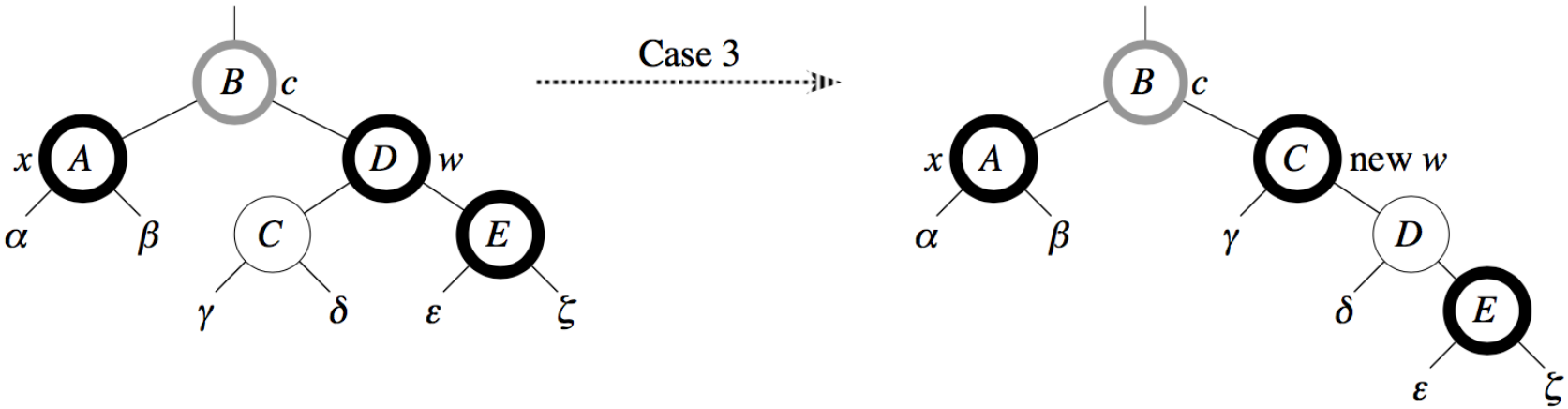


[Node with gray outline is of unknown color, denoted by c.]

- Take 1 black off x ($\Rightarrow$ singly black) and off w ($\Rightarrow$ red).
- Move that black to $p[x]$.
- Do the next iteration with $p[x]$ as the new x.
- If entered this case from case 1, then $p[x]$ was red $\Rightarrow$ new x is red & black $\Rightarrow$ color attribute of new x is RED $\Rightarrow$ loop terminates. Then new x is made black in the last line.

Case 3

Case 3: w is black, w's left child is red, and w's right child is black

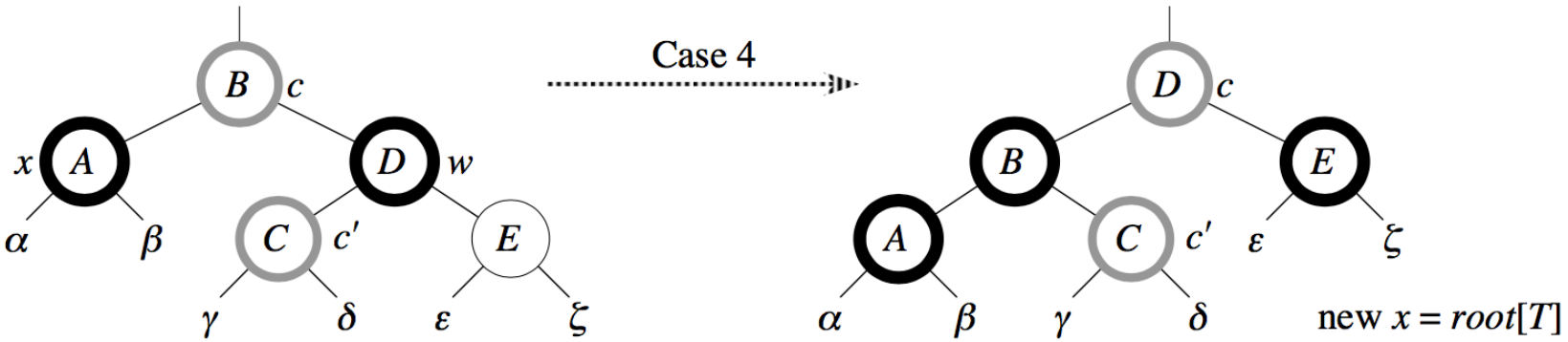


- Make w red and w's left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Case 4

Case 4: w is black, w's left child is black, and w's right child is red

or red



[Now there are two nodes of unknown colors, denoted by c and c'.]

- Make w be $p[x]$'s color (c).
- Make $p[x]$ black and w's right child black.
- Then left rotate on $p[x]$.
- Remove the extra black on x ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

Time analysis

Analysis

$O(\lg n)$ time to get through RB-DELETE up to the call of RB-DELETE-FIXUP.

Within RB-DELETE-FIXUP:

- Case 2 is the only case in which more iterations occur.
  - x moves up 1 level.
  - Hence, $O(\lg n)$ iterations.
- Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
- Hence, $O(\lg n)$ time.

Augument data structures (Chap 14)

We'll be looking at methods for *designing* algorithms. In some cases, the design will be intermixed with analysis. In other cases, the analysis is easy, and it's the design that's harder.

Augmenting data structures

- It's unusual to have to design an all-new data structure from scratch.
- It's more common to take a data structure that you know and store additional information in it.
- With the new information, the data structure can support new operations.
- But...you have to figure out how to *correctly maintain* the new information *without loss of efficiency*.

Aug. Red-Black trees

We want to support the usual dynamic-set operations from R-B trees, plus:

- OS-SELECT(x, i): return pointer to node containing the ith smallest key of the subtree rooted at x.
- OS-RANK(T, x): return the rank of x in the linear order determined by an inorder walk of T.

Augment by storing in each node x:

$size[x] = \#$ of nodes in subtree rooted at x .

- Includes x itself.
- Does not include leaves (sentinels).

Define for sentinel $size[nil[T]] = 0$.

Then $size[x] = size[left[x]] + size[right[x]] + 1$.

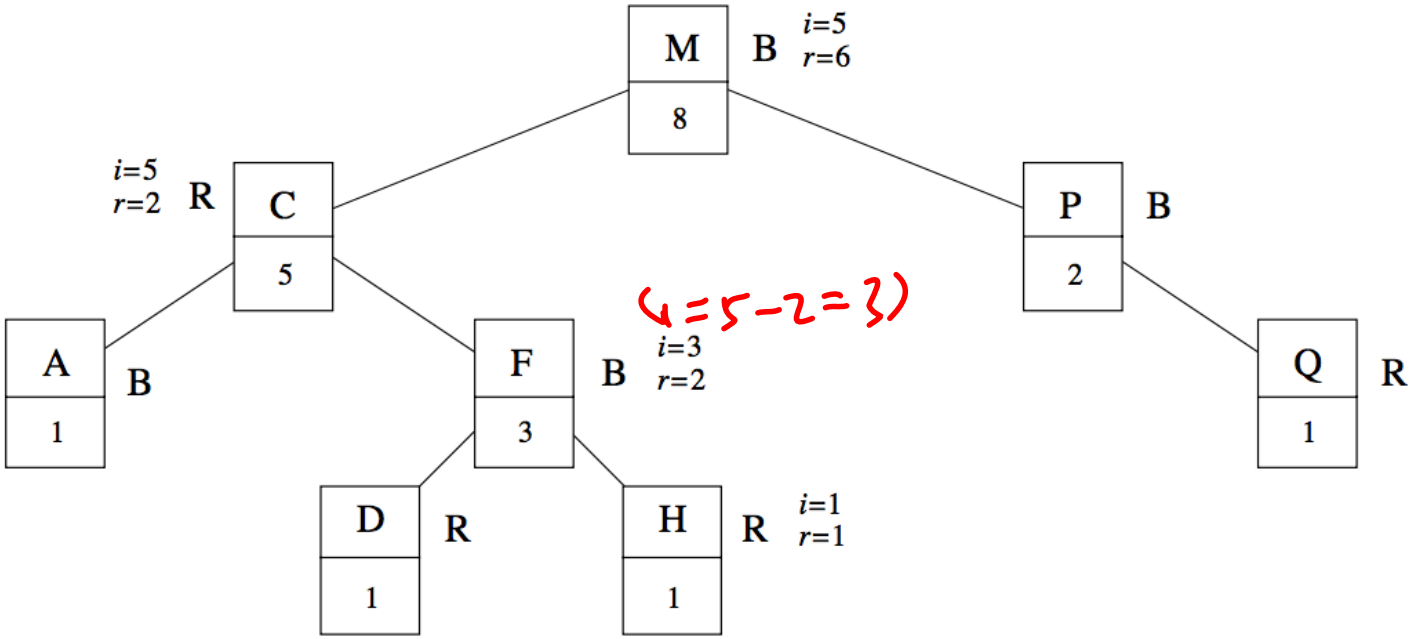
OS-Select

```
OS-SELECT(x, i)
r ← size[left[x]]+1
if i = r
   then return x
elseif i < r
   then return OS-SELECT(left[x], i)
else return OS-SELECT(right[x], i − r)
```

