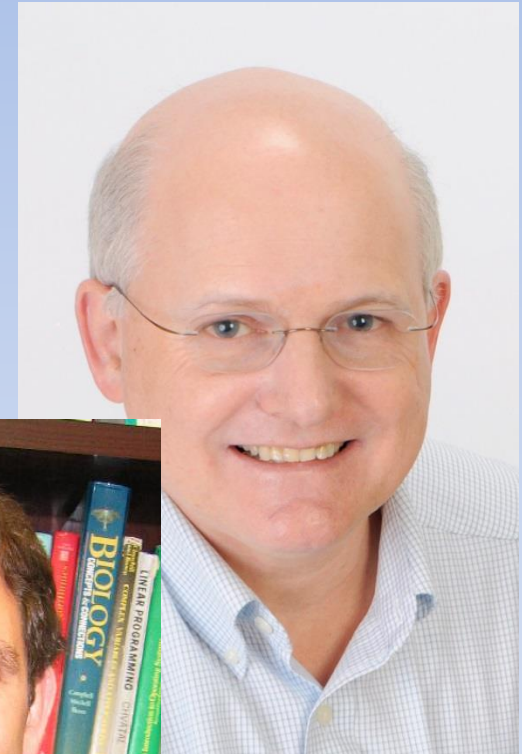


Design and Analysis of Algorithms

Midterm Review



INTRODUCTION TO

ALGORITHMS

THIRD EDITION

Midterm Review - Overview

I Foundations

	Introduction	3
1	The Role of Algorithms in Computing	5
	1.1 Algorithms	5
	1.2 Algorithms as a technology	11
2	Getting Started	16
	2.1 Insertion sort	16
	2.2 Analyzing algorithms	23
	2.3 Designing algorithms	29
3	Growth of Functions	43
	3.1 Asymptotic notation	43
	3.2 Standard notations and common functions	53
4	Divide-and-Conquer	65
	4.1 The maximum-subarray problem	68
	4.2 Strassen's algorithm for matrix multiplication	75
	4.3 The substitution method for solving recurrences	83
	4.4 The recursion-tree method for solving recurrences	88
	4.5 The master method for solving recurrences	93
★	4.6 Proof of the master theorem	97
5	Probabilistic Analysis and Randomized Algorithms	114
	5.1 The hiring problem	114
	5.2 Indicator random variables	118
	5.3 Randomized algorithms	122
★	5.4 Probabilistic analysis and further uses of indicator random variables	130

31	Number-Theoretic Algorithms	926
31.1	Elementary number-theoretic notions	927
31.2	Greatest common divisor	933
31.3	Modular arithmetic	939
31.4	Solving modular linear equations	946
31.5	The Chinese remainder theorem	950
31.6	Powers of an element	954
31.7	The RSA public-key cryptosystem	958
★	31.8 Primality testing	965
★	31.9 Integer factorization	975
32	String Matching	985
32.1	The naive string-matching algorithm	988
32.2	The Rabin-Karp algorithm	990
32.3	String matching with finite automata	995
★	32.4 The Knuth-Morris-Pratt algorithm	1002
33	Computational Geometry	1014
33.1	Line-segment properties	1015
33.2	Determining whether any pair of segments intersects	102
33.3	Finding the convex hull	1029
33.4	Finding the closest pair of points	1039
34	NP-Completeness	1048
34.1	Polynomial time	1053
34.2	Polynomial-time verification	1061
34.3	NP-completeness and reducibility	1067
34.4	NP-completeness proofs	1078
34.5	NP-complete problems	1086
35	Approximation Algorithms	1106
35.1	The vertex-cover problem	1108
35.2	The traveling-salesman problem	1111
35.3	The set-covering problem	1117
35.4	Randomization and linear programming	1123
35.5	The subset-sum problem	1128

	Introduction	3
1	The Role of Algorithms in Computing	5
1.1	Algorithms	5
1.2	Algorithms as a technology	11
2	Getting Started	16
2.1	Insertion sort	16
2.2	Analyzing algorithms	23
2.3	Designing algorithms	29
3	Growth of Functions	43
3.1	Asymptotic notation	43
3.2	Standard notations and common functions	53
4	Divide-and-Conquer	65
4.1	The maximum-subarray problem	68
4.2	Strassen's algorithm for matrix multiplication	75
4.3	The substitution method for solving recurrences	83
4.4	The recursion-tree method for solving recurrences	88
4.5	The master method for solving recurrences	93
★ 4.6	Proof of the master theorem	97
5	Probabilistic Analysis and Randomized Algorithms	114
5.1	The hiring problem	114
5.2	Indicator random variables	118
5.3	Randomized algorithms	122
★ 5.4	Probabilistic analysis and further uses of indicator random variables	130

	Introduction	3
1	The Role of Algorithms in Computing	5
1.1	Algorithms	5
1.2	Algorithms as a technology	11
2	Getting Started	16
2.1	Insertion sort	16
2.2	Analyzing algorithms	23
2.3	Designing algorithms	29
3	Growth of Functions	43
3.1	Asymptotic notation	43
3.2	Standard notations and common functions	53
4	Divide-and-Conquer	65
4.1	The maximum-subarray problem	68
4.2	Strassen's algorithm for matrix multiplication	75
4.3	The substitution method for solving recurrences	83
4.4	The recursion-tree method for solving recurrences	88
4.5	The master method for solving recurrences	93
★	4.6 Proof of the master theorem	97
5	Probabilistic Analysis and Randomized Algorithms	114
5.1	The hiring problem	114
5.2	Indicator random variables	118
5.3	Randomized algorithms	122
★	5.4 Probabilistic analysis and further uses of indicator random variables	130

	Introduction	229
10	Elementary Data Structures	232
	10.1 Stacks and queues	232
	10.2 Linked lists	236
	10.3 Implementing pointers and objects	241
	10.4 Representing rooted trees	246
11	Hash Tables	253
	11.1 Direct-address tables	254
	11.2 Hash tables	256
	11.3 Hash functions	262
	11.4 Open addressing	269
★	11.5 Perfect hashing	277
12	Binary Search Trees	286
	12.1 What is a binary search tree?	286
	12.2 Querying a binary search tree	289
	12.3 Insertion and deletion	294
★	12.4 Randomly built binary search trees	299
13	Red-Black Trees	308
	13.1 Properties of red-black trees	308
	13.2 Rotations	312
	13.3 Insertion	315
	13.4 Deletion	323
14	Augmenting Data Structures	339
	14.1 Dynamic order statistics	339
	14.2 How to augment a data structure	345
	14.3 Interval trees	348

	Introduction	229
10	Elementary Data Structures	232
	10.1 Stacks and queues	232
	10.2 Linked lists	236
	10.3 Implementing pointers and objects	241
	10.4 Representing rooted trees	246
11	Hash Tables	253
	11.1 Direct-address tables	254
	11.2 Hash tables	256
	11.3 Hash functions	262
	11.4 Open addressing	269
★	11.5 Perfect hashing	277
12	Binary Search Trees	286
	12.1 What is a binary search tree?	286
	12.2 Querying a binary search tree	289
	12.3 Insertion and deletion	294
★	12.4 Randomly built binary search trees	299
13	Red-Black Trees	308
	13.1 Properties of red-black trees	308
	13.2 Rotations	312
	13.3 Insertion	315
	13.4 Deletion	323
14	Augmenting Data Structures	339
	14.1 Dynamic order statistics	339
	14.2 How to augment a data structure	345
	14.3 Interval trees	348

Midterm Review

I Foundations

	Introduction	3
1	The Role of Algorithms in Computing	5
	1.1 Algorithms	5
	1.2 Algorithms as a technology	11
2	Getting Started	16
	2.1 Insertion sort	16
	2.2 Analyzing algorithms	23
	2.3 Designing algorithms	29
3	Growth of Functions	43
	3.1 Asymptotic notation	43
	3.2 Standard notations and common functions	53
4	Divide-and-Conquer	65
	4.1 The maximum-subarray problem	68
	4.2 Strassen's algorithm for matrix multiplication	75
	4.3 The substitution method for solving recurrences	83
	4.4 The recursion-tree method for solving recurrences	88
	4.5 The master method for solving recurrences	93
★	4.6 Proof of the master theorem	97
5	Probabilistic Analysis and Randomized Algorithms	114
	5.1 The hiring problem	114
	5.2 Indicator random variables	118
	5.3 Randomized algorithms	122
★	5.4 Probabilistic analysis and further uses of indicator random variables	130

	Introduction	3
1	The Role of Algorithms in Computing	5
1.1	Algorithms	5
1.2	Algorithms as a technology	11
2	Getting Started	16
2.1	Insertion sort	16
2.2	Analyzing algorithms	23
2.3	Designing algorithms	29
3	Growth of Functions	43
3.1	Asymptotic notation	43
3.2	Standard notations and common functions	53
4	Divide-and-Conquer	65
4.1	The maximum-subarray problem	68
4.2	Strassen's algorithm for matrix multiplication	75
4.3	The substitution method for solving recurrences	83
4.4	The recursion-tree method for solving recurrences	88
4.5	The master method for solving recurrences	93
★ 4.6	Proof of the master theorem	97
5	Probabilistic Analysis and Randomized Algorithms	114
5.1	The hiring problem	114
5.2	Indicator random variables	118
5.3	Randomized algorithms	122
★ 5.4	Probabilistic analysis and further uses of indicator random variables	130

Divide-and-Conquer

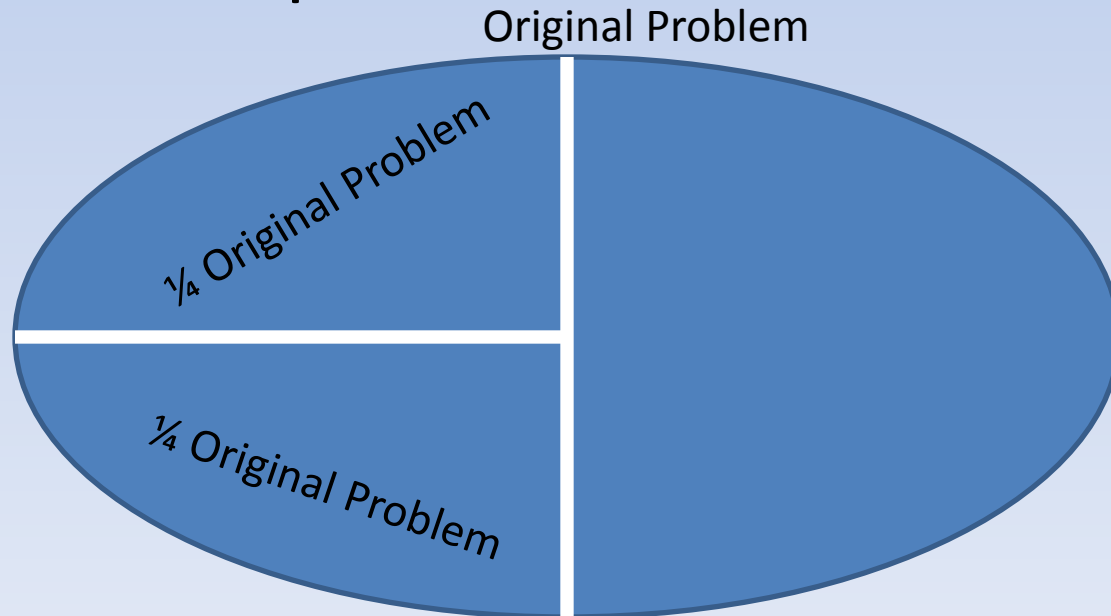
- **Divide** problem into some number of smaller instances of the same problem.
- **Conquer** the subproblems recursively until small enough to just solve.
- **Combine** the subproblem solutions into final solution.

Recurrences w/ Divide & Conquer

- Recurrences define functions recursively
- Recurrence describes function in terms of value on smaller inputs.

