COT 6405: Analysis of Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/6405F19.html

Recap of Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- QuickSort
- MergeSort
- HeapSort
- Bucket & Radix Sort
- Counting Sort

Worst Case: $O(N^2)$

Worst Case: O(N logN)

Avg Case: O(N log N)

Lower Bound for Comparison-based Sorting

Worst Case: O(N); Not comparison-based

Tree Sorting

- BST is a search structure that helps efficient search
  - Search can be done in O(h) time, where h = height of BST
  - Also inserts and deletes can be done in O(h) time
  - Unfortunately, Height h = O(N)
- Balanced BST improves BST with h = O(log N)
  - Thus search can be done in O(log N)
  - And, inserts and deletes too can be done in O(log N) time
- We can use BBSTs in the following way:
  - Repeatedly insert N items into a BBST
  - Repeatedly delete the smallest item from the BBST until it is empty
- N inserts and N deletes can be done in O(N log N) time

k-Selection; Median

- Select the k-th smallest item in list
- Naïve Solution
  - Sort;
  - pick the k-th smallest item in sorted list.

    O(n log n) time complexity
- Idea: Modify Partition from QuickSort
  - How?
- Randomized solution: Average case O(n)
- Improved Solution: worst case O(n)

Using Partition for k-Selection

```
PARTITION(array A, int p, int r)
x ← A[r]                                  ▷ Choose pivot
i ← p − 1
for j ← p to r − 1
        do if (A[j] ≤ x)
              then i ← i + 1
                    exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

QuickSelect: a variant of QuickSort

```
QuickSelect(array A, int k, int p, int r)
    ▷ Select k-th largest in subarray A[p..r]
if (p = r)
    then return A[p]
q ← Partition(A, p, r)
i ← q − p + 1        ▷ Compute rank of pivot
if (i = k)
    then return A[q]
if (i > k)
    then return QuickSelect(A, k, p, q)
    else return QuickSelect(A, k − i, q + 1, r)
```

k-Selection Time Complexity

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

- On the average:
  - Rank(pivot) = n / 2
- Average-case time
  - T(N) = T(N/2) + O(N)
  - T(N) = O(N)
- Worst-case time
  - T(N) = T(N-1) + O(N)
  - $T(N) = O(N^2)$

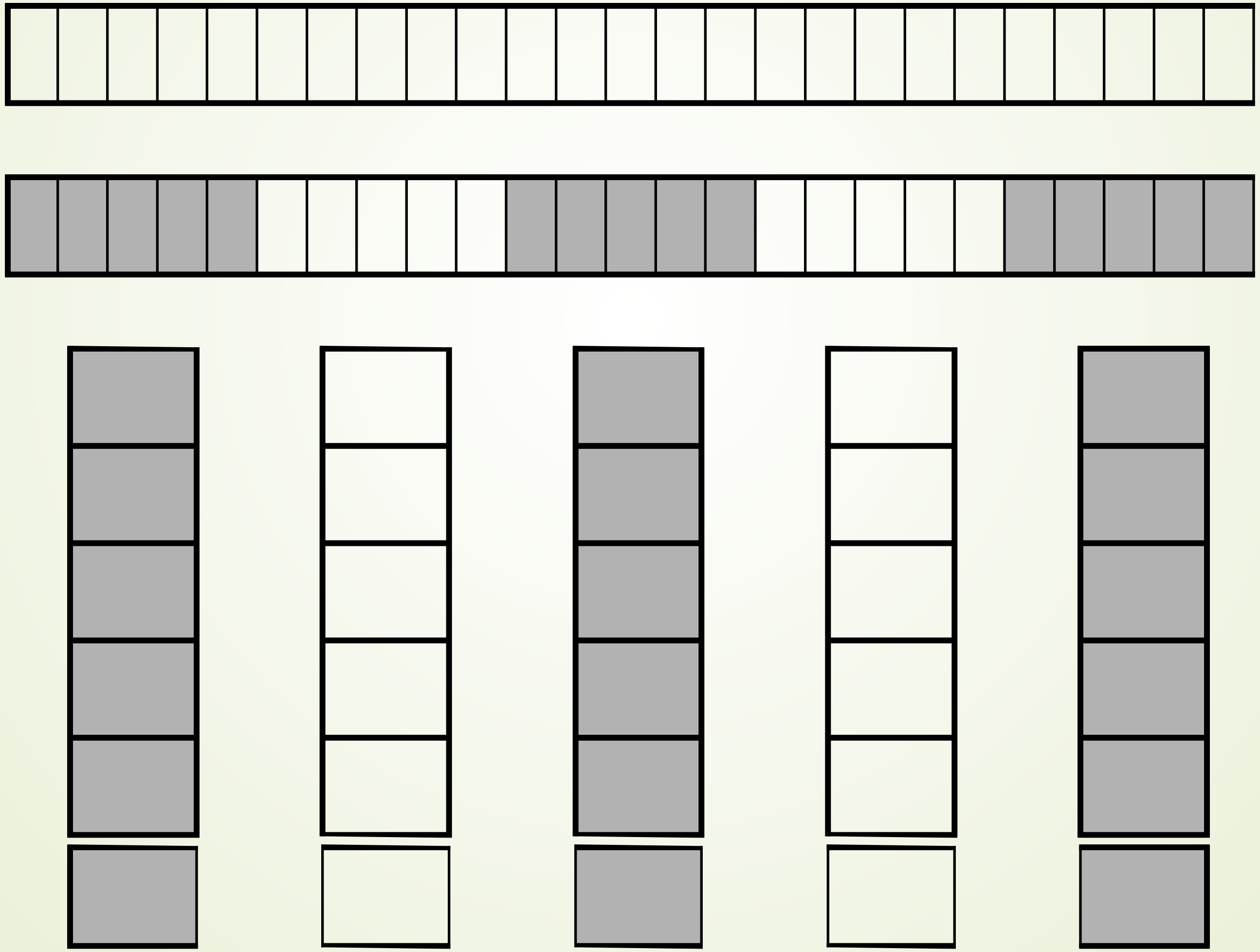
```
PARTITION(array A, int p, int r)
x ← A[r]                              ▷ Choose pivot
i ← p − 1
for j ← p to r − 1
        do if (A[j] ≤ x)
                then i ← i + 1
                        exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Randomized Solution for k-Selection