Initial call: OS-SELECT(*root*[*T*], *i*)

rank of x

Example



*[**Example above:** Ignore colors, but legal coloring shown with "R" and "B" notations. Values of i and r are for the example below.]*

Note: OK for keys to not be distinct. Rank is defined with respect to position in inorder walk. So if we changed D to C, rank of original C is 2, rank of D changed to C is 3.

Example: Try OS-SELECT($root[T]$, 5). *[Values shown in figure above. Returns node whose key is H.]*

Analysis

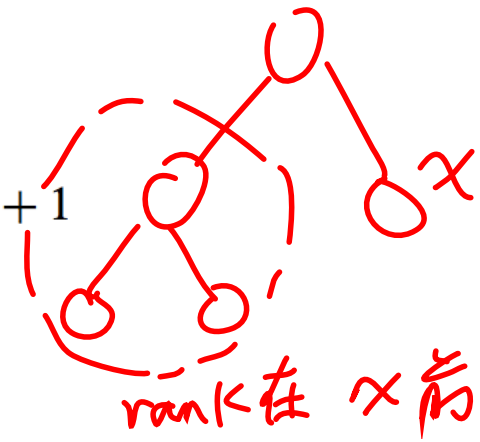
Correctness: r = rank of x within subtree rooted at x.

- If $i = r$, then we want x.
- If $i < r$, then ith smallest element is in x's left subtree, and we want the ith smallest element in the subtree.
- If $i > r$, then ith smallest element is in x's right subtree, but subtract off the r elements in x's subtree that precede those in x's right subtree.
- Like the randomized SELECT algorithm!

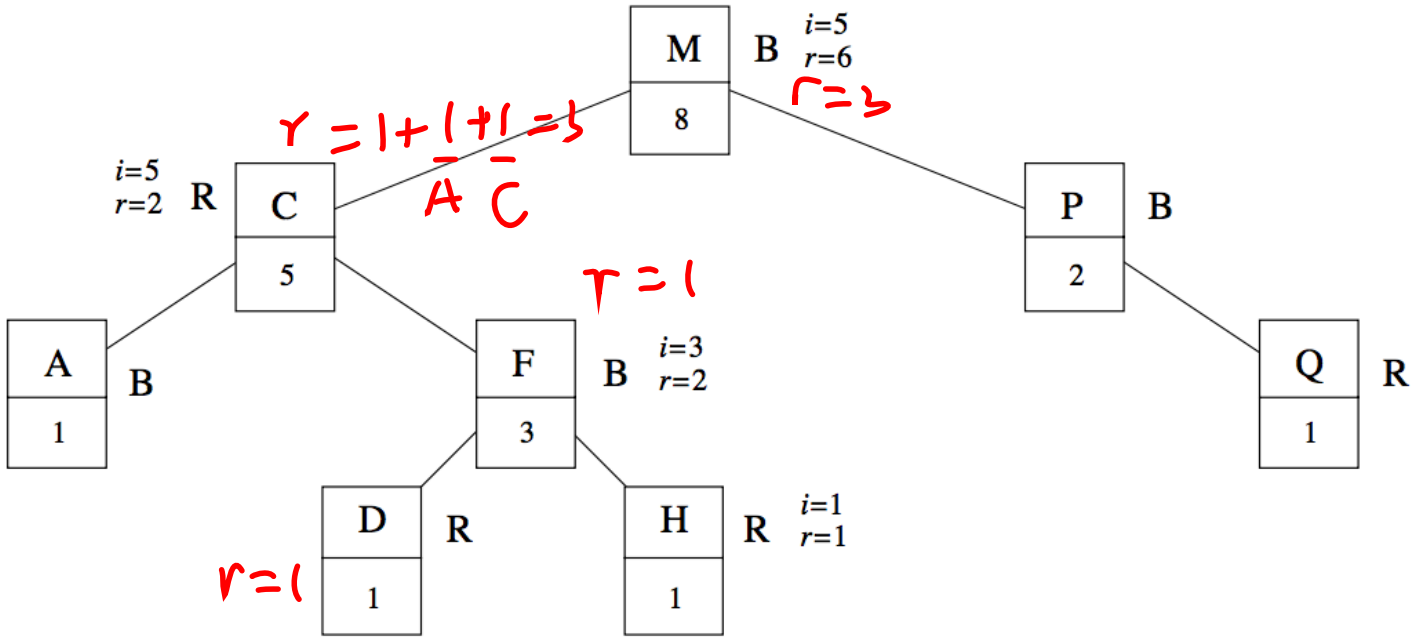
Analysis: Each recursive call goes down one level. Since R-B tree has $O(\lg n)$ levels, have $O(\lg n)$ calls $\Rightarrow$ $O(\lg n)$ time.

OS-Rank

```
OS-RANK(T, x)
r ← size[left[x]] + 1
y ← x
while y ≠ root[T]
    do if y = right[p[y]]
          then r ← r + size[left[p[y]]] + 1
       y ← p[y]
return r
```



Example



Demo: Node D.

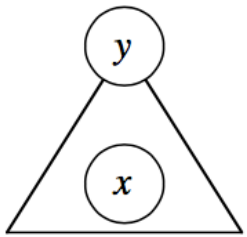
Analysis

Loop invariant: At start of each iteration of **while** loop, r = rank of $key[x]$ in subtree rooted at y.

Initialization: Initially, r = rank of $key[x]$ in subtree rooted at x, and $y = x$.

Termination: Loop terminates when $y = root[T] \Rightarrow$ subtree rooted at y is entire tree. Therefore, r = rank of $key[x]$ in entire tree.

Maintenance: At end of each iteration, set $y \leftarrow p[y]$. So, show that if r = rank of $key[x]$ in subtree rooted at y at start of loop body, then r = rank of $key[x]$ in subtree rooted at $p[y]$ at end of loop body.

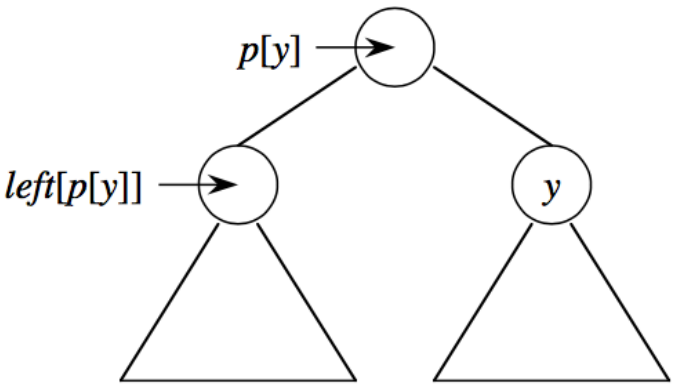


[r = # of nodes in subtree rooted at y preceding x in inorder walk]

Analysis

Must add nodes in y's sibling's subtree.

