COT 6405: Analysis of Algorithms

Giri NARASIMHAN

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

Definition of big-Oh

- We say that
  - $F(n) = O(G(n))$

  If there exists two <u>positive</u> constants, c and n_0, such that
  - For all $n \geq n_0$, we have $F(n) \leq c\, G(n)$
- Thus, to show that $F(n) = O(G(n))$, you need to find two positive constants that satisfy the condition mentioned above
- Also, to show that $F(n) \neq O(G(n))$, you need to show that for any value of c, there does not exist a positive constant n_0 that satisfies the condition mentioned above

Algorithm Analysis

- Worst-case time complexity*
  - Worst possible time of all input instances of length N
- (Worst-case) space complexity
  - Worst possible spaceof all input instances of length N
- Average-case time complexity
  - Average time of all input instances of length N

Computation Tree for A on n inputs

- Assume A is a comparison-based sorting alg
- Every node represents a comparison between two items in A
- Branching based on result of comparison
- Leaf corresponds to algorithm halting with output
- Every input follows a path in tree
- Different inputs follow different paths
- Time complexity on input x = depth of leaf where it ends on input x

Upper and Lower Bounds

- Time Complexity of a Problem
  - **Difficulty**: Since there can be many algorithms that solve a problem, what time complexity should we pick?
  - **Solution**: Define upper bounds and lower bounds within which the time complexity lies.
- What is the **upper** bound on time complexity of sorting?
  - **Answer**: Since SelectionSort runs in worst-case $O(N^2)$ and MergeSort runs in $O(N \log N)$, either one works as an upper bound.
  - **Critical Point**: Among all upper bounds, the best is the lowest possible upper bound, i.e., time complexity of the best algorithm.
- What is the **lower** bound on time complexity of sorting?
  - **Difficulty**: If we claim that lower bound is $O(f(N))$, then we have to prove that no algorithm that sorts N items can run in worst-case time $o(f(N))$.

Lower Bounds

- It's possible to prove lower bounds for many comparison-based problems.
- For comparison-based problems, for inputs of length N, if there are P(N) possible solutions, then
  - any algorithm needs $\boxed{\log_2(P(N))}$ to solve the problem.
- Binary Search on a list of N items has at least N + 1 possible solutions. Hence lower bound is
  - $\log_2(N+1)$.
- Sorting a list of N items has at least N! possible solutions. Hence lower bound is
  - $\log_2(N!) = O(N \log N)$
- Thus, MergeSort is an optimal algorithm.
  - Because its worst-case time complexity equals lower bound!

Beating the Lower Bound

- Bucket Sort
  - Runs in time O(N+K) given N integers in range [a+1, a+K]
  - If K = O(N), we are able to sort in O(N)
  - How is it possible to beat the lower bound?
  - Only because we know more about the data.
  - If nothing is know about the data, the lower bound holds.
- Radix Sort
  - Runs in time O(d(N+K)) given N items with d digits each in range [1,K]
- Counting Sort
  - Runs in time O(N+K) given N items in range [a+1, a+K]

Stable Sort

- A sort is stable if equal elements appear in the same order in both the input and the output.
- Which sorts are stable? Homework!

Order Statistics

- Maximum, Minimum
  - Upper Bound
    - O(n) because ??
    - We have an algorithm with a single for-loop: n-1 comparisons
  - Lower Bound
    - n-1 comparisons
- MinMax
  - Upper Bound: 2(n-1) comparisons
  - Lower Bound: 3n/2 comparisons
- Max and 2ndMax
  - Upper Bound: (n-1) + (n-2) comparisons
  - Lower Bound: Harder to prove

7	3	1	9	4	8	2	5

$\underline{\text{Rank}}_A(x)$ = position of x in sorted order o

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)
- <u>Rank</u>(pivot) = 1 + # of items that are smaller than pivot
- If <u>Rank</u>(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

QuickSelect: a variant of QuickSort

$\text{QuickSelect}(\mathit{array}\ A, \mathit{int}\ k, \mathit{int}\ p, \mathit{int}\ r)$