- Uses RandomizedPartition instead of Partition
  - RandomizedPartition picks the pivot uniformly at random from among the elements in the list to be partitioned.
- Randomized k-Selection runs in O(N) time on the average
- Worst-case behavior is very poor $O(N^2)$

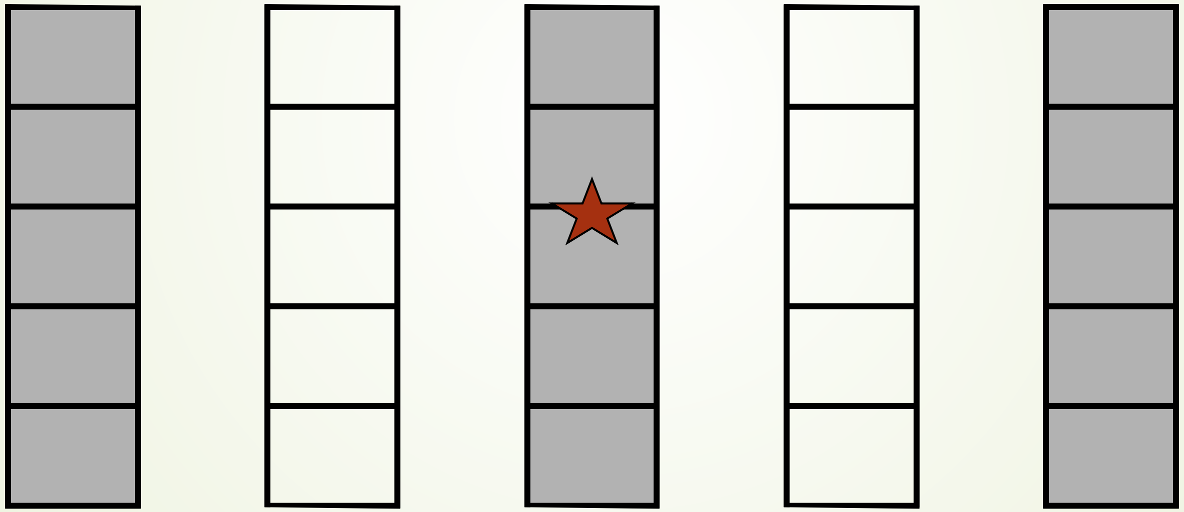
k-Selection & Median: Improved Algorithm