Divide & Conquer

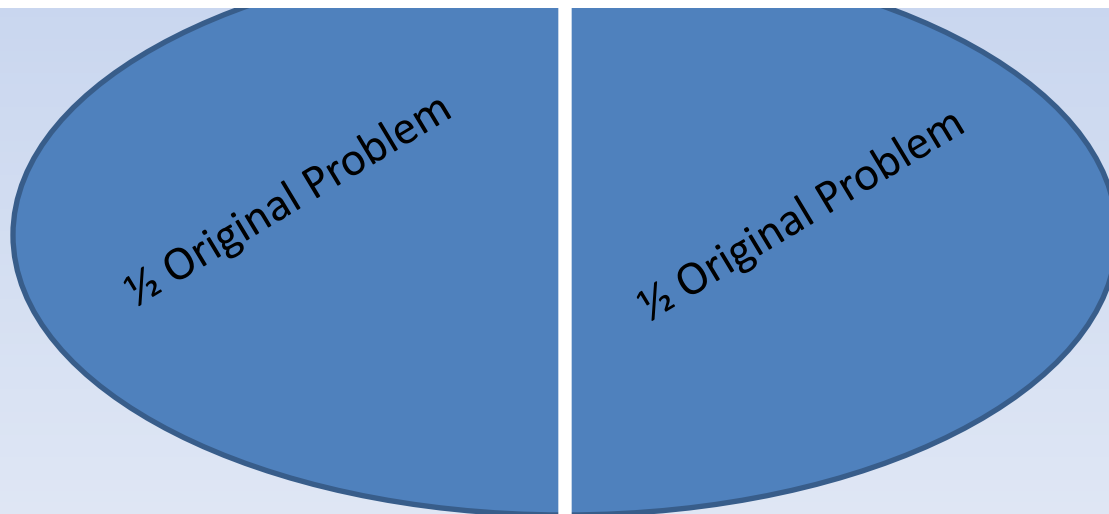
- Break the problem into two equal parts.
- Solve these parts recursively
 - Down to some base case
- Combine two partial solutions



Divide & Conquer: findMax

- Break
- Solve
 - Divide
- Combine

```
6  #include <iostream>
7  #include <vector>
8  #include <cstdlib>
9
10 using namespace std;
11
12 int findMax(int i, int j, vector<int>& A){
13     int mid;
14     int leftMax, rightMax;
15     if (j<=i){
16         return A.at(i);
17     }
18     mid = (i+j)/2;
19     leftMax = findMax(i, mid, A);
20     rightMax = findMax(mid+1, j, A);
21     return max( leftMax, rightMax);
22 }
```



Recurrence Example

- Assume we have an algorithm that:
 - breaks a problem into two equal parts
 - Recursive solves each of the parts
 - Combines the two sub-solutions into the final solution doing constant work to combine solutions.
 - Comparing the two max's
- Model this situation with the recurrence:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(1), & \text{if } n > 1 \end{cases}$$

Recurrence Example

- Assume we have an algorithm that:
 - breaks a problem into two equal parts
 - Recursive solves each of the parts
 - Combines the two sub-solutions into the final solution doing work proportional to input size.
- Model this situation with the recurrence:

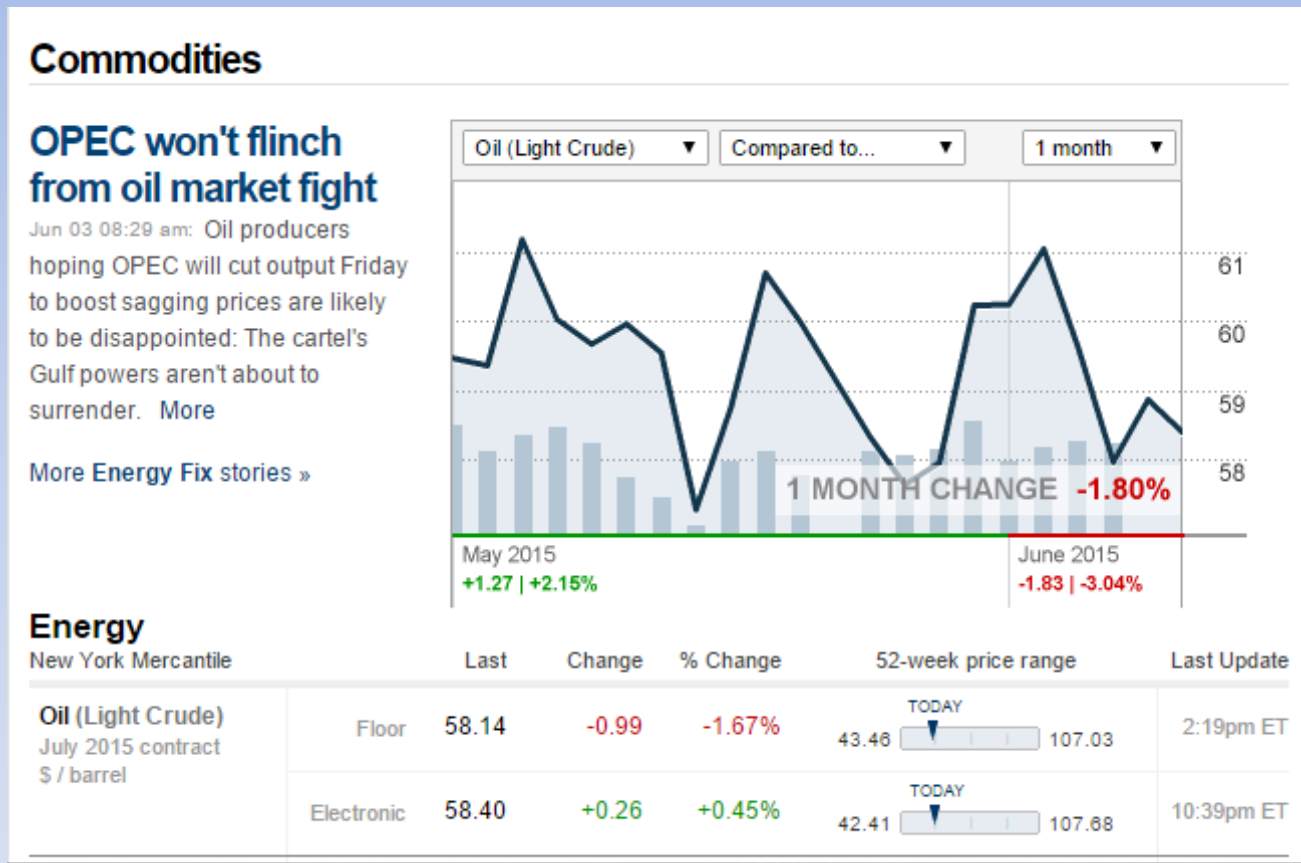
$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & \text{if } n > 1 \end{cases}$$

Recurrence Example

- Assume we have an algorithm that:
 - breaks a problem into two **unequal** parts
 - First part is 2/3 of items
 - Second part is 1/3 of items
 - Combines the two sub-solutions into the final solution doing work proportional to input size.
- Model this situation with the recurrence:

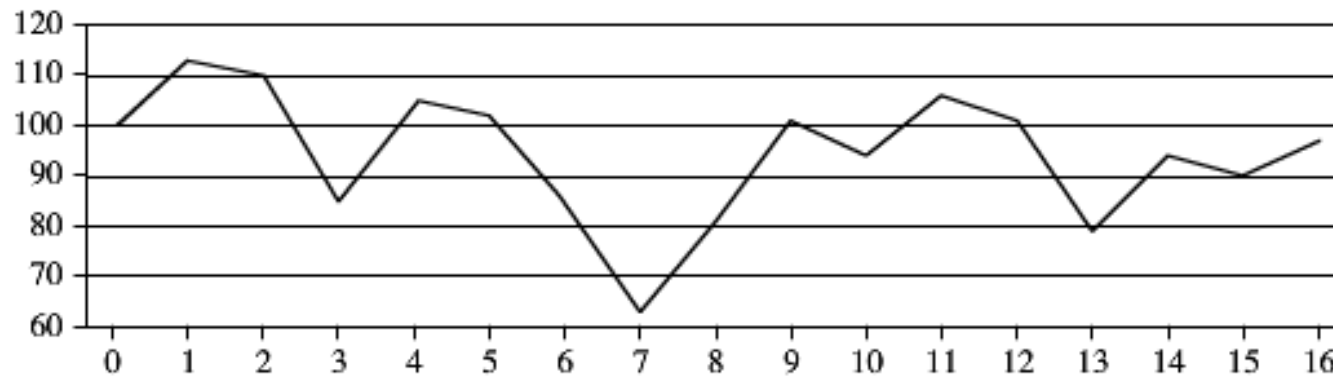
$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + \Theta(n), & \text{if } n > 1 \end{cases}$$

Divide & Conquer Example: The Maximum-Subarray Problem



- Buy Low/Sell High
- How much could I have made???

Divide & Conquer Example: The Maximum-Subarray Problem



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

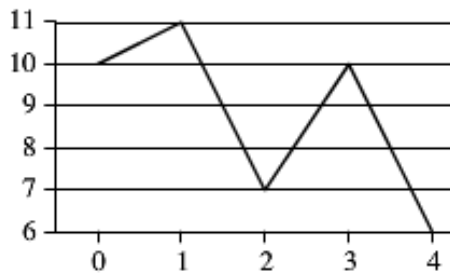
Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

- Buy Low/Sell High
- How much could I have made???

NOT: Buy Lowest/Sell Highest

4.1 The maximum-subarray problem

69



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	-4	3	-4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

- Not as simple as buying at lowest & selling at highest!

Where do I begin?

- Simple Solution frequently possible
 - Brute-Force
- Try every possible buy and sell date:
 - Number of dates = n
 - Number of buy and sell dates =
 - $\binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$
- Can I Do Better???

Consider Transformation

- Want to develop an algorithm $o(n^2)$
 - Meaning strictly smaller than n^2
- To help: Look @ daily change in price
 - NOT daily price

70

Chapter 4 Divide-and-Conquer

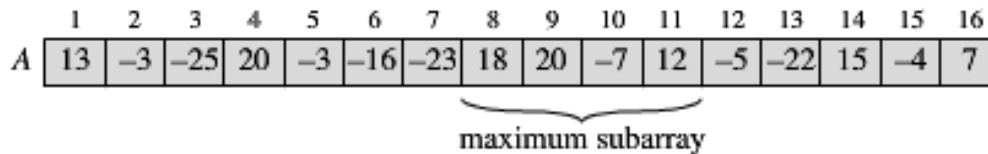


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A .

- Find contiguous subarray whose values have largest sum.
- Maximum-Subarray Problem

Maximum Subarray w/ Divide & Conquer

- Divide our array in half.
- Now, Solution to entire problem must be one of three cases:
 - Entirely in first half
 - Entirely in second half
 - Beginning in first half and ending in second half.
- First Half and Second Half solutions found Recursively.

Maximum Subarray Crossing Midpoint

- Not a smaller version of original problem.
- Subarray *must cross* midpoint.
- Any subarray crossing midpoint must be made of two subarrays:
 - Subarray starting in left and ending at midpoint
 - Subarray starting at midpoint and ending in right.