- If y is a left child, its sibling's subtree follows all nodes in y's subtree $\Rightarrow$ don't change r.
- If y is a right child, all nodes in y's sibling's subtree precede all nodes in y's subtree $\Rightarrow$ add size of y's sibling's subtree, plus 1 for $p[y]$, into r.



Analysis: y goes up one level in each iteration $\Rightarrow$ $O(\lg n)$ time.

Maintain subtree sizes

Maintaining subtree sizes

- Need to maintain $size[x]$ fields during insert and delete operations.
- Need to maintain them efficiently. Otherwise, might have to recompute them all, at a cost of $\Omega(n)$.

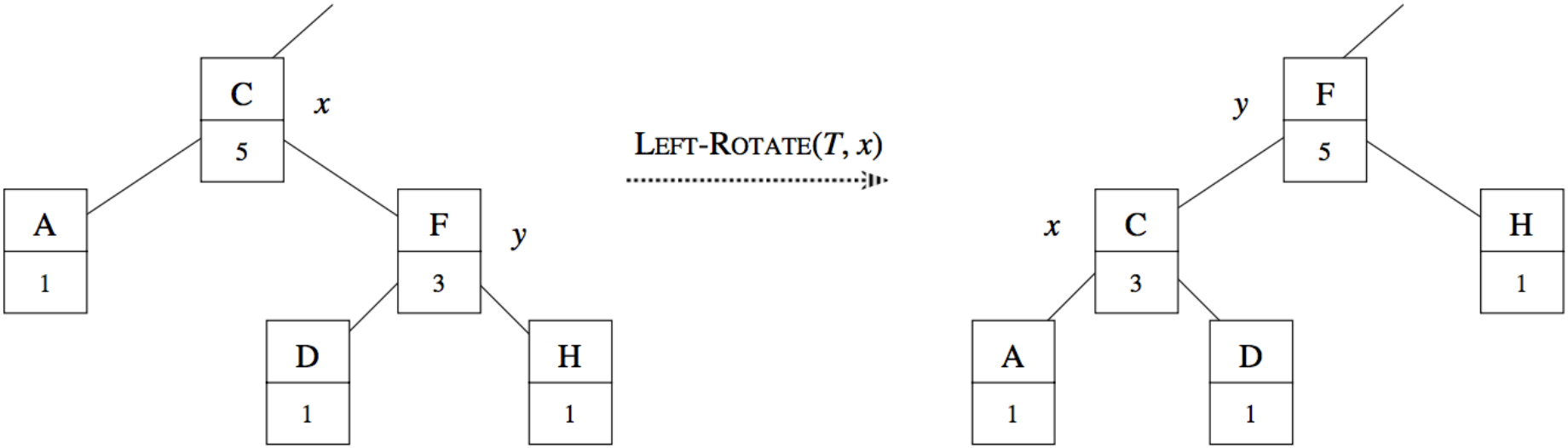
Will see how to maintain without increasing $O(\lg n)$ time for insert and delete.

For insertion

Insert:

- During pass downward, we know that the new node will be a descendant of each node we visit, and only of these nodes. Therefore, increment *size* field of each node visited.
- Then there's the fixup pass:
  - Goes up the tree.
  - Changes colors $O(\lg n)$ times.
  - Performs ≤ 2 rotations.
- Color changes don't affect subtree sizes.
- Rotations do!
- But we can determine new sizes based on old sizes and sizes of children.

For insertion



$$size[y] \leftarrow size[x]$$
$$size[x] \leftarrow size[left[x]] + size[right[x]] + 1$$

- Similar for right rotation.
- Therefore, can update in $O(1)$ time per rotation $\Rightarrow$ $O(1)$ time spent updating *size* fields during fixup.
- Therefore, $O(\lg n)$ to insert.

For deletion

Delete: Also 2 phases:

1. Splice out some node y.
2. Fixup.

After splicing out y, traverse a path $y \rightarrow root$, decrementing *size* in each node on path. $O(\lg n)$ time.

During fixup, like insertion, only color changes and rotations.

- ≤ 3 rotations $\Rightarrow O(1)$ time spent updating *size* fields during fixup.
- Therefore, $O(\lg n)$ to delete.

Done!

$O(\lg n) + O(\lg n) + O(1)$

Summary: how do we aug. data structures

1. Choose an underlying data structure.
2. Determine additional information to maintain.
3. Verify that we can maintain additional information for existing data structure operations. maintain running time
4. Develop new operations.

Don't need to do these steps in strict order! Usually do a little of each, in parallel.

Don't need to do these steps in strict order! Usually do a little of each, in parallel.

How did we do them for OS trees?

1. R-B tree.
2. $size[x]$.
3. Showed how to maintain *size* during insert and delete.
4. Developed OS-SELECT and OS-RANK.

R-B trees are amenable

Red-black trees are particularly amenable to augmentation.

Theorem

Augment a R-B tree with field f, where $f[x]$ depends only on information in x, $left[x]$, and $right[x]$ (including $f[left[x]]$ and $f[right[x]]$). Then can maintain values of f in all nodes during insert and delete without affecting $O(\lg n)$ performance.

Proof Since $f[x]$ depends only on x and its children, when we alter information in x, changes propagate only upward (to $p[x]$, $p[p[x]]$, ..., $root$).

Height = $O(\lg n) \Rightarrow O(\lg n)$ updates, at $O(1)$ each.

R-B trees are amenable

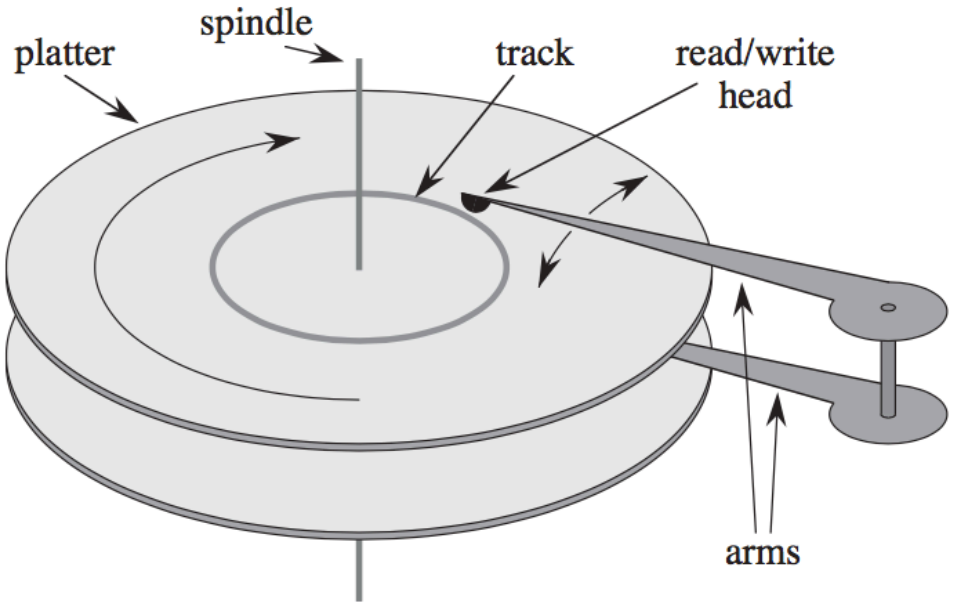
Insertion: Insert a node as child of existing node. Even if can't update f on way down, can go up from inserted node to update f. During fixup, only changes come from color changes (no effect on f) and rotations. Each rotation affects f of ≤ 3 nodes (x,y, and parent), and can recompute each in $O(1)$ time. Then, if necessary, propagate changes up the tree. Therefore, $O(\lg n)$ time per rotation. Since ≤ 2 rotations, $O(\lg n)$ time to update f during fixup.