- Start with initial array



k-Selection & Median: Improved Algorithm(Cont'd)

- Use median of medians as pivot



- $T(n) < O(n) + T(n/5) + T(3n/4)$

ImprovedSelect

```
IMPROVEDSELECT(array A, int k, int p, int r)
    ▷ Select k-th largest in subarray A[p..r]
if (p = r)
    then return A[p]
    else N ← r − p + 1
Partition A[p..r] into subsets of 5 elements and
collect all medians of subsets in B[1..⌈N/5⌉].
Pivot ← IMPROVEDSELECT(B, 1, ⌈N/5⌉, ⌈N/10⌉)
q ← PIVOTPARTITION(A, p, r, Pivot)
i ← q − p + 1        ▷ Compute rank of pivot
if (i = k)
    then return A[q]
if (i > k)
    then return IMPROVEDSELECT(A, k, p, q − 1)
    else return IMPROVEDSELECT(A, k − i, q + 1, r)
```

PivotPartition

```
PivotPartition(array A, int p, int r, item Pivot)
    ▷ Partition using provided Pivot
    i ← p − 1
    for j ← p to r
            do if (A[j] ≤ Pivot)
                    then i ← i + 1
                            exchange A[i] ↔ A[j]
    return i + 1
```

Data Structure Evolution

- Standard operations on data structures
  - Search
  - Insert
  - Delete
- Linear Lists
  - Implementation: Arrays (Unsorted and Sorted)
- Dynamic Linear Lists
  - Implementation: Linked Lists
- Dynamic Trees
  - Implementation: Binary Search Trees

BST: Search

```
TREESEARCH(node x, key k)
  ▷ Search for key k in subtree rooted at node x
  if ((x = NIL) or (k = key[x]))
      then return x
  if (k < key[x])
      then return TREESEARCH(left[x], k)
      else return TREESEARCH(right[x], k)
```

Time Complexity: O(h)
h = height of binary search tree

Not O(log n) — Why?

BST: Insert

```
TreeInsert(tree T, node z)
    ▷ Insert node z in tree T
y ← NIL
x ← root[T]
while (x ≠ NIL)
        do y ← x
            if (key[z] < key[x])
                then x ← left[x]
                else x ← right[x]
p[z] ← y
if (y = NIL)
    then root[T] ← z
    else if (key[z] < key[y])
                then left[y] ← z
                else right[y] ← z
```

Time Complexity: $O(h)$
h = height of binary search tree

Search for x in T

Insert x as leaf in T

BST: Delete

```
TREEDELETE(tree T, node z)
  ▷ Delete node z from tree T
  if ((left[z] = NIL) or (right[z] = NIL))
     then y ← z
     else y ← TREE-SUCCESSOR(z)
  if (left[y] ≠ NIL)
     then x ← left[y]
     else x ← right[y]
  if (x ≠ NIL)
     then p[x] ← p[y]
  if (p[y] = NIL)
     then root[T] ← x
     else if (y = left[p[y]])
             then left[p[y]] ← x
             else right[p[y]] ← x
  if (y ≠ z)
     then key[z] ← key[y]
          cop y's satellite data into z
  return y
```

Time Complexity: O(h)
h = height of binary search tree

Set y as the node to be deleted. It has at most one child, and let that child be node x

If y has one child, then y is deleted and the parent pointer of x is fixed.

The child pointers of the parent of x is fixed.

The contents of node z are fixed.