Crossing Midpoint Subarray

4.1 The maximum-subarray problem

71

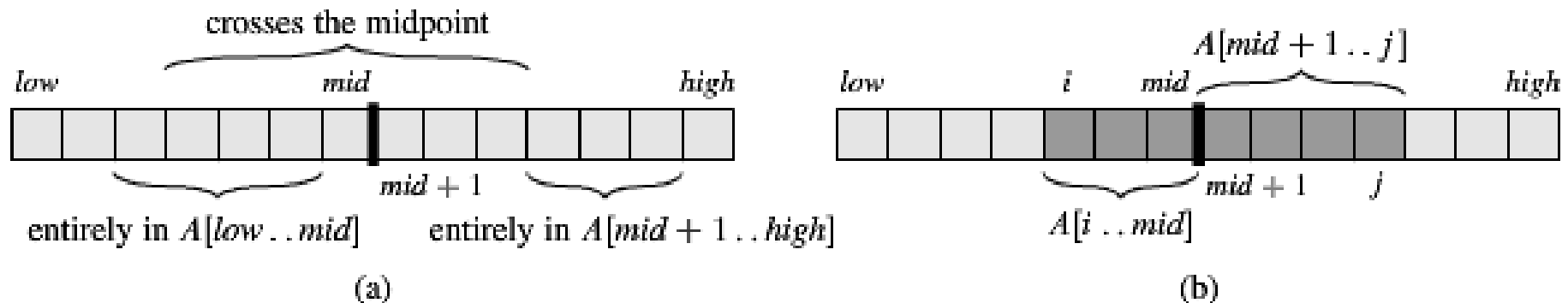



Figure 4.4 (a) Possible locations of subarrays of $A[low..high]$: entirely in $A[low..mid]$, entirely in $A[mid+1..high]$, or crossing the midpoint mid . (b) Any subarray of $A[low..high]$ crossing the midpoint comprises two subarrays $A[i..mid]$ and $A[mid+1..j]$, where $low \leq i \leq mid$ and $mid < j \leq high$.

- Find maximum subarray ending at mid
- Find maximum subarray starting and mid
- Combine

FIND-MAX-CROSSING-SUBARRAY(*A, low, mid, high*)

```
1  left-sum =  $-\infty$  
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum =  $-\infty$ 
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

Pseudocode

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
1  if high == low
2      return (low, high, A[low])           // base case: only one element
3  else mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
4      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum  $\geq$  right-sum and left-sum  $\geq$  cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum  $\geq$  left-sum and right-sum  $\geq$  cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

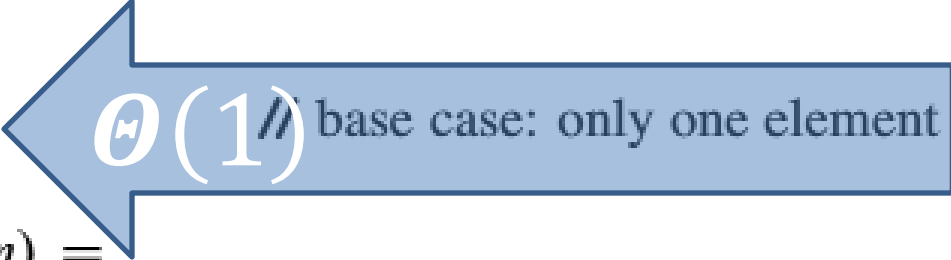
Analyzing Performance

- Algorithm is Recursive:
 - Establish Recurrence Relation
- Analysis Assumes problem size is power of 2
 - Therefore all subproblem sizes are integers
- Clearly the base case $T(1)$ on line 2 takes constant time
 - Return (low, high, A[low])
 - $T(1) = \Theta(1)$

Pseudocode

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
1  if high == low
2      return (low, high, A[low])
3  else mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
4      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum  $\geq$  right-sum and left-sum  $\geq$  cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum  $\geq$  left-sum and right-sum  $\geq$  cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```



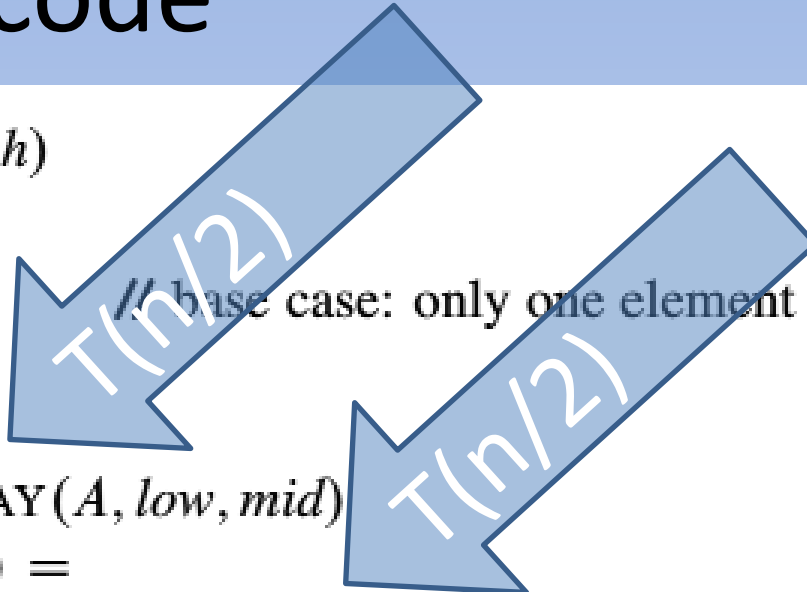
Analyzing Performance (2)

- Recursive case given problem size a power of 2:
 - Each of the Two Recursive calls to Find-Maximum-Subarray is applied to a problem size $n/2$.
 - $T(n/2) + T(n/2) = 2T(n/2)$ required by Lines 4 and 5

Pseudocode

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
1  if high == low
2      return (low, high, A[low])
3  else mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
4      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum  $\geq$  right-sum and left-sum  $\geq$  cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum  $\geq$  left-sum and right-sum  $\geq$  cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```



// base case: only one element

Pseudocode

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
1  if high == low
2      return (low, high, A[low]) // base case: only one element
3  else mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
4      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum  $\geq$  right-sum and left-sum  $\geq$  cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum  $\geq$  left-sum and right-sum  $\geq$  cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

Analyzing Performance (3)

- $\Theta(n)$ required by Find-Max-Crossing-Subarray

FIND-MAX-CROSSING-SUBARRAY(*A*, *low*, *mid*, *high*)

```
1  left-sum =  $-\infty$ 
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum =  $-\infty$ 
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```



$\Theta(n/2)$

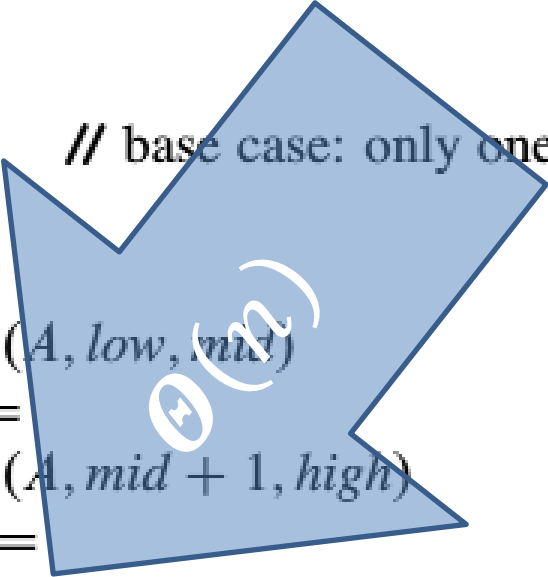


$\Theta(n/2)$

Analyzing Performance (4)

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
1  if high == low
2      return (low, high, A[low])
3  else mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
4      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum  $\geq$  right-sum and left-sum  $\geq$  cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum  $\geq$  left-sum and right-sum  $\geq$  cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```



// base case: only one element

Analyzing Performance (5)

- $$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1)$$
$$= 2T(n/2) + \Theta(n)$$

Analyzing Performance (5)

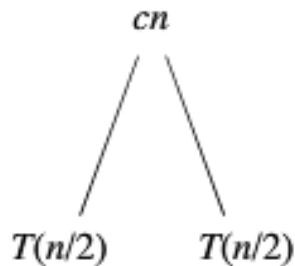
- $$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1)$$
$$= 2T(n/2) + \Theta(n)$$

38

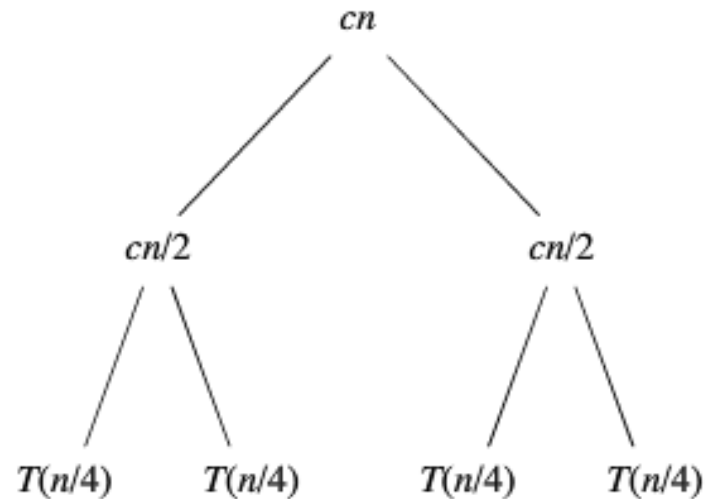
Chapter 2 Getting Started



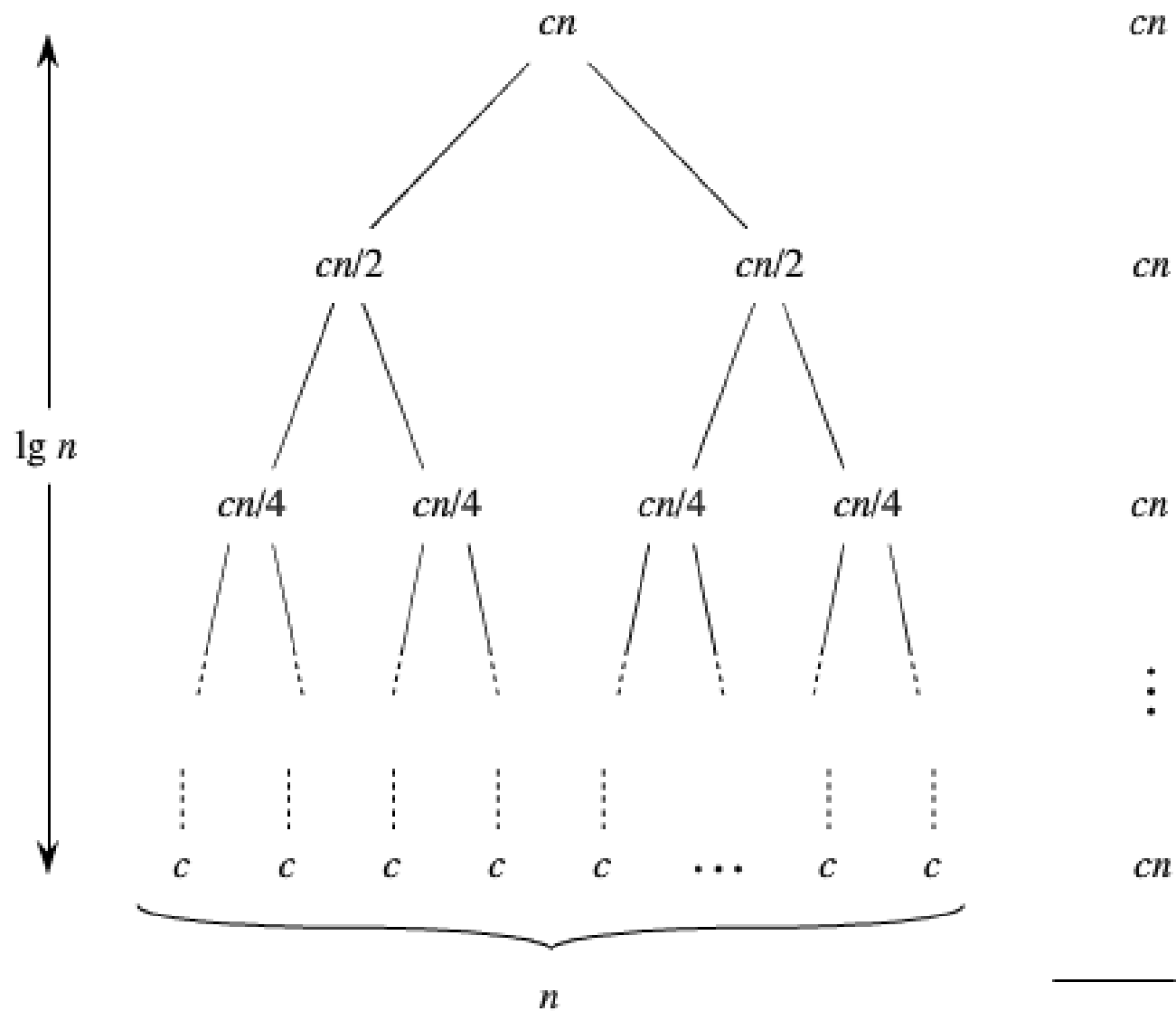
$T(n)$



(a)



(c)



(d)

$$T(n) = cn \lg n + cn$$

Analyzing Performance (5)

- Model final recurrence:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & \text{if } n > 1 \end{cases}$$

- $T(n) = \Theta(n \lg n)$

Divide & Conquer Summary

- Split the problem in half and recursively solve smaller versions
 - Creating a $\text{Lg}(n)$ recursion tree
- Merge sub solutions in $\Theta(n)$
 - Each level of recursion has the same n items
 - Depth of recursion is $\text{Lg } n$
- Smart Merge Procedure needed!