$\quad\triangleright$ Select k-th largest in subarray $A[p..r]$

if $(p = r)$

$\quad$ **then return** $A[p]$

$q \leftarrow \text{Partition}(A, p, r)$

$i \leftarrow q - p + 1 \qquad \triangleright$ Compute rank of pivot

if $(i = k)$

$\quad$ **then return** $A[q]$

if $(i > k)$

$\quad$ **then return** $\text{QuickSelect}(A, k, p, q)$

$\quad$ **else return** $\text{QuickSelect}(A, k - i, q + 1, r)$

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 $O(N^2)$

Readings for next class

- Trees,
- Binary Trees,
- Binary Search Trees,
- Balanced Binary Search Trees

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

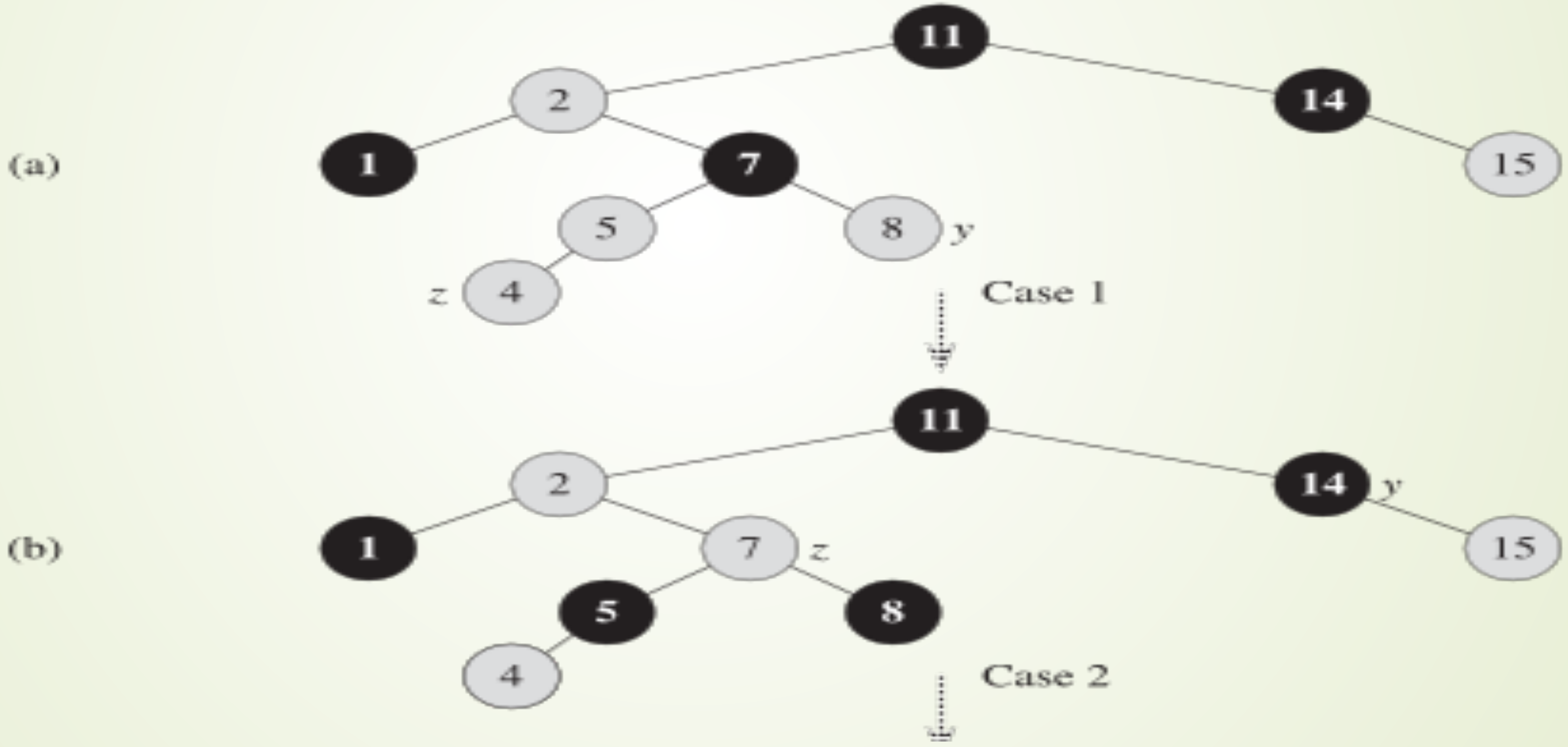
- Every node in a red-black tree is colored either red or black.
  - The root is always black.
  - Every path on the tree, from the root down to the leaf, has the same number of black nodes.
  - No red node has a red child.
  - Every NIL pointer points to a special node called NIL[T] and is colored black.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