Common Data Structures

	Search	Insert	Delete	Comments
Unsorted Arrays	O(N)	O(1)	O(N)	
Sorted Arrays	O(log N)	O(N)	O(N)	
Unsorted Linked Lists	O(N)	O(1)	O(N)	
Sorted Linked Lists	O(N)	O(N)	O(N)	
Binary Search Trees	O(H)	O(H)	O(H)	H = O(N)
Balanced BSTs	O(log N)	O(log N)	O(log N)	As H = O(log N)

Animations

- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
- https://visualgo.net/
- http://www.cs.armstrong.edu/liang/animation/animation.html
- http://www.cs.jhu.edu/~goodrich/dsa/trees/
- https://www.youtube.com/watch?v=Y-5ZodPvhmM
- http://www.algoanim.ide.sk/

Red-Black (RB) Trees

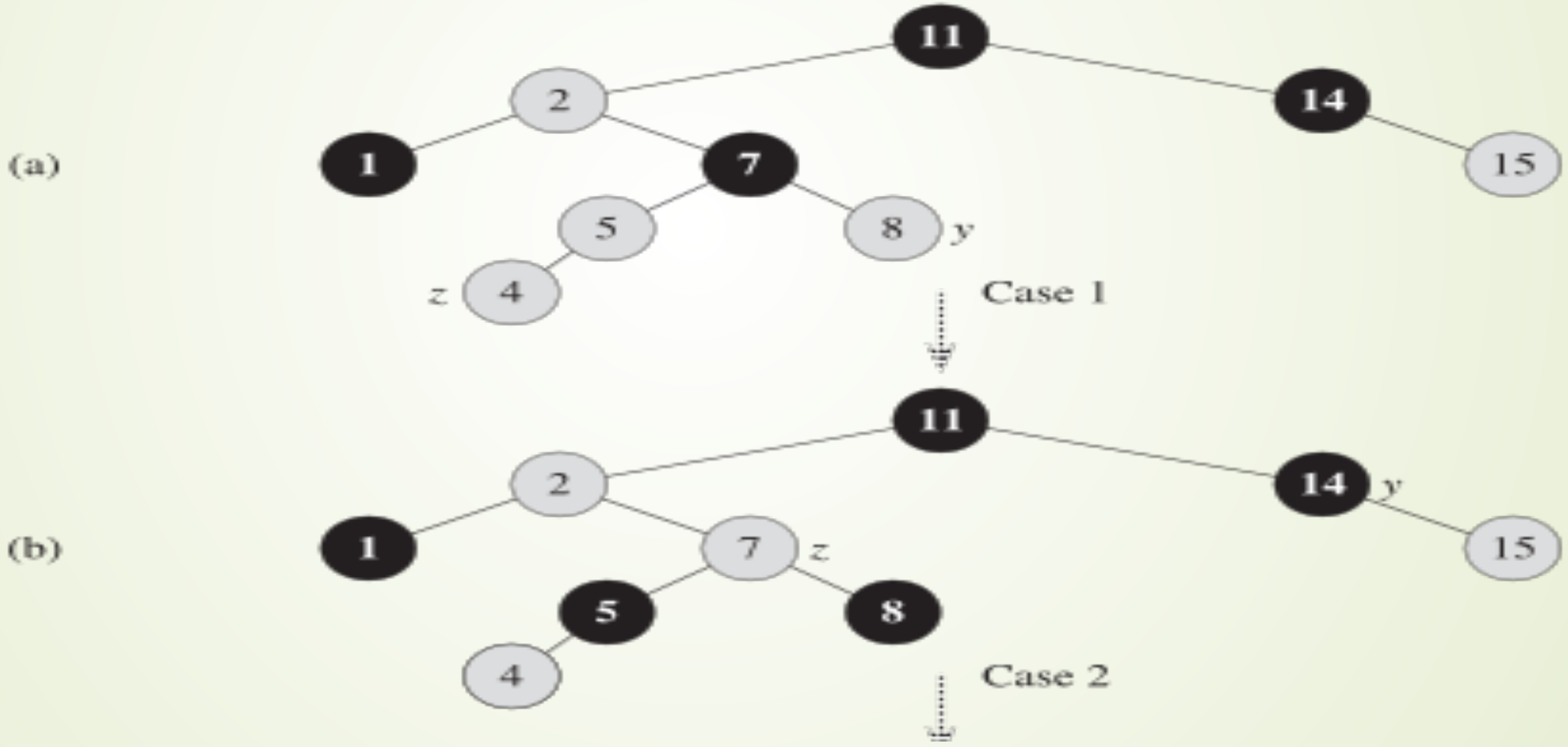
- Every node in a red-black tree is colored either red or black.
  - The root is always black.
  - Every path on the tree, from the root down to the leaf, has the same number of black nodes.
  - No red node has a red child.
  - Every NIL pointer points to a special node called NIL[T] and is colored black.
- Every RB-Tree with n nodes has black height at most logn
- Every RB-Tree with n nodes has height at most 2logn