Divide & Conquer w/ Merge sort

- Sort Problem: Take an n -element sequence of numbers and find a permutation where elements are ordered smallest to largest.
- Merge sort:
 - **Divide** the n -element sequence into two subsequence of $n/2$ elements each.
 - **Conquer** the two subsequences recursively.
 - **Combine** the two sorted subsequences for the final solution.

Divide & Conquer w/ Merge sort

- Recursion bottoms out when the sequence is of size 1, which is sorted by definition.
- Key operation is MERGING two sorted sequences.
- Merging is done in time $\Theta(n)$ where n is the total number of items to be merged.

Merge Sort

MERGE-SORT(A, p, r)

1 **if** $p < r$

2 $q = \lfloor (p + r) / 2 \rfloor$

3 MERGE-SORT(A, p, q)

4 MERGE-SORT($A, q + 1, r$)

5 MERGE(A, p, q, r)

- Develop Recurrence Relation
 - Divide step (and base case): $\Theta(1)$
 - Conquer: $T(n) = T(n/2) + T(n/2)$
 - Combine: Merge procedure is $\Theta(n)$

Merge Sort

MERGE-SORT(A, p, r)

1 **if** $p < r$

2 $q = \lfloor (p + r) / 2 \rfloor$

3 MERGE-SORT(A, p, q)

4 MERGE-SORT($A, q + 1, r$)

5 MERGE(A, p, q, r)

- Model final recurrence:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & \text{if } n > 1 \end{cases}$$

- $T(n) = \Theta(n \lg n)$

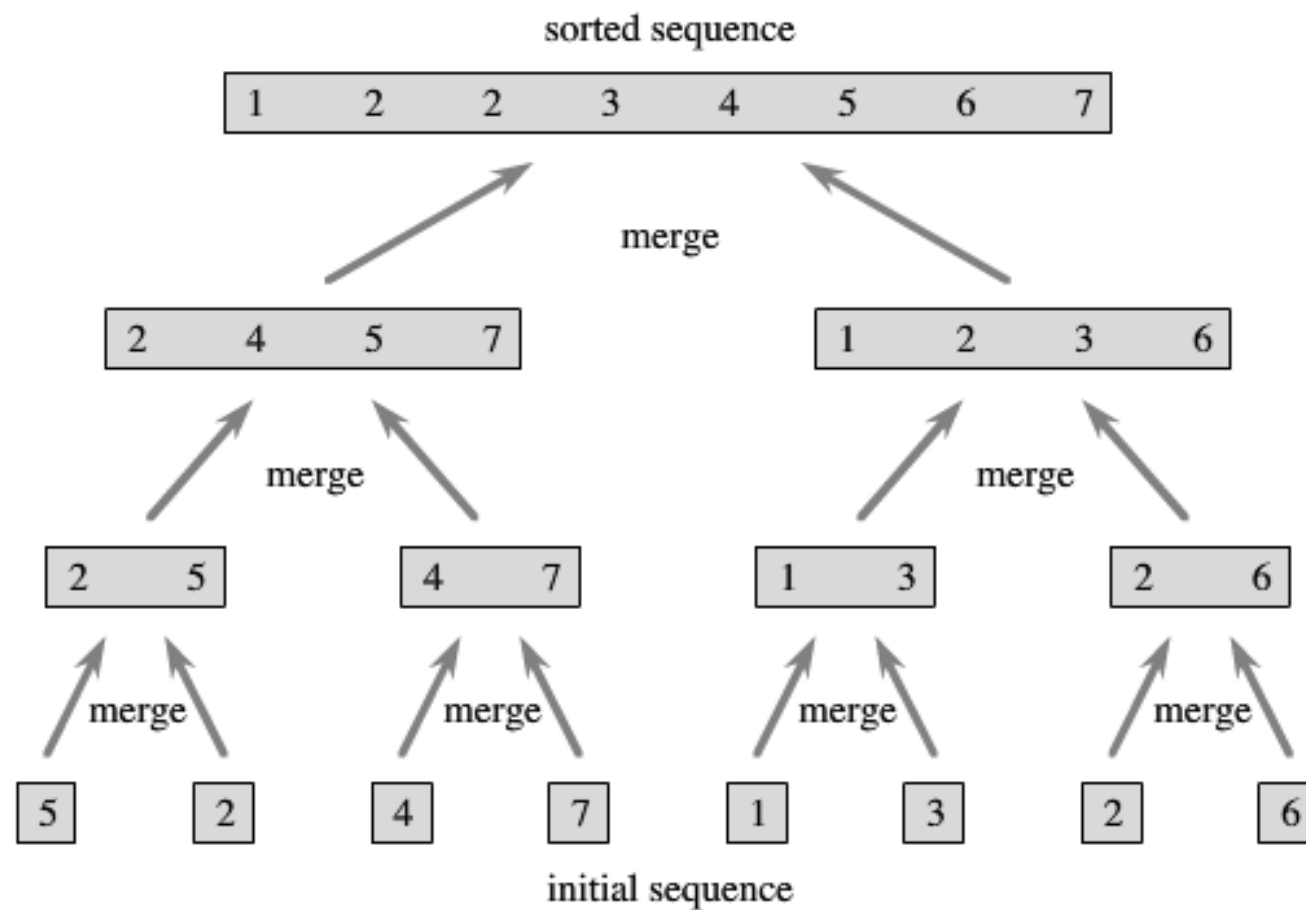
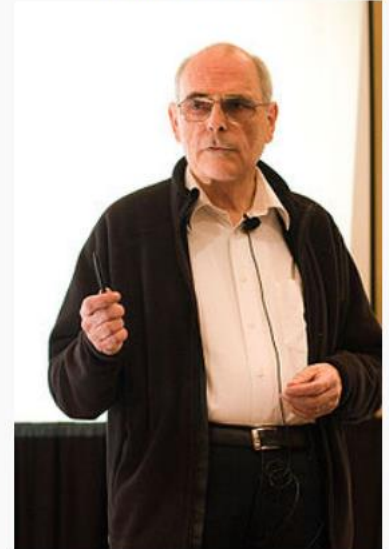


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Strassen Matrix Multiplication

- First published in 1969
- Improves upon the standard matrix multiplication algorithm $O(n^3)$
- $O(n^{\lg 7}) = n^{2.807355}$

Volker Strassen



Volker Strassen giving the Knuth Prize lecture at SODA 2009

Born	April 29, 1936 (age 79) Düsseldorf-Gerresheim, Prussia
Nationality	German
Fields	Mathematics
Institutions	University of Konstanz
Alma mater	University of Göttingen
Doctoral advisor	Konrad Jacobs <small>(de)</small>
Doctoral students	Peter Bürgisser Joos Heintz Joachim von zur Gathen

SQUARE-MATRIX-MULTIPLY(A, B)

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

- Triple loop: n^3

Simple Divide & Conquer

Break Matrix into 4 Quadrants

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \quad (4.9)$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \quad (4.10)$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \quad (4.11)$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \quad (4.12)$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \quad (4.13)$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \quad (4.14)$$

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 
```

- Have we done better?

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```

1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 

```

- Have we done better?
- Base Case:

$$T(1) = \Theta(1) . \tag{4.15}$$

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```

1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 

```

- Have we done better?
- Recursive Case:
 - Partition Matrices: $\Theta(1)$ w/ Indices Calculations
 - Add two $n/2$ Square Matrices: $(n/2)^2$ elements each
 - $n^2/4 \Rightarrow \Theta(n^2)$
 - 8 Recursive calls of size $n/2$

$$\begin{aligned}
 T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\
 &= 8T(n/2) + \Theta(n^2) .
 \end{aligned}
 \tag{4.16}$$

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```

1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 
```

- Have we done better?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases} \quad (4.17)$$

- We'll look closer at solving this recurrence, but for now:
 - $T(n) = \Theta(n^3)$
- We have NOT done better!!
- Where's the problem??