Delete: Same idea. After splicing out a node, go up from there to update f. Fixup has ≤ 3 rotations. $O(\lg n)$ per rotation $\Rightarrow O(\lg n)$ to update f during fixup.

■ (theorem)

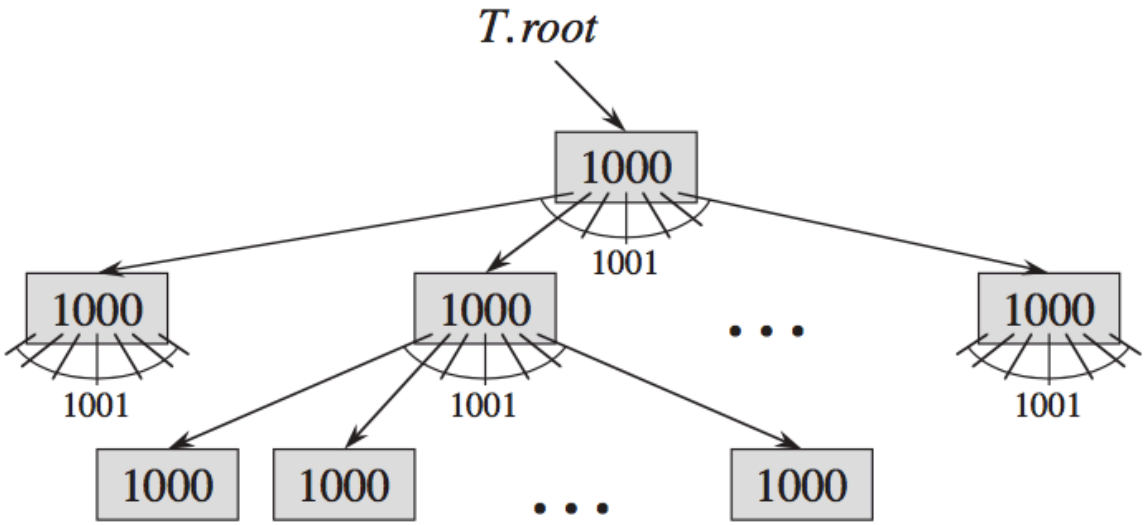
For some attributes, can get away with $O(1)$ per rotation. Example: *size* field.

B trees (Chap 18)



- B tree: balanced tree working well for secondary storages, e.g, disks
- Time consuming: disk read
- Basic unit for read: page, fixed length of bits
- Try to make each page a node to reduce disk reads

B trees example



1 node,
1000 keys

1001 nodes,
1,001,000 keys

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

B trees definition

1. Every node x has the following attributes:

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

   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,

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

2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

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

B trees definition

4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ 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 may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[2]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

B trees 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 of a B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys. Thus, T, whose height is h, has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h it has at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. Thus, the number n of keys satisfies the inequality

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

By simple algebra, we get $t^h \leq (n+1)/2$. Taking base-t logarithms of both sides proves the theorem. ■

B trees search

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3       i = i + 1
4  if i ≤ x.n and k == x.key_i
5       return (x, i)
6  elseif x.leaf
7       return NIL
8  else DISK-READ(x.c_i)
9       return B-TREE-SEARCH(x.c_i, k)
```

Running time: $O(th) = O(t \log_t n)$

B trees search example

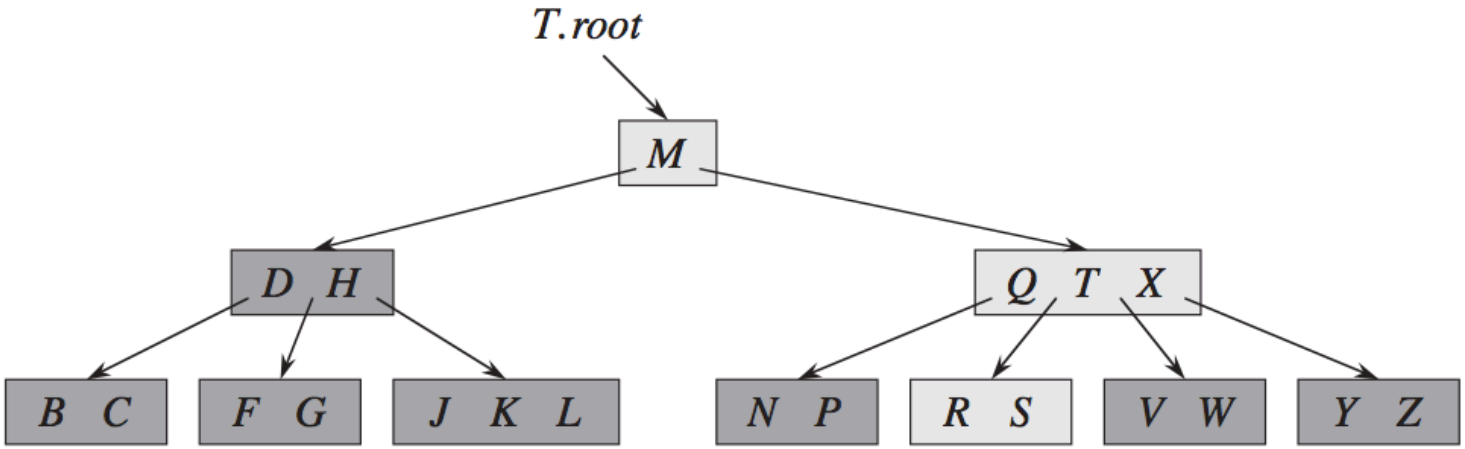


Figure 18.1 illustrates the operation of B-TREE-SEARCH. The procedure examines the lightly shaded nodes during a search for the key R.

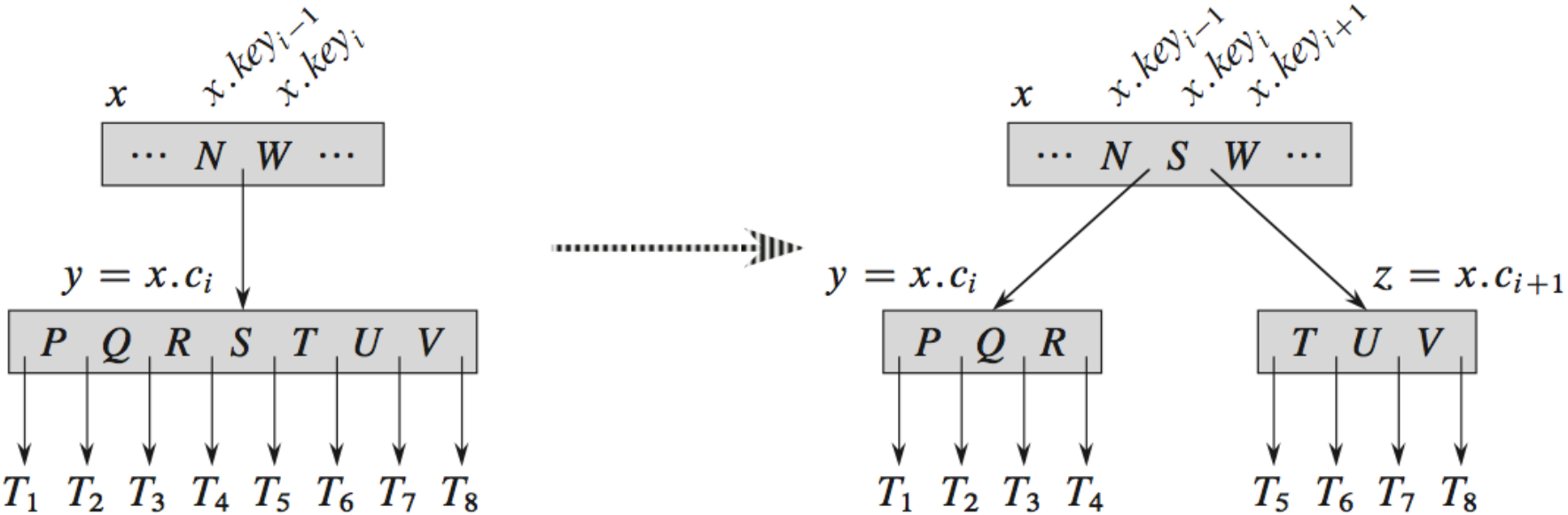
Create a B tree

B-TREE-CREATE(T)

1 x = ALLOCATE-NODE()
2 $x.leaf$ = TRUE
3 $x.n = 0$
4 DISK-WRITE(x)
5 $T.root = x$

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

Split child


x
x.key_{i−1}
x.key_i
··· N W ···
y = x.c_i
P Q R S T U V
T1 T2 T3 T4 T5 T6 T7 T8
x
x.key_{i−1}
x.key_i
x.key_{i+1}
··· N S W ···
y = x.c_i
z = x.c_{i+1}
P Q R
T U V
T1 T2 T3 T4
T5 T6 T7 T8

Split child: $O(1)$

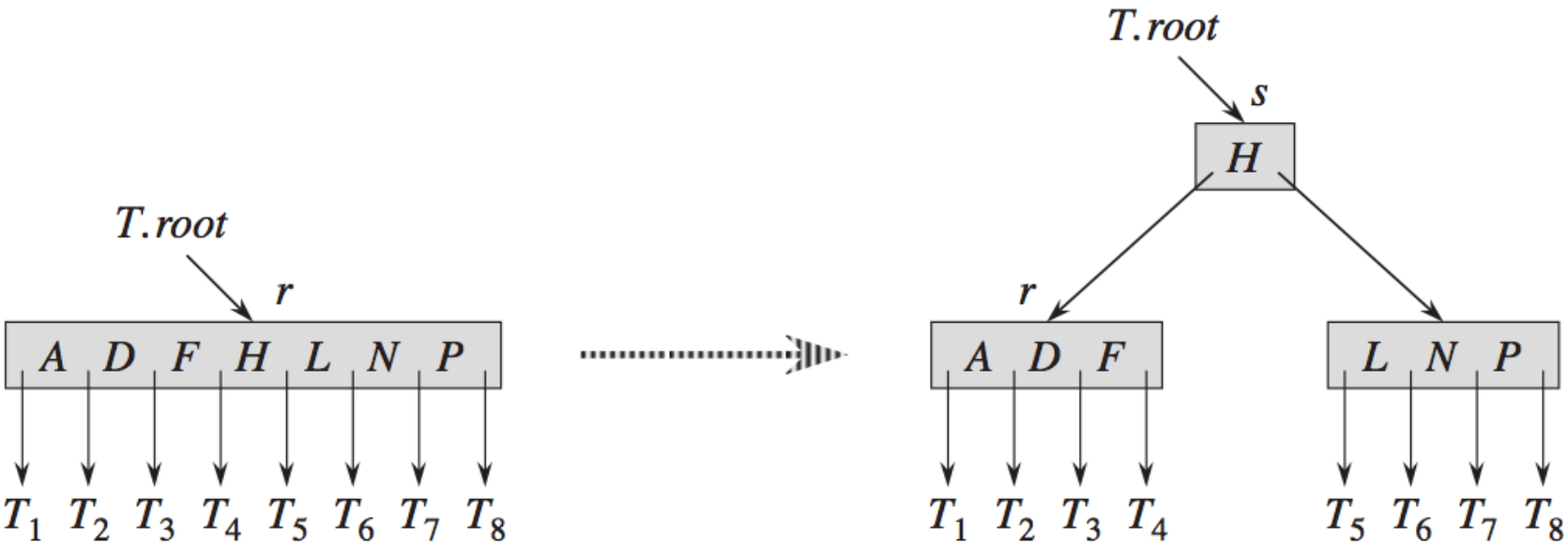
B-TREE-SPLIT-CHILD(x, i)