Red-Black Tree Insert

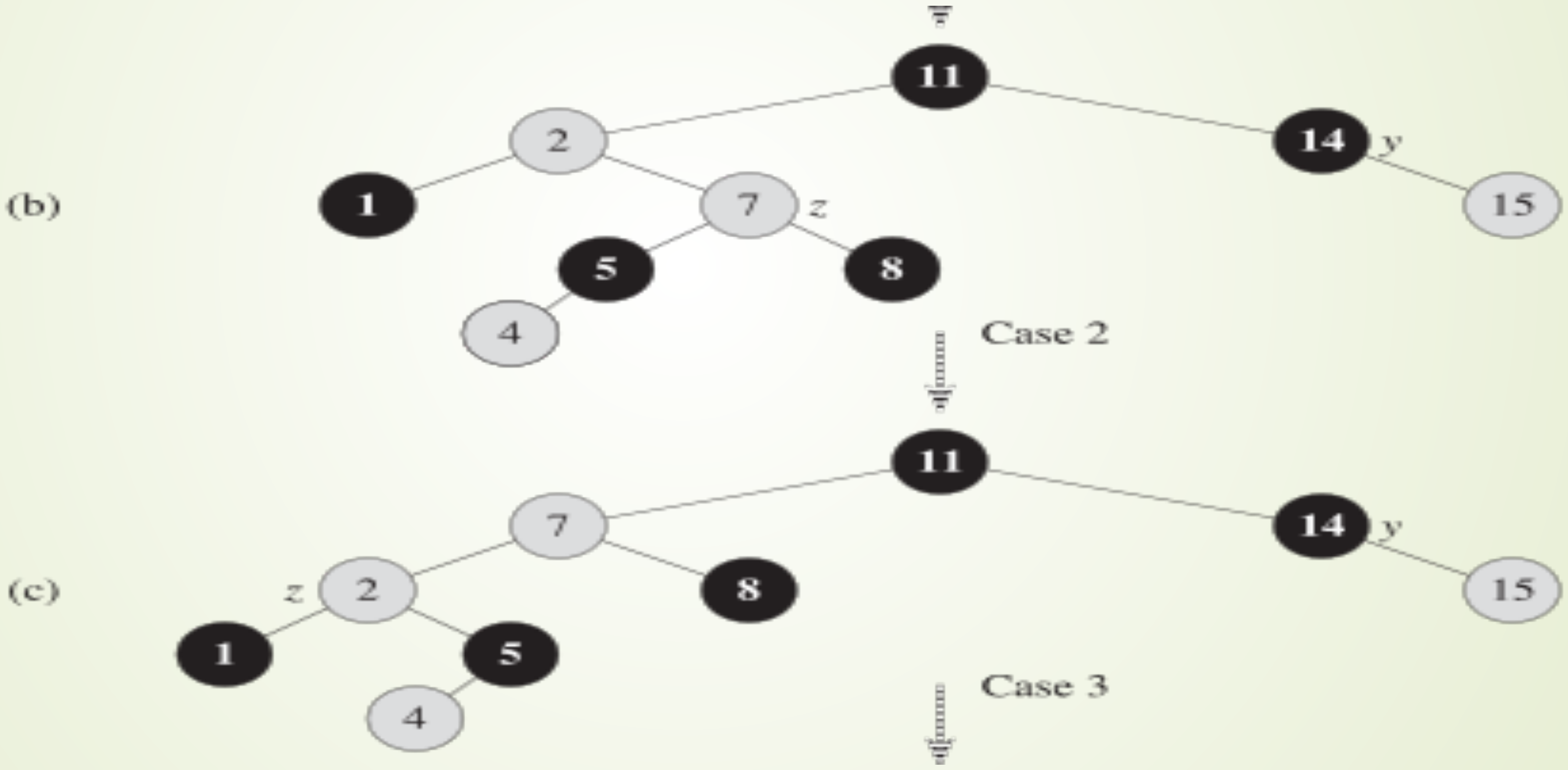
```
RB-Insert (T,z)          // pg 315
        // Insert node z in tree T
        y = NIL[T]
        x = root[T]
        while (x ≠ NIL[T]) do
                y = x
                if (key[z] < key[x])
                        x = left[x]
                        x = right[x]
        p[z] = y
        if (y == NIL[T])
                root[T] = z
        else if (key[z] < key[y])
                left[y] = z
        else right[y] = z
        // new stuff
        left[z] = NIL[T]
        right[z] = NIL[T]
        color[z] = RED
        RB-Insert-Fixup (T,z)
```

```
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               // C-1
            color[p[z]] = BLACK
            color[y] = BLACK
            z = p[p[z]]
            color[z] = RED
          else    if (z == right[p[z]]) then // C-2
                  z = p[z]
                  LeftRotate(T,z)
            color[p[z]] = BLACK       // C-3
            color[p[p[z]]] = RED
            RightRotate(T,p[p[z]])
      else
          // Symmetric code: "right" ↔ "left"
          . . .
    color[root[T]] = BLACK
```