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```

1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 

```

• Have we done better?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

TOO MANY SUBPROBLEMS!!

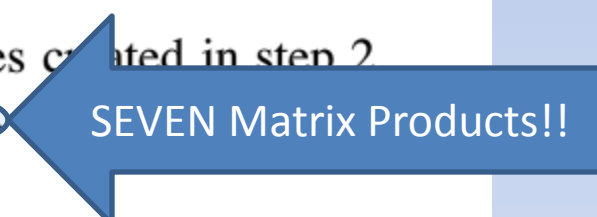
• We'll look closer at solving this recurrence, but for now:

- $T(n) = \Theta(n^3)$
- We have NOT done better!!
- Where's the problem??
 - Strassen's Improvement... FEWER SUBPROBLEMS!!!

Strassen's Method

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices S_1, S_2, \dots, S_{10} , each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products P_1, P_2, \dots, P_7 . Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

Strassen's Method

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices S_1, S_2, \dots, S_{10} , each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2 recursively compute seven matrix products P_1, P_2, \dots, P_7 .  SEVEN Matrix Products!!
 $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

- Key Contribution: 7 Recursive Calls!!!
 - NOT 8!!

Recurrence Relation w/ Strassen's Method


$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases} \quad (4.18)$$

SEVEN Matrix Products!!

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices S_1, S_2, \dots, S_{10} , each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products P_1, P_2, \dots, P_7 . Each P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

SEVEN Matrix Products!!

Recurrence Relation w/ Strassen's Method

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases} \quad (4.18)$$


SEVEN Matrix Products!!

- Instead of $T(n) = \Theta(n^3)$
– $\Theta(n^{\log_2 8})$
- For Strassen's Algorithm $\Theta(n^{\log_2 7})$

Finding Closest Pair of Points

Finding the closest pair of points w/ Divide & Conquer

- Consider the Problem of finding the two closest pair of points in a set Q of points.
- Brute force will look at all pairs resulting in $\Theta(n^2)$
- Textbook provides an algorithm whose running time is described by our familiar recurrence... $T(n) = 2T(n/2) + O(n)$
- Resulting runtime is therefore $O(n \lg n)$

Divide-and-Conquer Algorithm

- Algorithm takes as input:
 - P a subset of points
 - X and Y which contain points sorted by x -coordinate and y -coordinate respectively.

Divide-and-Conquer Algorithm

- Divide:
 - Find a vertical line that bisects the point set P in half, P_L and P_R
 - Divide the X and Y into X_L, X_R, Y_L, Y_R based on whether they are in P_L or P_R maintaining the sorted orders.
- Conquer:
 - Now make two recursive calls with (P_L, X_L, Y_L) and (P_R, X_R, Y_R) . Let δ equal the minimum distance of the closest pairs from the two recursive calls.
- Combine:
 - Uses the δ from recursive calls to limit the points tested for possible closest pairs overlapping center.

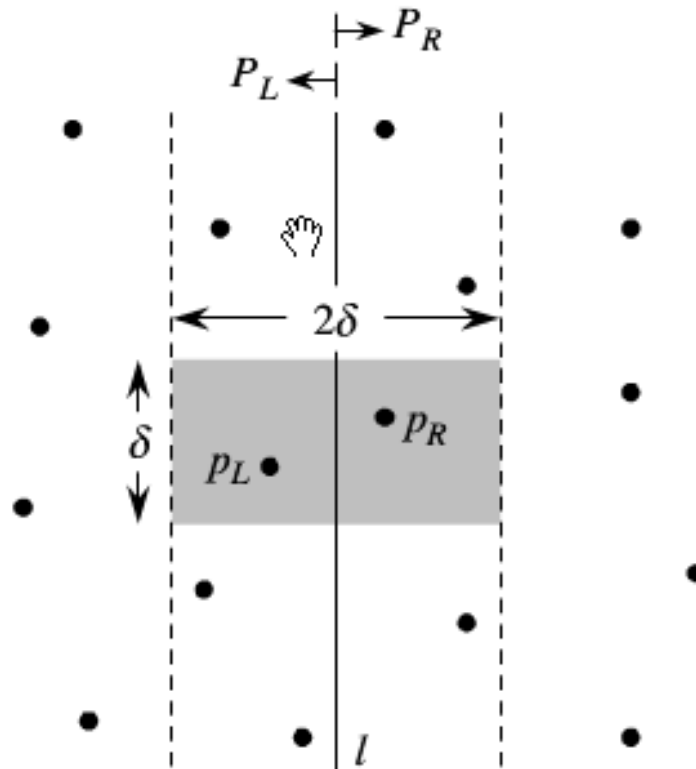
Combine Step

- Closest pair is either the pair with distance δ found by one of the recursive calls
- OR Closest pair is a pair of points with one point in P_L and the other in P_R .
- Algorithm determines whether there is a pair with one point in P_L and the other point in P_R and whose distance is less than δ .
- NOTE: if a pair of points has distance less than δ , both points of the pair must be within δ units of line l .
 - Thus residing in the 2δ –wide vertical strip centered at line l .

Combine Step

1042

Chapter 33 Computational Geometry

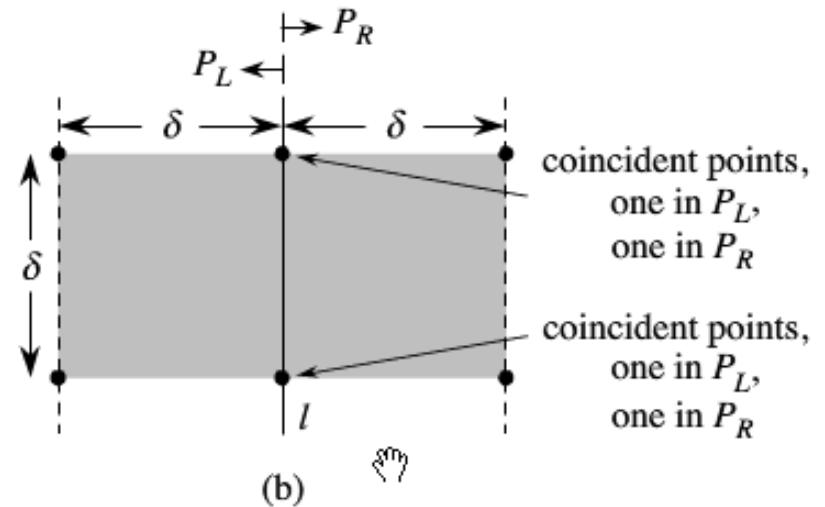


(a)

Combine

1. Create array Y' by removing from Y all but points in 2δ -wide vertical strip leaving array sorted by y-coordinate (like Y)
2. For each point p in array Y' :
 - Try to find points in Y' that are within δ units of p .
 - Only 7 points that follow p need be considered!
3. If pair of points closer than δ found in step 2 return them, else return closest pair from recursive step.

Checking 7



- Assume closest pair of points i $p_L \in P_L$ and $p_R \in P_R$.
- Distance δ' between p_L and p_R is strictly less than δ .
- Point p_L must be on or to the left of line l and less than δ units away.
- Point p_R is on or to the right of l and less than δ units away.
- Moreover, p_L and p_R are within δ units of each other vertically.
- Points p_L and p_R must be within a $\delta \times 2\delta$ rectangle centered at line l .
- At most 8 points of P can reside within the $\delta \times 2\delta$ rectangle since these points are at least δ units apart.
- Even if p_L occurs as early as possible in Y' and p_R as late as possible, p_R is in one of the 7 positions following p_L .

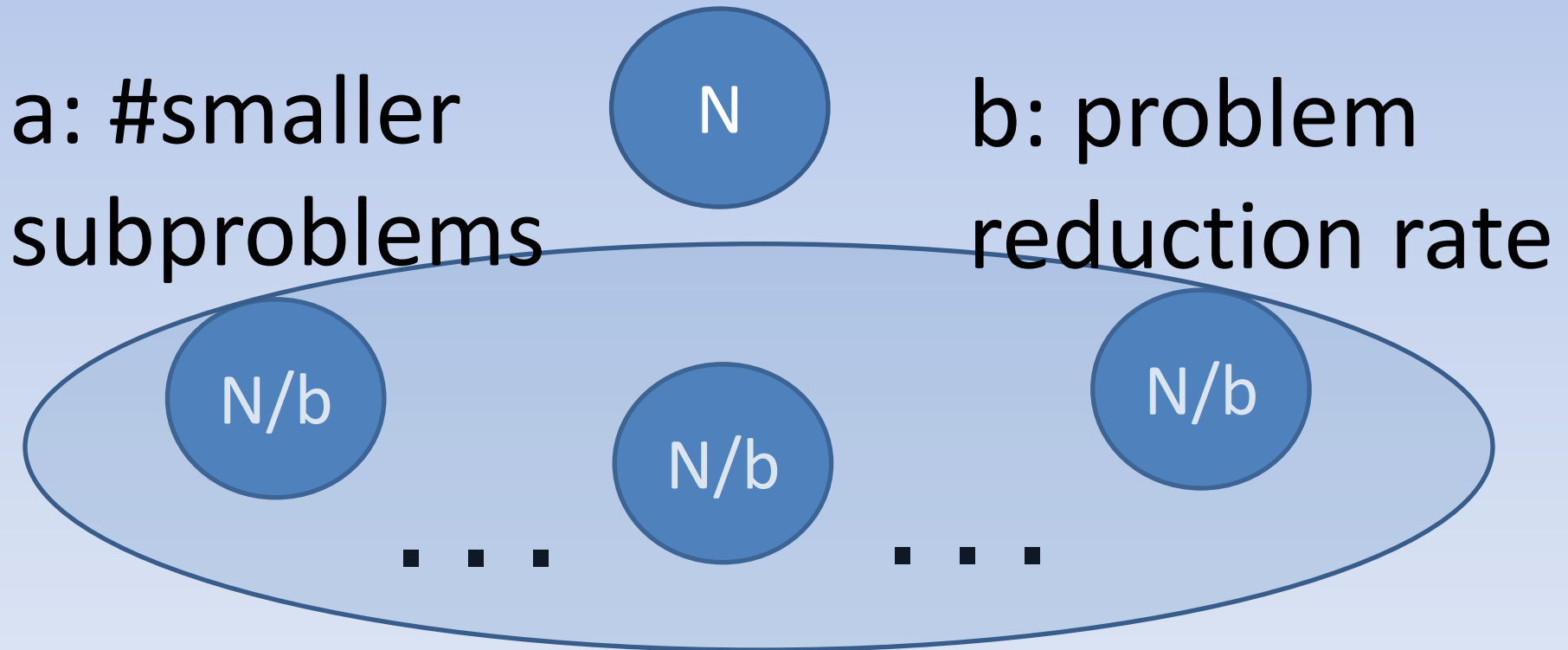
Running Time

- Goal Recurrence was $T(n) = 2T(n/2) + O(n)$
- Main challenge is sorted arrays X_L , X_R , Y_L , and Y_R
- Method used can be viewed as the opposite of the Merge procedure of merge sort.

Solving Recurrences

Solving Recurrences w/ Master Method

- Perhaps we can come up with some general formula for recurrences:



Solving Recurrences w/ Master Method

- Perhaps we can come up with some general formula for recurrences:
- $T(N) = aT(N/b) + O(N^d)$
 - N^d work down to merge subproblems!
 - Book based around $n^{\log_b a}$
 - Look at relationship between $\log_b a$ and d

Solving Recurrences w/ Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{array}{ll} O(n^d \log n) & \text{if } a = b^d \text{ (Case 1)} \\ O(n^d) & \text{if } a < b^d \text{ (Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (Case 3)} \end{array}$$

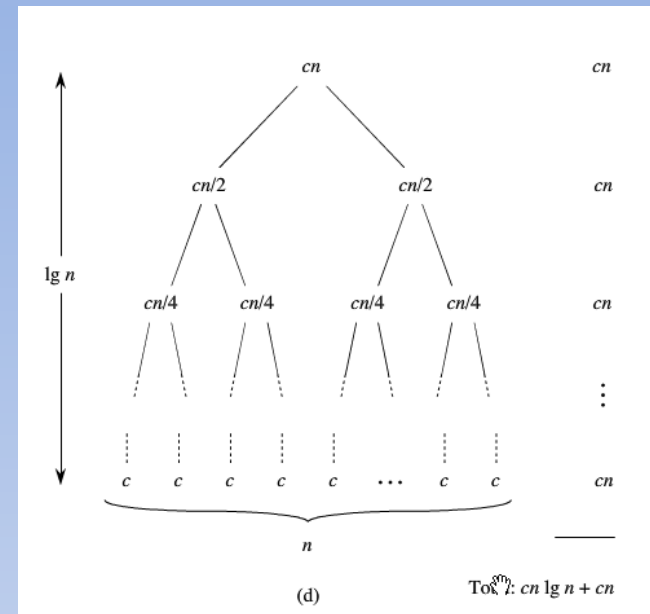
Solving Recurrences w/ Master Method

- Perhaps we can come up with some general formula for recurrences:
- $T(N) = aT(N/b) + O(N^d)$
 - N^d work down to merge subproblems!
- 3 Cases to think about!

Master Method

Case 1

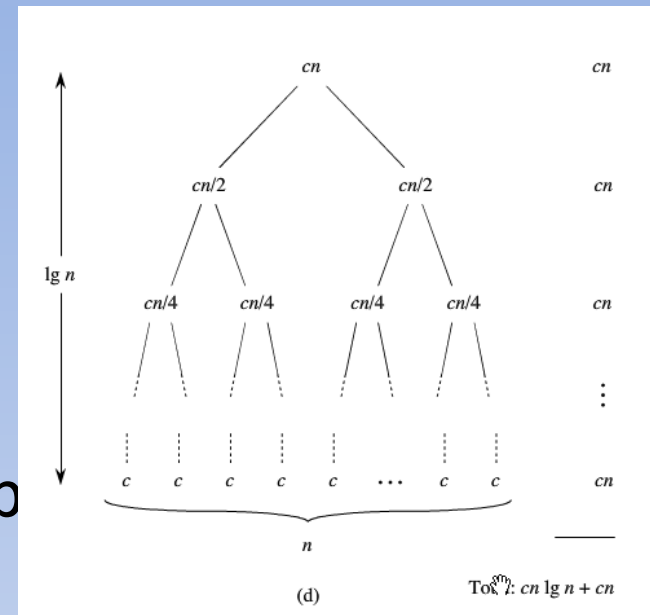
- $T(N) = aT(N/b) + O(N^d)$
 - N^d work down to merge subproblems!
- Assume $a = b^d$
 - a (or equivalently b^d) subproblems each level
 - Each subproblem has size $\frac{N}{b^{\text{depth}}}$ and there are a^{depth}
 - if $d=1$, then $a=b$!!!
 - n^d work to merge
 - $(\frac{N}{b^{\text{depth}}})^d$
- $O(N^d \log N)$