```
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t
 9          z.c_j = y.c_{j+t}
10  y.n = t − 1
11  for j = x.n + 1 downto i + 1
12      x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15      x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

B trees insert

```
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)
```

Split root

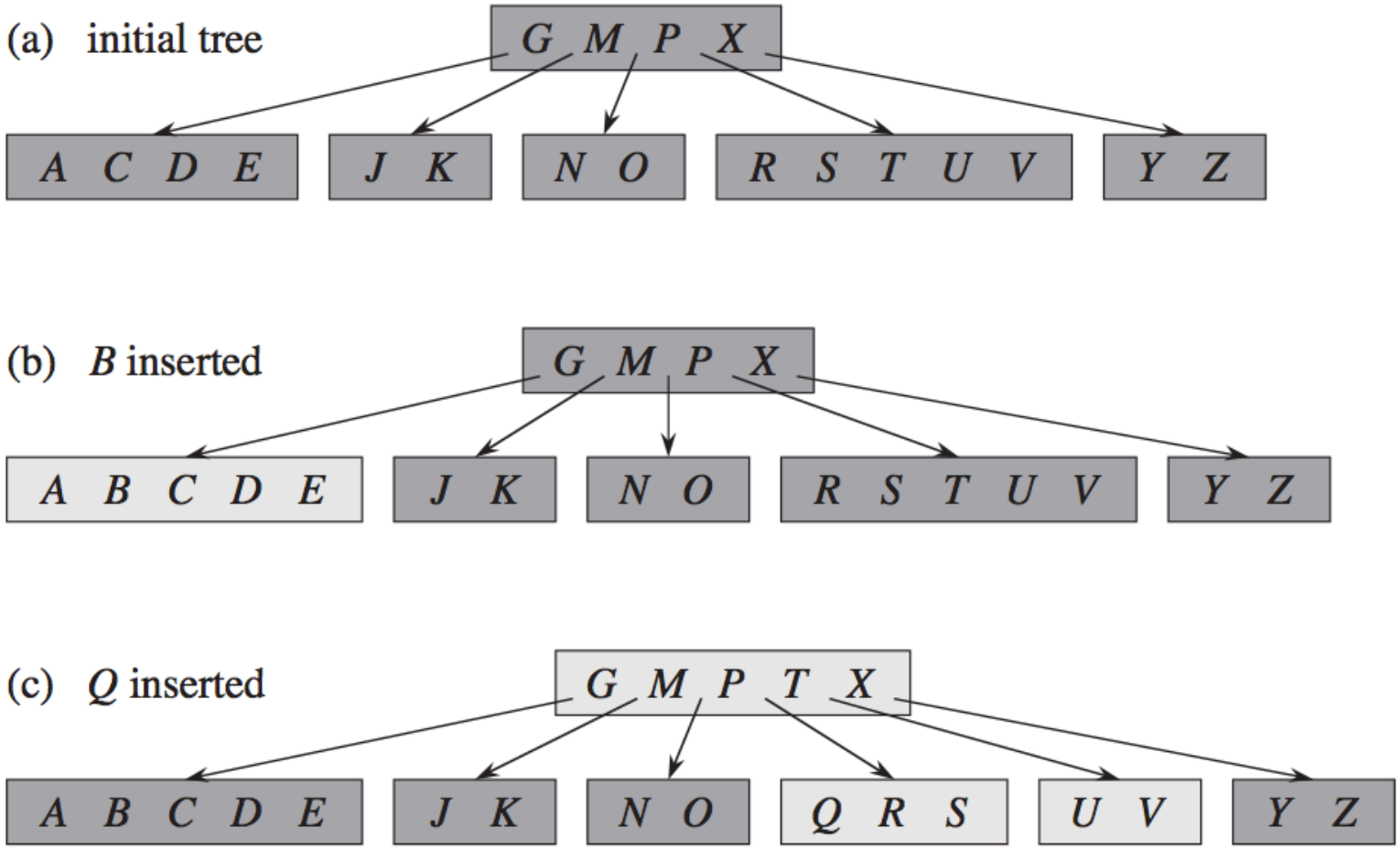

T.root
r
A D F H L N P
T1 T2 T3 T4 T5 T6 T7 T8
T.root
s
H
r
A D F
L N P
T1 T2 T3 T4
T5 T6 T7 T8