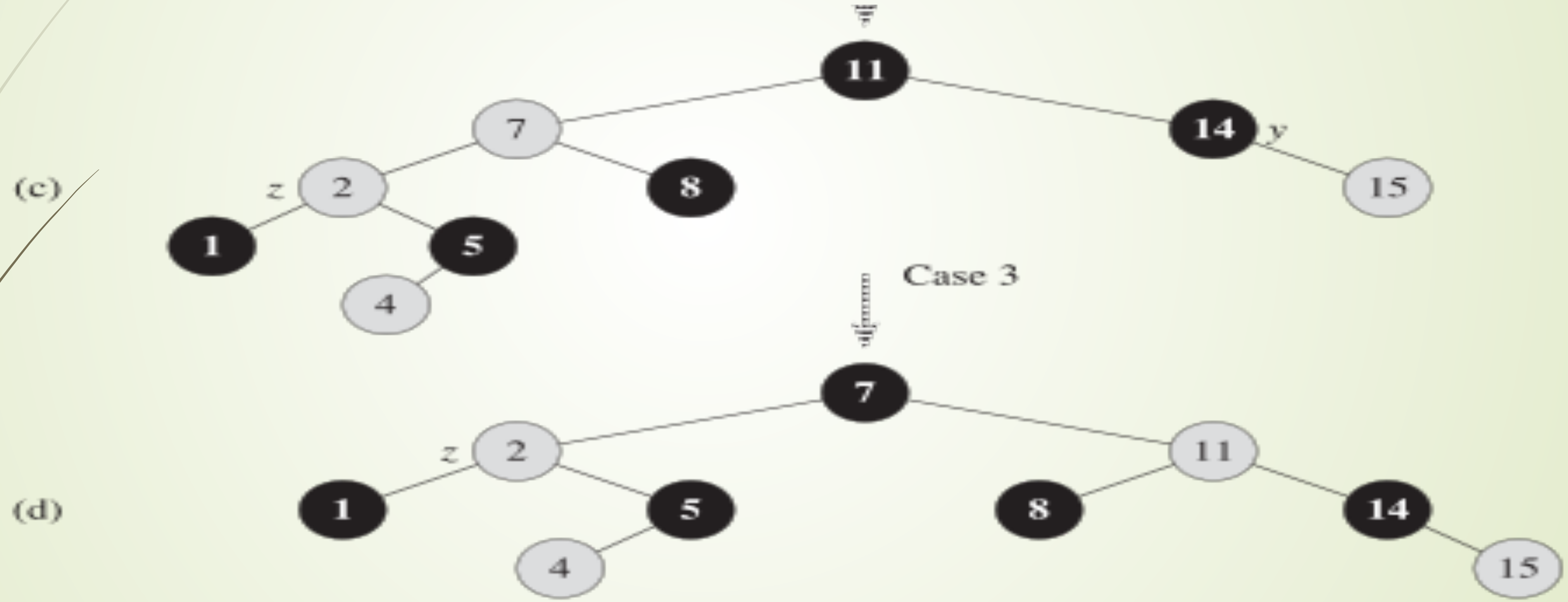
Case 1: Non-elbow; sibling of parent (y) red