 - Balance between work at each level and depth
- Example:
 - $a = 2, b=2, d=1$
 - $b^d = 2^1 = 2 = a$
 - $N \lg_2 N$



Master Method

Case 1

- $T(N) = aT(N/b) + O(N^d)$
 - N^d work down to merge subproblems



- Assume $a = b^d$
 - a (or equivalently b^d) subproblems each level
 - Each subproblem has size $\frac{N}{b^{\text{depth}}}$ and there are a^{depth}
 - if $d=1$, then $a=b$!!!
 - n^d work to merge at each level
 - $(\frac{N}{b^{\text{depth}}})^d a^{\text{depth}} = N^d (\frac{a}{b^d})^{\text{depth}}$

Solving Recurrences w/ Master Method

- $\text{Work}(\text{level}_j) \leq a^j * c(N/b^j)^d$
 - a^j : This is the number of nodes at level j
 - $c(N/b^j)^d$: Each of the nodes do this much work
 - (N^d) where N is reduced to N/b^j
- Levels = Depth = $\log_b N$

$$\text{Total Work} = cN^d \cdot \sum_{j=0}^{\log_b N} \left(\frac{a}{b^d}\right)^j$$

- NOTE: $a = b^d$ means $a/b^d = 1$
 - Master Method case 1: $cN^d \cdot \log_b N = O(N^d \log N)$

Master Method w/ Case 2

- $T(N) = aT(N/b) + O(N^d)$
 - N^d work down to merge subproblems!
- Assume $a < b^d$
 - Number of subproblems is growing slowly
- $O(N^d)$
 - Total work dominated by the combine step at root!
 - Good (subproblem simplification) is beating evil (subproblem proliferation)!
- $T(N) = 2T(N/2) + O(n^2)$
 - Less work is being done at each level
 - Work at level 2 is $2T(N/2) = 2(N/2)^2 = 2(N^2/4) = \mathbf{N^2/2}$
 - If 4 problem were being created: $4 = b^d$ so $4(N^2/4) = \mathbf{N^2}$ at each level!
- $a = 2, b=2, d=2$
 - $a = 2 < b^d = 2^2 = 4$
- Work in total is dominated by root!
 - $O(N^2)$

Master Method w/ Case 3

- $T(N) = aT(N/b) + O(N^d)$
 - N^d work down to merge subproblems!
- Assume $a > b^d$
- $T(N) = 4T(N/2) + O(N)$
 - More work is being done at each level
 - Level 2 is now $4(N/2) = \mathbf{2N}$
 - Each level now has more work!
- $a = 4$ $b=2$, $d=1$
 - $a = 4 > b^d = 2^1 = 2$
- Work down at the leaves is overwhelming all other levels.
- How many leaves?
 - Next Slide

Master Method w/ Case 3

- Assume $a > b^d$
- $T(N) = 4T(N/2) + O(n)$
 - More work is being done at each level
- $a = 4$ $b=2$, $d=1$
 - $a = 4 > b^d = 2^1 = 2$
- How many leaves
 - Each node has 4 children
 - 4^{depth} number of leaves
- Depth = $\log_b N$
 - $4^{\log_2 N} = N^{\log_2 4}$

Master Method w/ Case 3

- Depth = $\log_b N$
 - $4^{\log_2 N} = N^{\log_2 4}$

$$4^{\log_2 N} = N^{\log_2 4}$$

$$\log_2(4^{\log_2 N}) = \log_2(N^{\log_2 4})$$

$$\log_2 N \log_2 4 = \log_2 4 \log_2 N$$

- $4^{\log_2 N} = N^{\log_2 4} = N^2$

Solving Recurrences w/ Master Method Case 2

$$Total\ Work = cN^d \cdot \sum_{j=0}^{j=\log_b N} \left(\frac{a}{b^d}\right)^j$$

- IF: $a < b^d$ means $a/b^d < 1$
- SO: $(a/b^d)^j$ is getting smaller
- SO Asymptotically : Total work dominated by root where $j=0$ and $T(N) = cN^d$

Solving Recurrences w/ Master Method Case 3

$$Total\ Work = cN^d \cdot \sum_{j=0}^{j=\log_b N} \left(\frac{a}{b^d}\right)^j$$

- Finally: $a > b^d$ means $a/b^d > 1$
- SO: $(a/b^d)^j$ is growing exponentially!
- Total work is going to be dominated asymptotically by leaves:

$$Total\ Work \leq cN^d \cdot \left(\frac{a}{b^d}\right)^{\log_b N}$$

Solving Recurrences w/ Master Method Case 3

$$\text{Total Work} \leq cN^d \cdot \left(\frac{a}{b^d}\right)^{\log_b N}$$

$$cN^d \cdot \left(\frac{a}{b^d}\right)^{\log_b N} = cN^d \cdot a^{\log_b N} \cdot b^{-d \log_b N}$$

$$b^{-d \log_b N} = N^{-d}$$

$$cN^d \cdot a^{\log_b N} \cdot b^{-d \log_b N} = c \frac{N^d}{N^d} \cdot a^{\log_b N}$$

$$O(a^{\log_b N})$$

Solving Recurrences w/ Master Method Case 3

Total Work =

$$O(a^{\log_b n}) = O(n^{\log_b a})$$

$$\log_b(a^{\log_b n}) = \log_b(n^{\log_b a})$$

$$\log_b n \cdot \log_b a = \log_b a \cdot \log_b n$$

$$\text{Total Work} = O(n^{\log_b a})$$

Textbook

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

- Consider $f(n) = n^d$
- Case 1: $n^d < cn^{\lg_b a}$
 - $\log(n^d) < \log(n^{\lg_b a})$
 - $d < \lg_b a$
 - $b^d < b^{\lg_b a}$
 - $b^d < a$
- $\Theta(n^{\log_b a})$
 - Case 3 from earlier!
 - Work dominated by leaves!!

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

- Consider $f(n) = n^d$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$
- $n^d > cn^{\log_b a + \epsilon}$
 - $\log(n^d) > \log(n^{\log_b a + \epsilon}) \geq \log(n^{\log_b a})$
 - $d > \lg_b a$
 - $b^d > b^{\lg_b a}$
 - $b^d > a$
- $\Theta(f(n)) = O(n^d)$

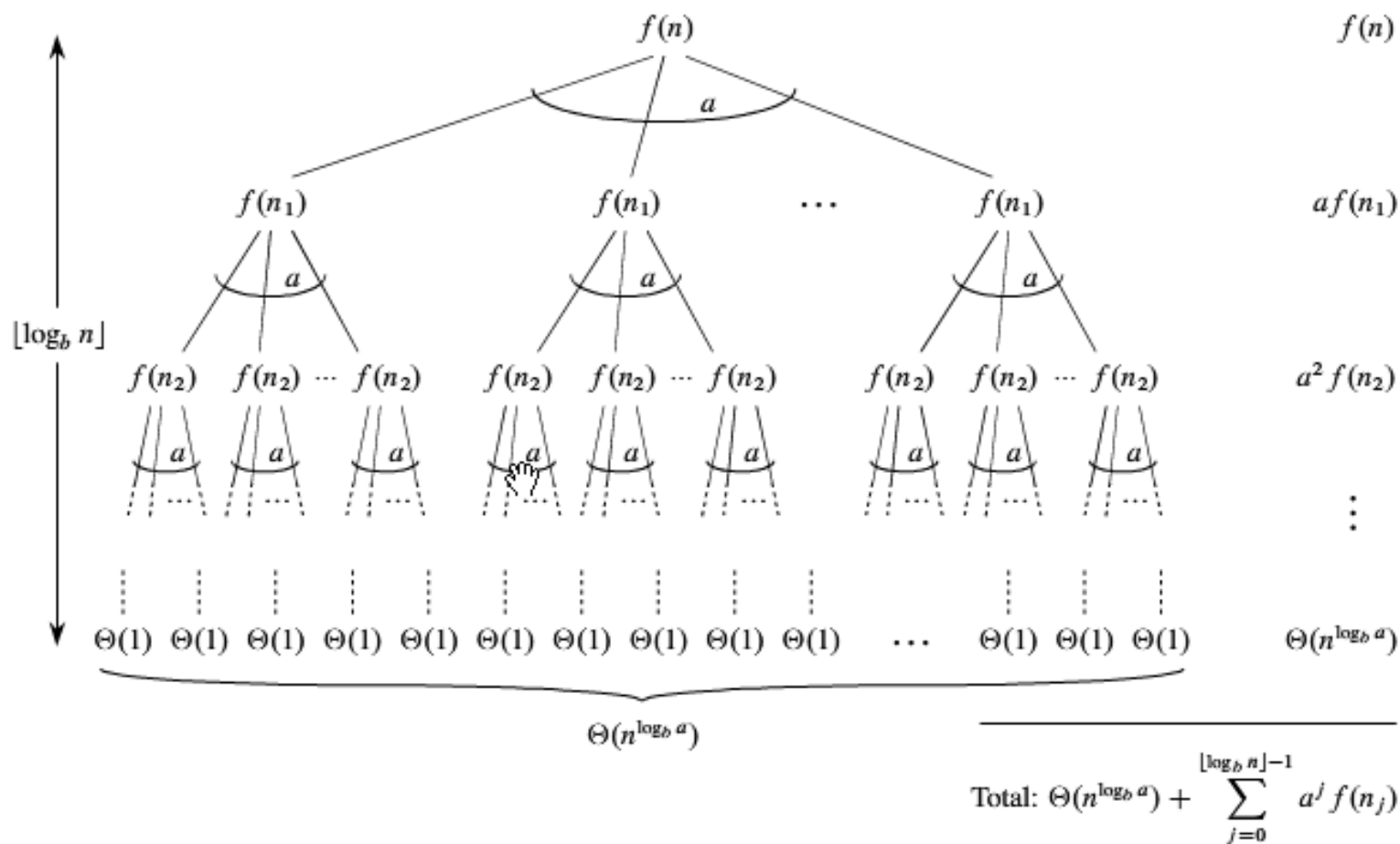


Figure 4.8 The recursion tree generated by $T(n) = aT(\lceil n/b \rceil) + f(n)$. The recursive argument n_j is given by equation (4.27).

Solving Recurrences w/ Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{array}{ll} O(n^d \log n) & \text{if } a = b^d \text{ (Case 1)} \\ O(n^d) & \text{if } a < b^d \text{ (Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (Case 3)} \end{array}$$

31	Number-Theoretic Algorithms	926
31.1	Elementary number-theoretic notions	927
31.2	Greatest common divisor	933
31.3	Modular arithmetic	939
31.4	Solving modular linear equations	946
31.5	The Chinese remainder theorem	950
31.6	Powers of an element	954
31.7	The RSA public-key cryptosystem	958
★	31.8 Primality testing	965
★	31.9 Integer factorization	975
32	String Matching	985
32.1	The naive string-matching algorithm	988
32.2	The Rabin-Karp algorithm	990
32.3	String matching with finite automata	995
★	32.4 The Knuth-Morris-Pratt algorithm	1002
33	Computational Geometry	1014
33.1	Line-segment properties	1015
33.2	Determining whether any pair of segments intersects	102
33.3	Finding the convex hull	1029
33.4	Finding the closest pair of points	1039
34	NP-Completeness	1048
34.1	Polynomial time	1053
34.2	Polynomial-time verification	1061
34.3	NP-completeness and reducibility	1067
34.4	NP-completeness proofs	1078
34.5	NP-complete problems	1086
35	Approximation Algorithms	1106
35.1	The vertex-cover problem	1108
35.2	The traveling-salesman problem	1111
35.3	The set-covering problem	1117
35.4	Randomization and linear programming	1123
35.5	The subset-sum problem	1128

Algorithm

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)sm