B trees insert nonfull

```
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)
```

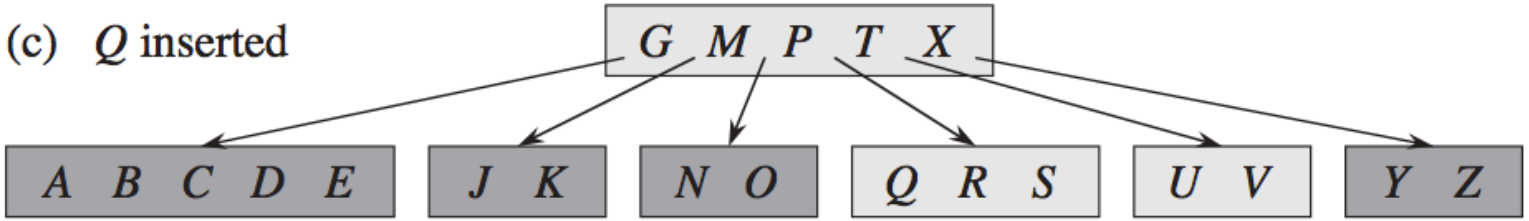
Running time: $O(th) = O(t \log_t n)$

B trees insertion examples

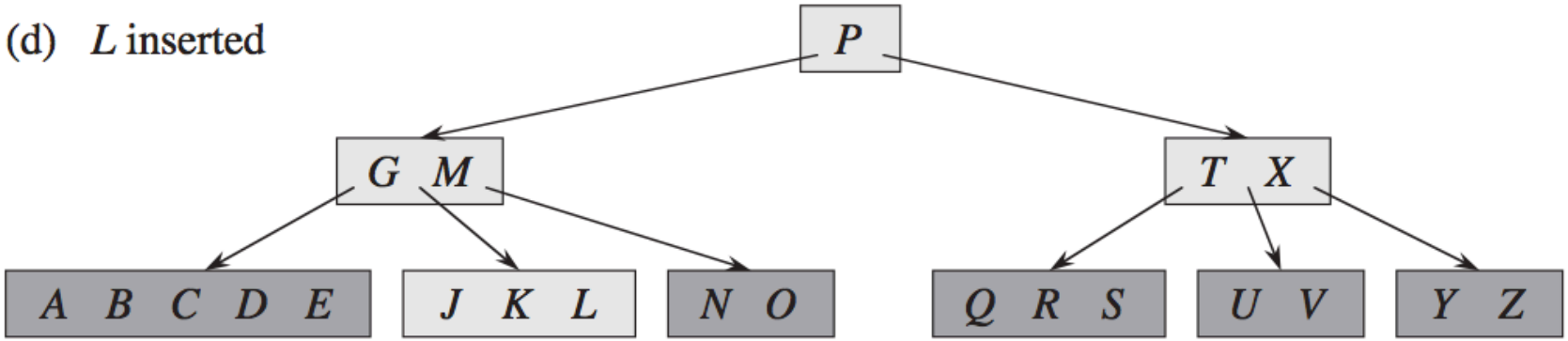


B trees insertion examples

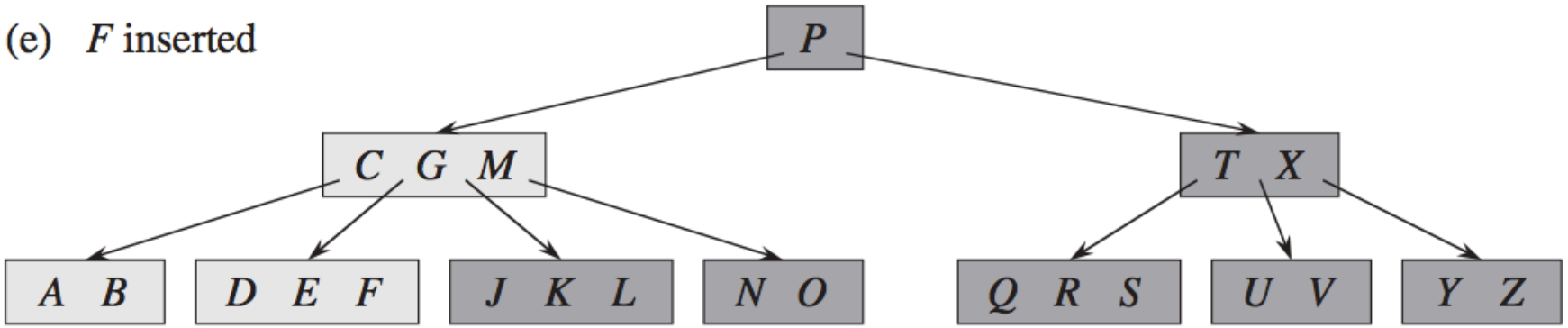
(c) Q inserted



(d) L inserted



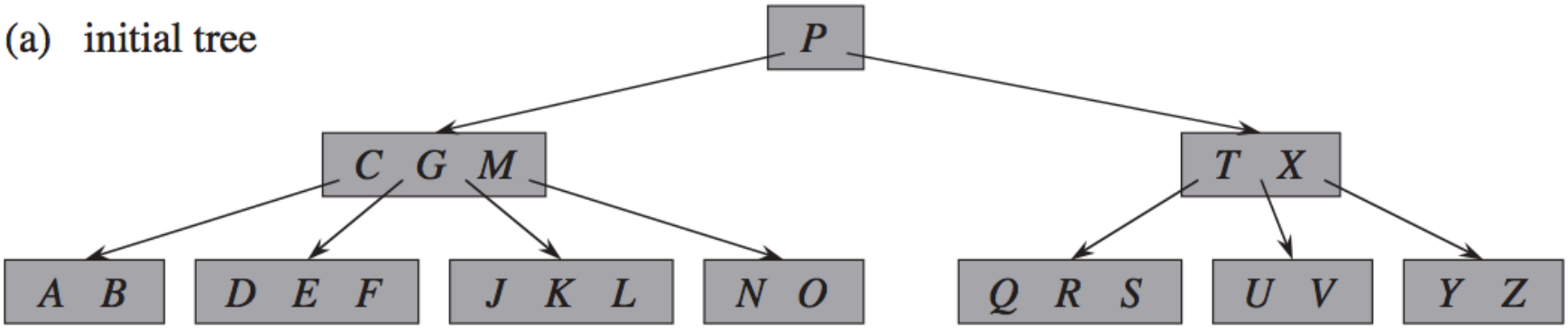
(e) F inserted



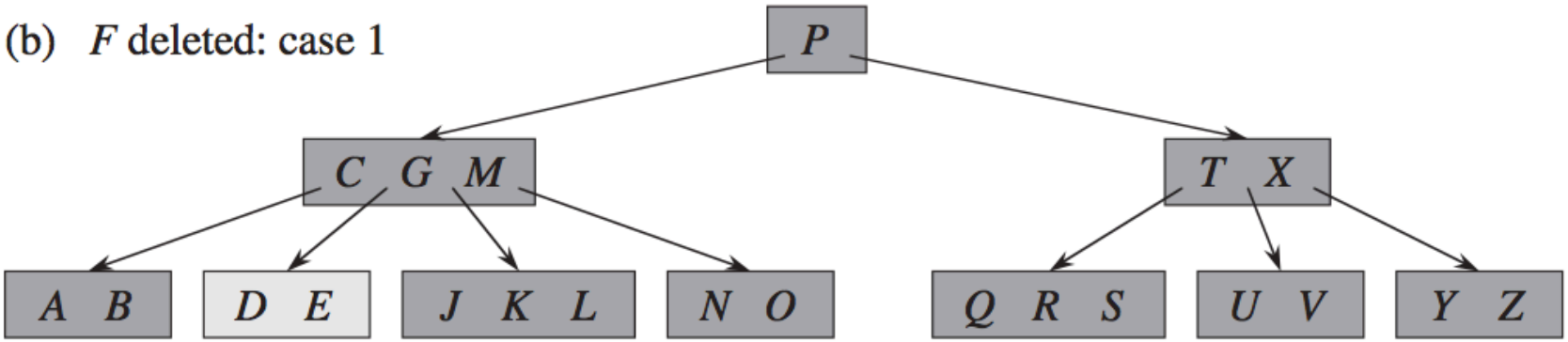
B trees deletion

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

(a) initial tree