Case 2: Elbow case



Case 3: Non-elbow; sibling of parent black



Rotations

```
LeftRotate(T,x)    // pg 278
        // right child of x becomes x's parent.
        // Subtrees need to be readjusted.
        y = right[x]
        right[x] = left[y]     // y's left subtree becomes x's right
        p[left[y]] = x
        p[y] = p[x]
        if (p[x] == NIL[T]) then
              root[T] = y
        else if (x == left[p[x]]) then
              left[p[x]] = y
        else right[p[x]] = y
        left[y] = x
        p[x] = y
```

More Dynamic Operations

	Search	Insert	Delete	Comments
Unsorted Arrays	O(N)	O(1)	O(N)	
Sorted Arrays	O(log N)	O(N)	O(N)	
Unsorted Linked Lists	O(N)	O(1)	O(N)	
Sorted Linked Lists	O(N)	O(N)	O(N)	
Binary Search Trees	O(H)	O(H)	O(H)	H = O(N)
Balanced BSTs	O(log N)	O(log N)	O(log N)	As H = O(log N)

	Se/In/De	Rank	Select	Comments
Balanced BSTs	O(log N)	O(N)	O(N)	
Augmented BBSTs	O(log N)	O(log N)	O(log N)	

Operations on Dynamic RB Trees

- K-Selection
  - Select an item with a specified rank
- "Efficient" solution not possible without preprocessing
- Preprocessing - store additional information at nodes
- Inverse of K-Selection
  - Find rank of an item in the tree
- What information should be stored?
  - Rank
  - ??

OS-Rank

```
OS-RANK(x,y)
// Different from text (recursive version)
// Find the rank of x in the subtree rooted at y
 r = size[left[y]] + 1
 if x = y then return r
else if ( key[x] < key[y] ) then
    return OS-RANK(x,left[y])
else return r + OS-RANK(x,right[y] )
```

Time Complexity O(log n)

OS-Select

```
OS-SELECT(x,i) //page 304
// Select the node with rank i
// in the subtree rooted at x
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)
```

Time Complexity O(log n)

RB-Tree Augmentation

- Augment x with **Size(x)**, where
  - Size(x) = size of subtree rooted at x
  - Size(NIL) = 0

Augmented Data Structures

- Why is it needed?
  - Because basic data structures not enough for all operations
  - storing extra information helps execute special operations more efficiently.
- Can any data structure be augmented?
  - Yes. Any data structure can be augmented.
- Can a data structure be augmented with any additional information?
  - Theoretically, yes.
- How to choose which additional information to store.
  - Only if we can maintain the additional information efficiently under all operations. That means, with additional information, we need to perform old and new operations efficiently maintain the additional information efficiently.

How to augment data structures

1. choose an underlying data structure
2. determine additional information to be maintained in the underlying data structure,
3. develop new operations,
4. verify that the additional information can be maintained for the modifying operations on the underlying data structure.

Augmenting RB-Trees

Theorem 14.1, page 309

Let f be a field that augments a red-black tree T with n nodes, and f(x) can be computed using only the information in nodes x, left[x], and right[x], including f[left[x]] and f[right[x]].

Then, we can maintain f(x) during insertion and deletion without asymptotically affecting the O(log n) performance of these operations.

For example,

size[x] = size[left[x]] + size[right[x]] + 1

rank[x] = ?

Augmenting information for RB-Trees

- Parent
- Height
- Any associative function on all previous values or all succeeding values.
- Next
- Previous

Reading for next class

- Red Black Trees
  - Properties
  - Invariants
  - Insert and Delete
- Mathematical Induction