```
1   $d_1 = \text{DIRECTION}(p_3, p_4, p_1)$ 
2   $d_2 = \text{DIRECTION}(p_3, p_4, p_2)$ 
3   $d_3 = \text{DIRECTION}(p_1, p_2, p_3)$ 
4   $d_4 = \text{DIRECTION}(p_1, p_2, p_4)$ 
5  if  $((d_1 > 0 \text{ and } d_2 < 0) \text{ or } (d_1 < 0 \text{ and } d_2 > 0)) \text{ and}$   
     $((d_3 > 0 \text{ and } d_4 < 0) \text{ or } (d_3 < 0 \text{ and } d_4 > 0))$ 
6      return TRUE
7  elseif  $d_1 == 0 \text{ and } \text{ON-SEGMENT}(p_3, p_4, p_1)$ 
8      return TRUE
9  elseif  $d_2 == 0 \text{ and } \text{ON-SEGMENT}(p_3, p_4, p_2)$ 
10     return TRUE
11 elseif  $d_3 == 0 \text{ and } \text{ON-SEGMENT}(p_1, p_2, p_3)$ 
12     return TRUE
13 elseif  $d_4 == 0 \text{ and } \text{ON-SEGMENT}(p_1, p_2, p_4)$ 
14     return TRUE
15 else return FALSE
```

Algorithm

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)

DIRECTION(p_i, p_j, p_k)

1 **return** $(p_k - p_i) \times (p_j - p_i)$

ON-SEGMENT(p_i, p_j, p_k)

1 **if** $\min(x_i, x_j) \leq x_k \leq \max(x_i, x_j)$ and $\min(y_i, y_j) \leq y_k \leq \max(y_i, y_j)$

2 **return** TRUE

3 **else return** FALSE

4 **if** $d_1 == 0$ and ON-SEGMENT(p_3, p_4, p_1)

5 **return** TRUE

6 **elseif** $d_2 == 0$ and ON-SEGMENT(p_3, p_4, p_2)

7 **return** TRUE

8 **elseif** $d_3 == 0$ and ON-SEGMENT(p_1, p_2, p_3)

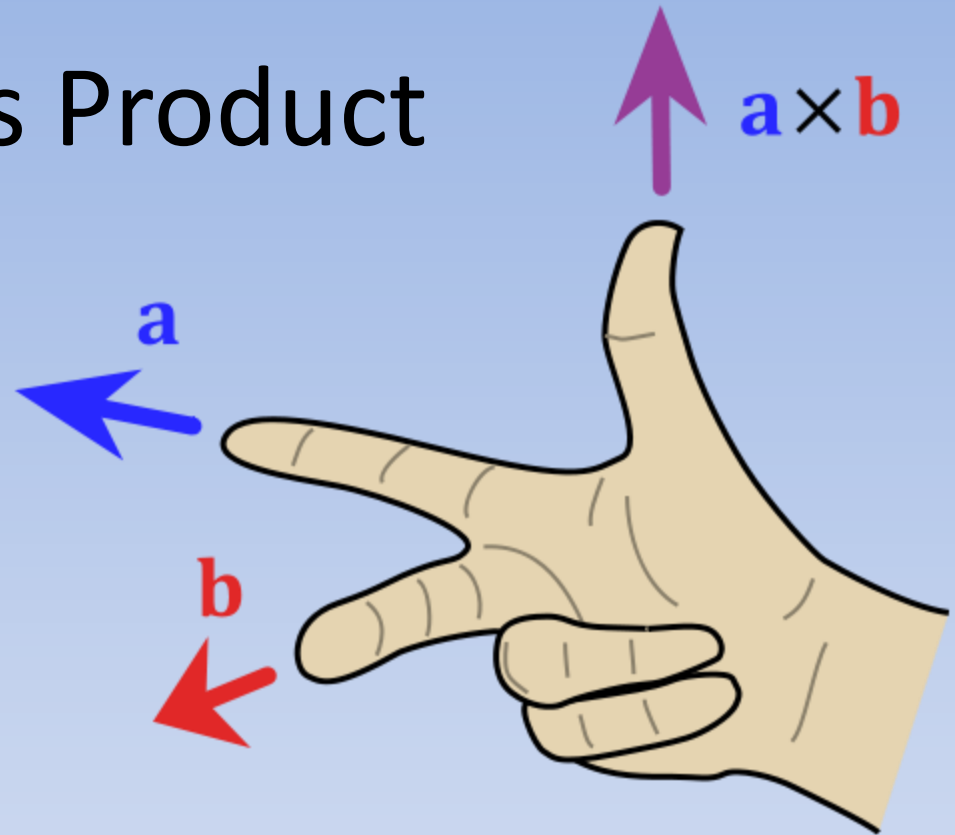
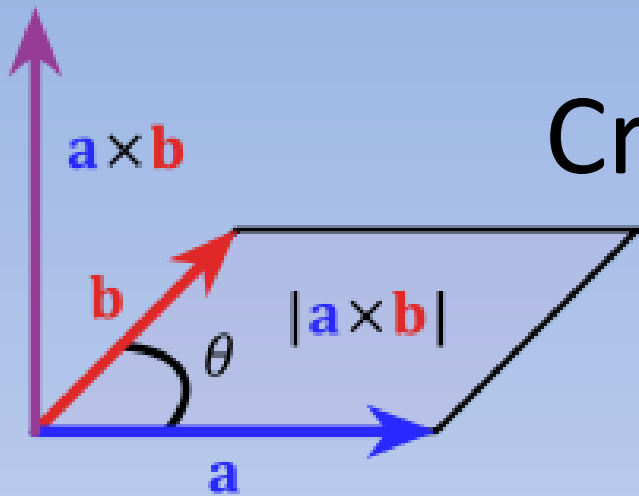
9 **return** TRUE

10 **elseif** $d_4 == 0$ and ON-SEGMENT(p_1, p_2, p_4)

11 **return** TRUE

12 **else return** FALSE

Cross Product



- Cross Product of two vectors \mathbf{p}_1 and \mathbf{p}_2 is:
 - Vector perpendicular to \mathbf{p}_1 and \mathbf{p}_2 according to the “right-hand rule”
 - Magnitude of vector is $|\mathbf{p}_1 \times \mathbf{p}_2|$

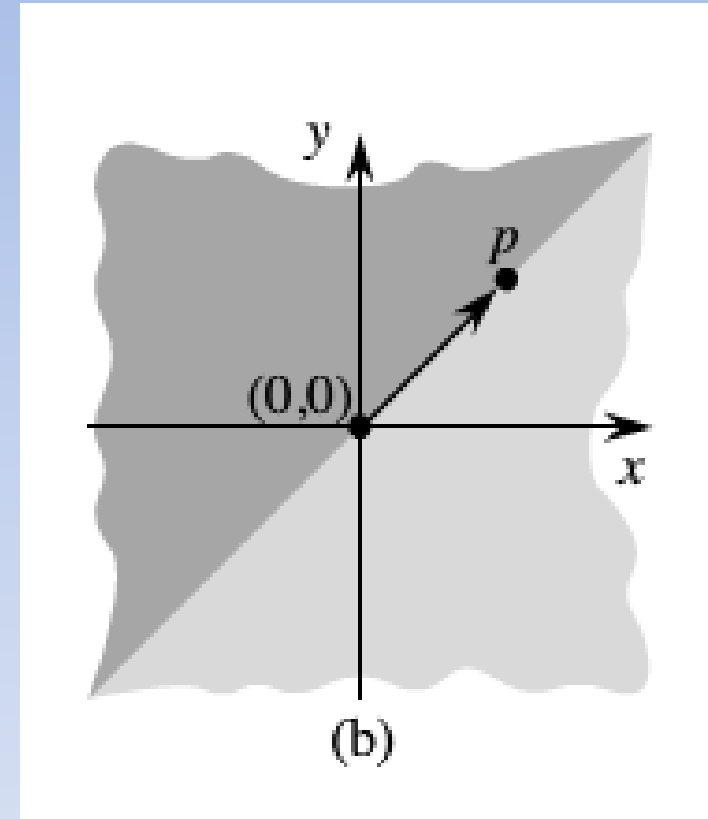
Cross Product as Determinant

- $p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$
 $= x_1 y_2 - x_2 y_1$
 $= -p_2 \times p_1$
- For our Purposes: treat cross product as value
 $x_1 y_2 - x_2 y_1$

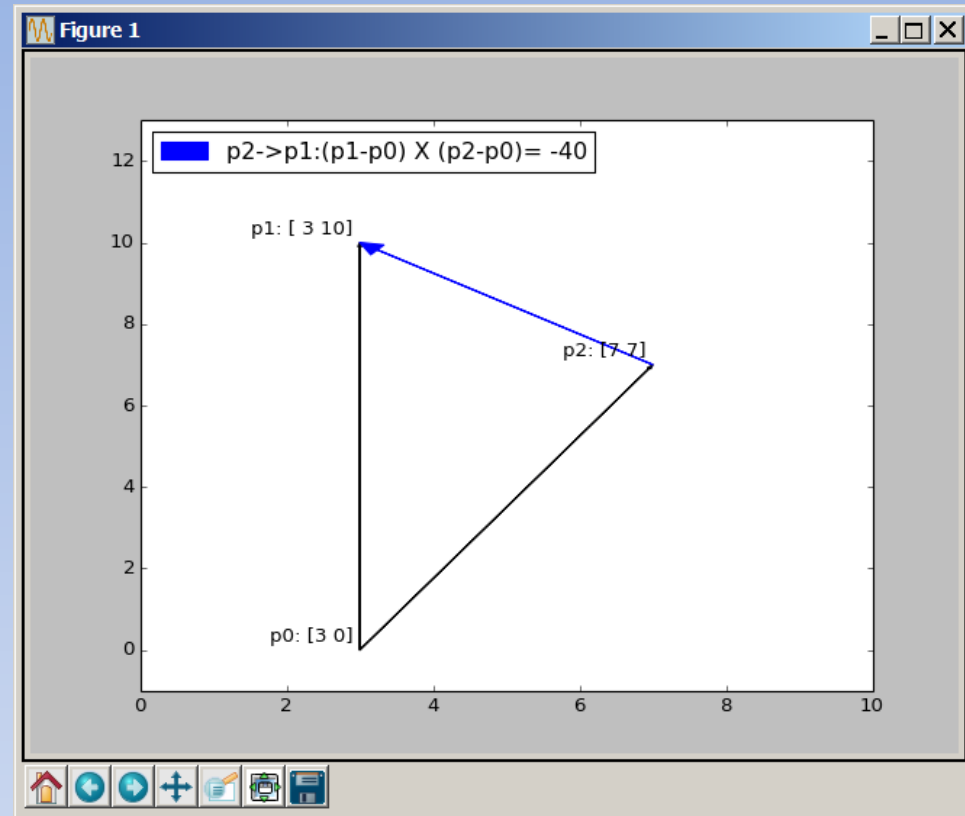