(b) F deleted: case 1



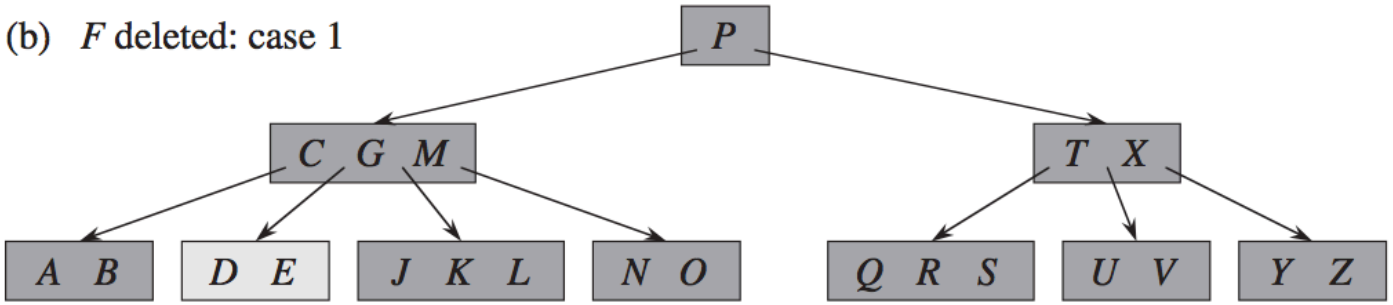
B trees deletion

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

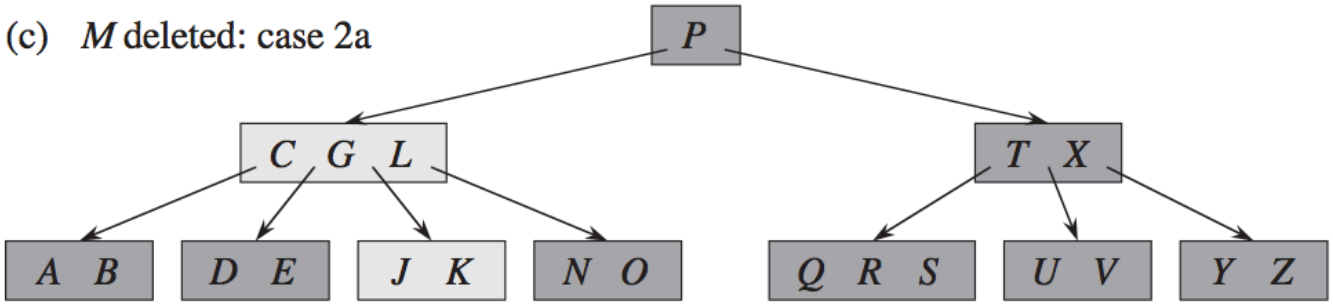
a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

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. (We can find k' and delete it in a single downward pass.)

(b) *F* deleted: case 1



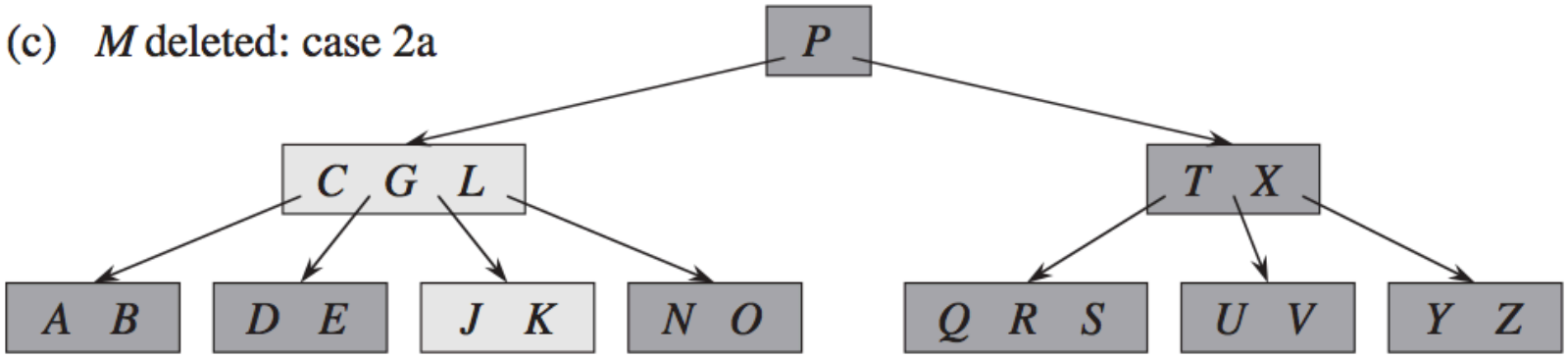
(c) *M* deleted: case 2a



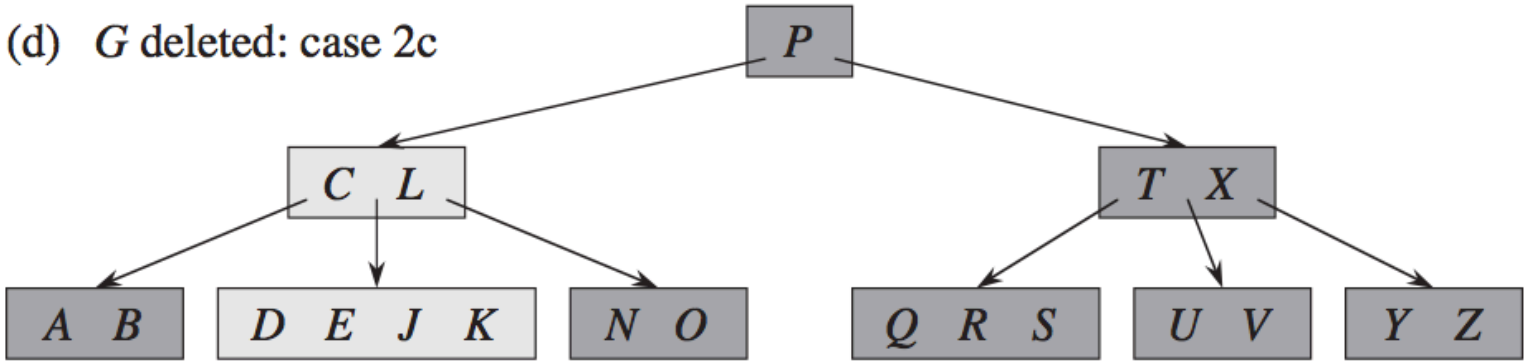
B trees deletion

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.