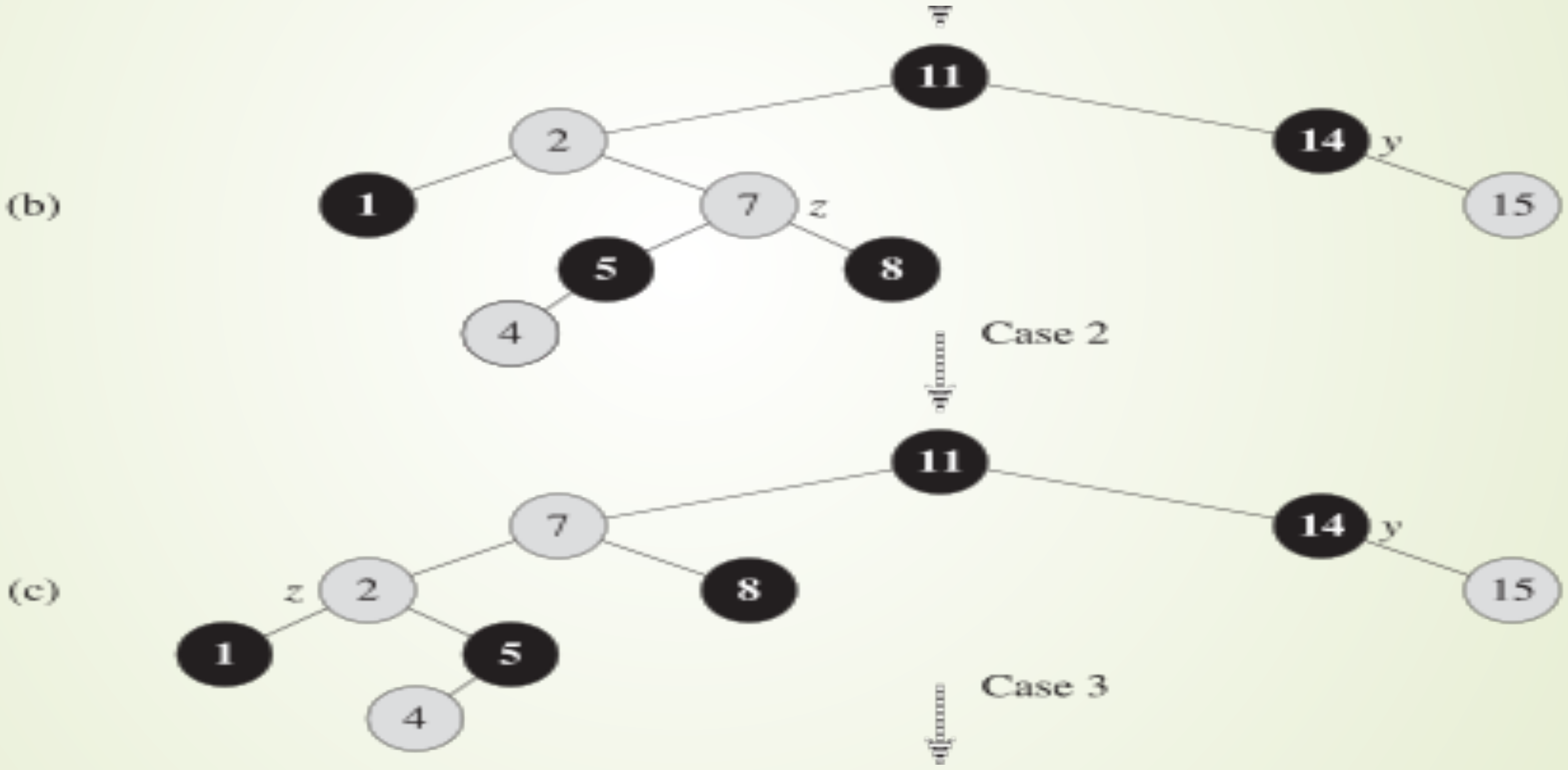
```
RB-Insert (T,z)         // pg 315
        // Insert node z in tree T
        y = NIL[T]
        x = root[T]
        while (x ≠ NIL[T]) do
                y = x
                if (key[z] < key[x])
                        x = left[x]
                        x = right[x]
        p[z] = y
        if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

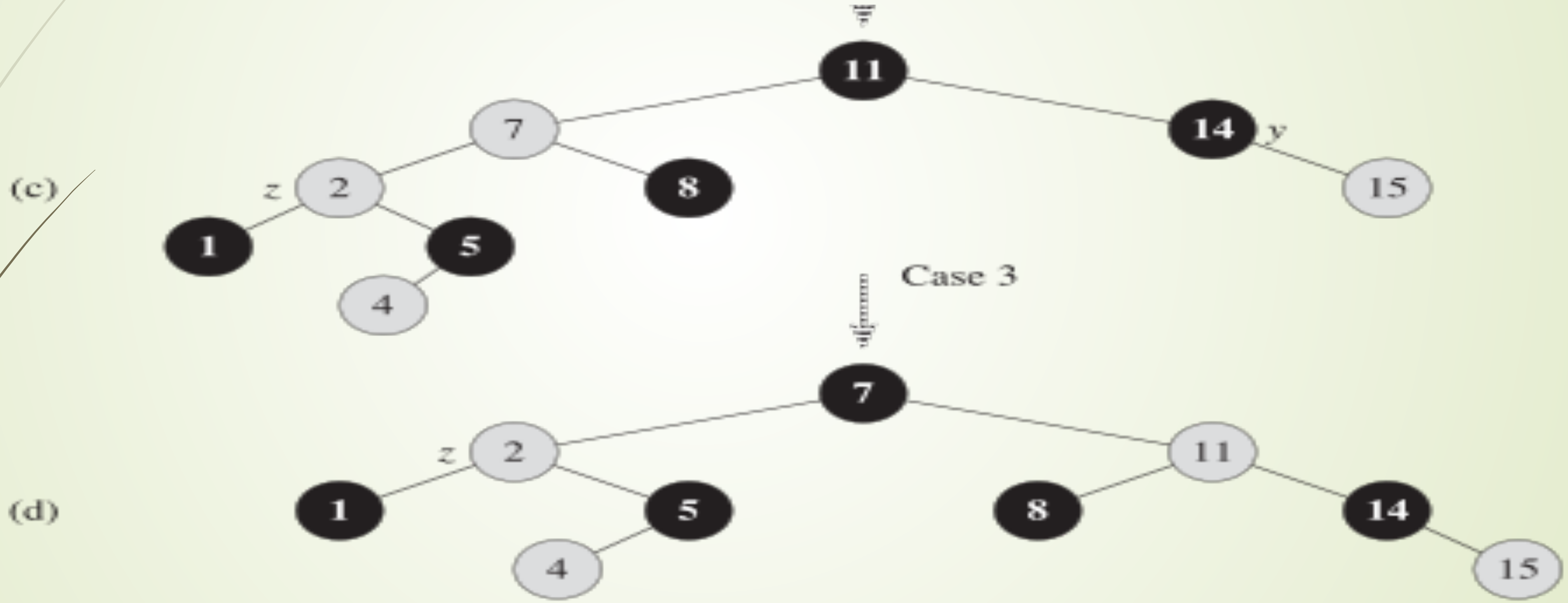
Case 1: Non-elbow; sibling of parent (y) red



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
```

Reading for next class

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