(c) *M* deleted: case 2a



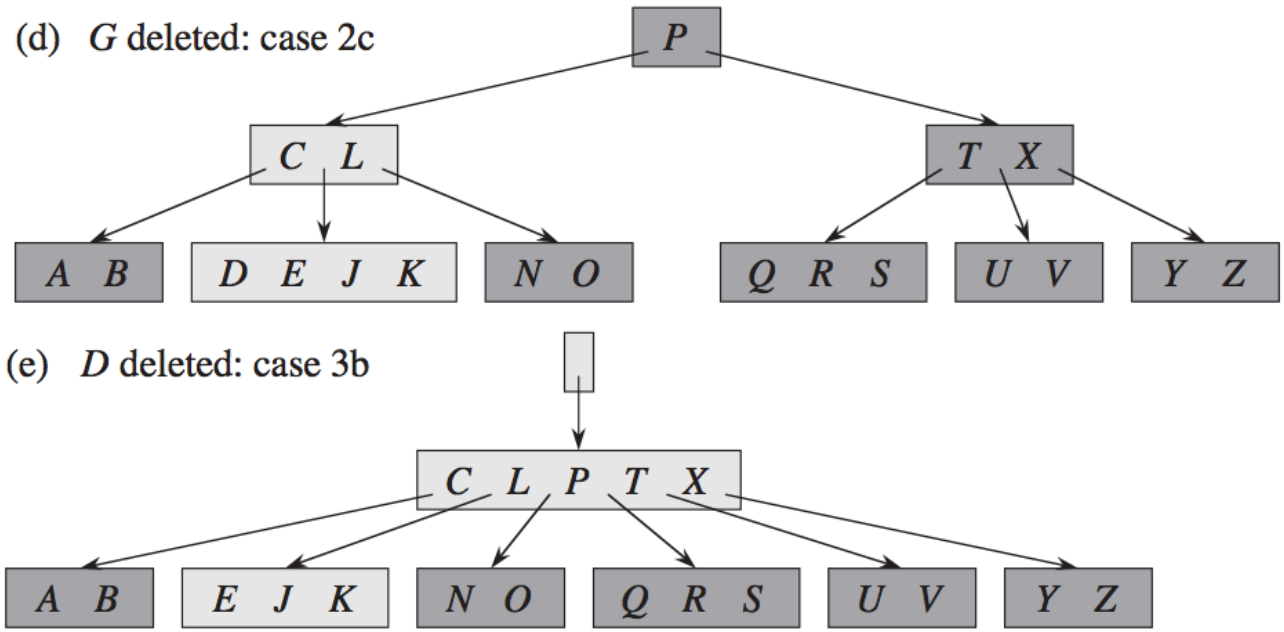
(d) *G* deleted: case 2c



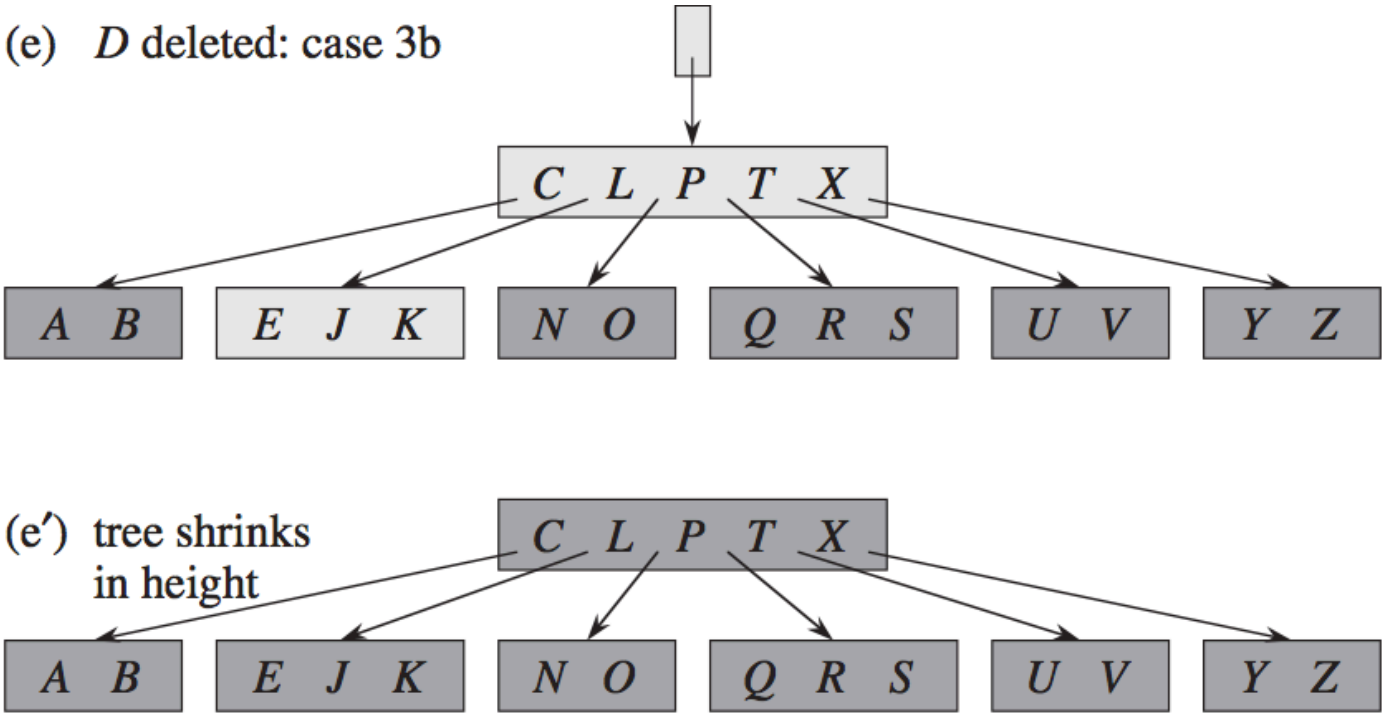
B trees deletion

3. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ 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.

b. 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 a key from x down into the new merged node to become the median key for that node.

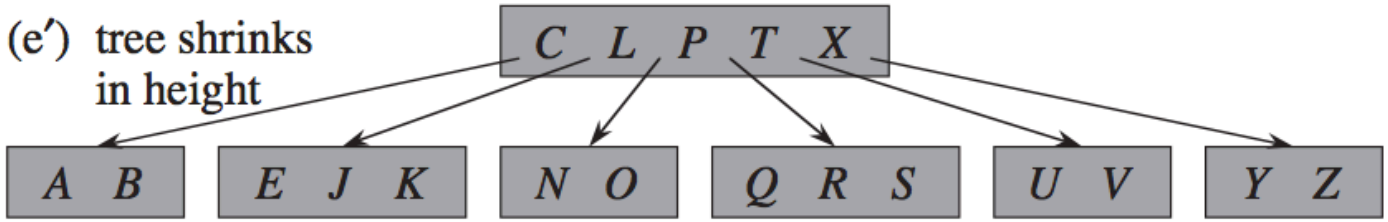


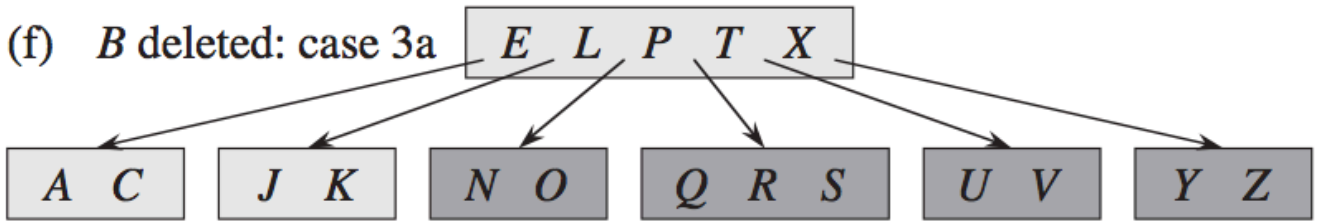
B trees deletion



B trees deletion

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





Running time: $O(th) = O(t \log_t n)$

Homework 3

1. Exercise 12.3-3
2. Problem 13-2
3. Exercise 14.1-6
4. Exercise 18.3-2
5. Problem 18-2

Due date: Nov 2, Thu