Sign of the Crossproduct:

Clockwise versus Counterclockwise

- Sign of the Crossproduct $p_1 \times p_2$ is positive, then p_1 is counterclockwise from p_2 .
- Sign of the Crossproduct $p_1 \times p_2$ is 0, then p_1 and p_2 are colinear.
- Dark Region contains vectors that are CounterClockwise with respect to P
 - Sign of the Crossproduct Negative



Clockwise Versus Counterclockwise



- Determine:
 - With respect to a common end point p_0
 - is $\overrightarrow{p_0 p_1}$ is closer to $\overrightarrow{p_0 p_2}$ in a clockwise or counterclockwise direction

Clockwise Versus Counterclockwise

- Translate to use p_0 as origin:
 - $p_1 - p_0$ denotes vector $p'_1 = (x'_1, y'_1)$
 - $x'_1 = x_1 - x_0$
 - $y'_1 = y_1 - y_0$
 - $p_2 - p_0$ defined similarly
- Compute Cross Product:
 - $(p_1 - p_0) \times (p_2 - p_0)$
 - $= (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$
- If Cross Product :
 - Positive: $\overrightarrow{p_0 p_1}$ is clockwise from $\overrightarrow{p_0 p_2}$
 - Negative: $\overrightarrow{p_0 p_1}$ is *counterclockwise* from $\overrightarrow{p_0 p_2}$

Cross Product: Left Turn or Right Turn

33.1 Line-segment properties

1017

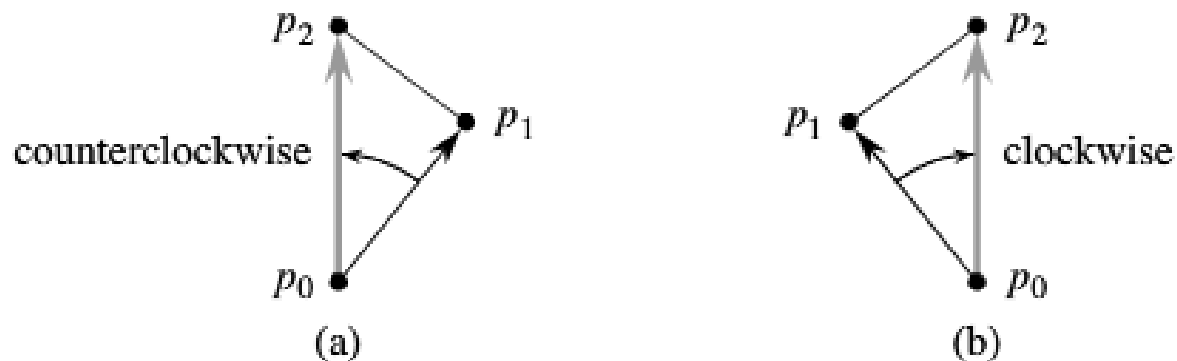
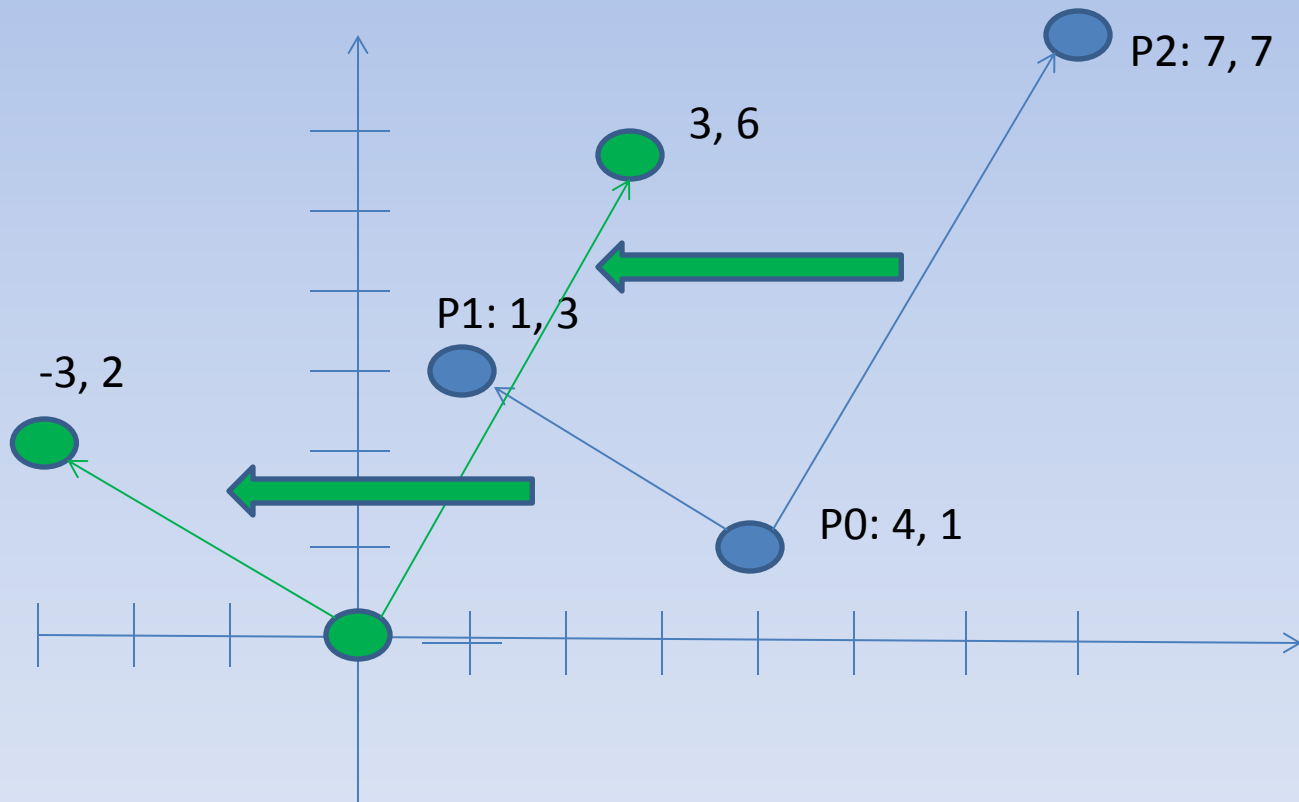


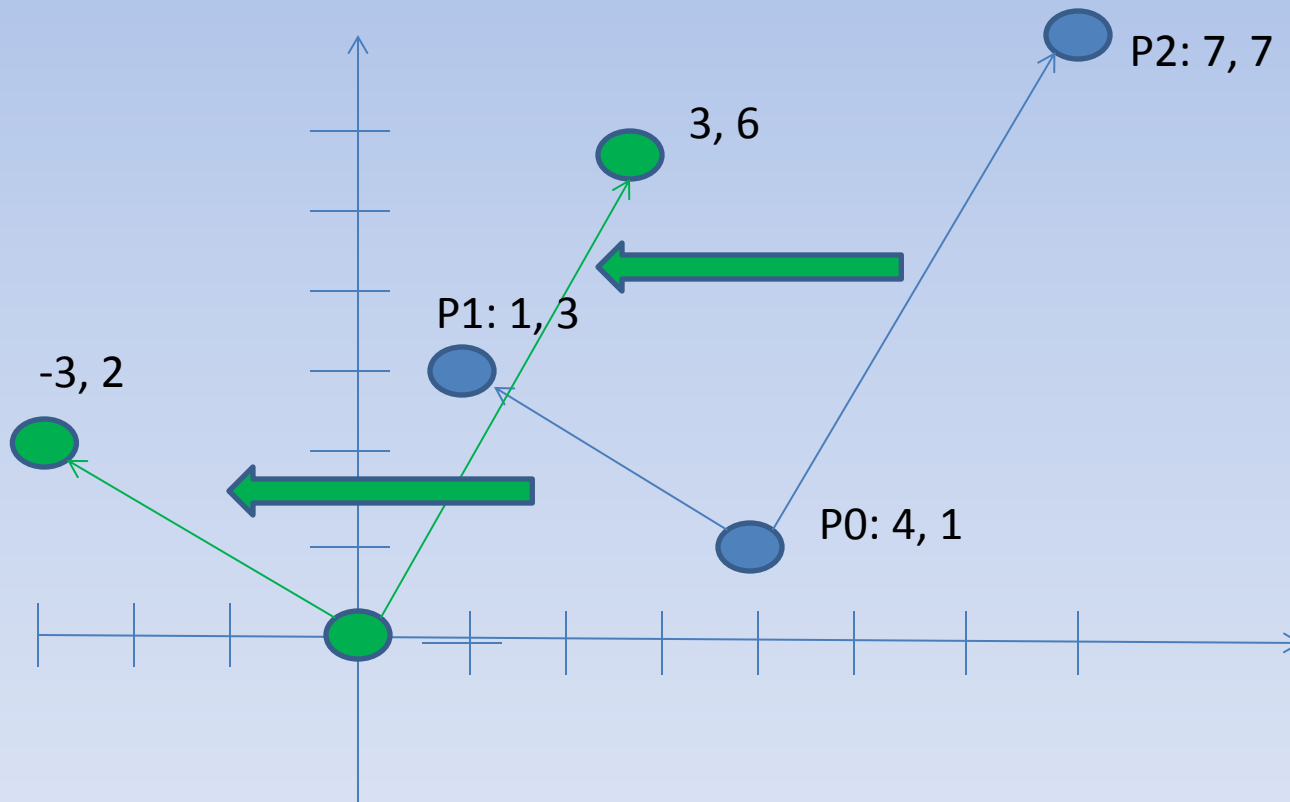
Figure 33.2 Using the cross product to determine how consecutive line segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{p_1 p_2}$ turn at point p_1 . We check whether the directed segment $\overrightarrow{p_0 p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0 p_1}$. (a) If counterclockwise, the points make a left turn. (b) If clockwise, they make a right turn.

- Cross products answer query without angles!

Move Vectors to Origin

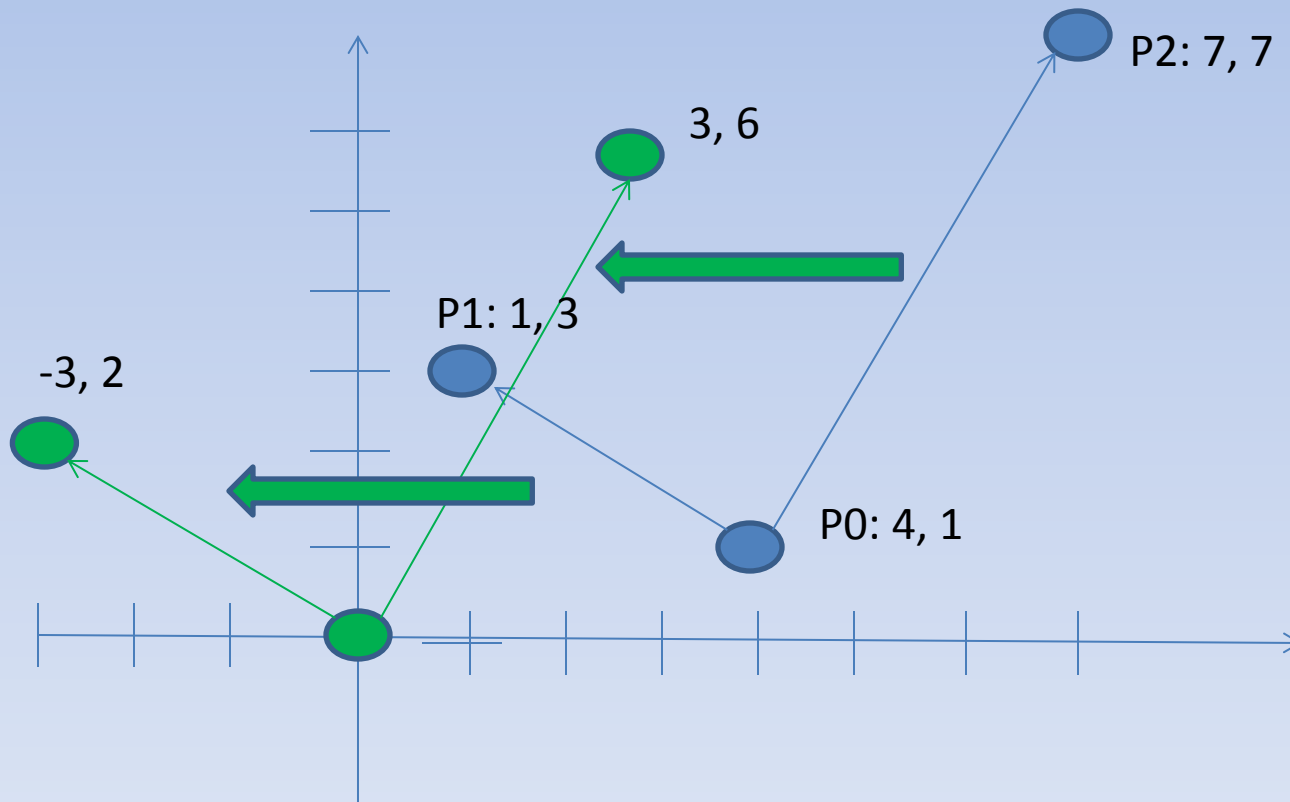


Sign of the Crossproduct Calculated from Origin



return $(-3,2) \times (3,6) = (-3*6) - (2*3) = -18 - 6 = -24$

Sign of the Crossproduct Calculated from Origin



return (3,6) X (-3,2) = (3*2) - (6*(-3)) = 6 - (-18) = 24

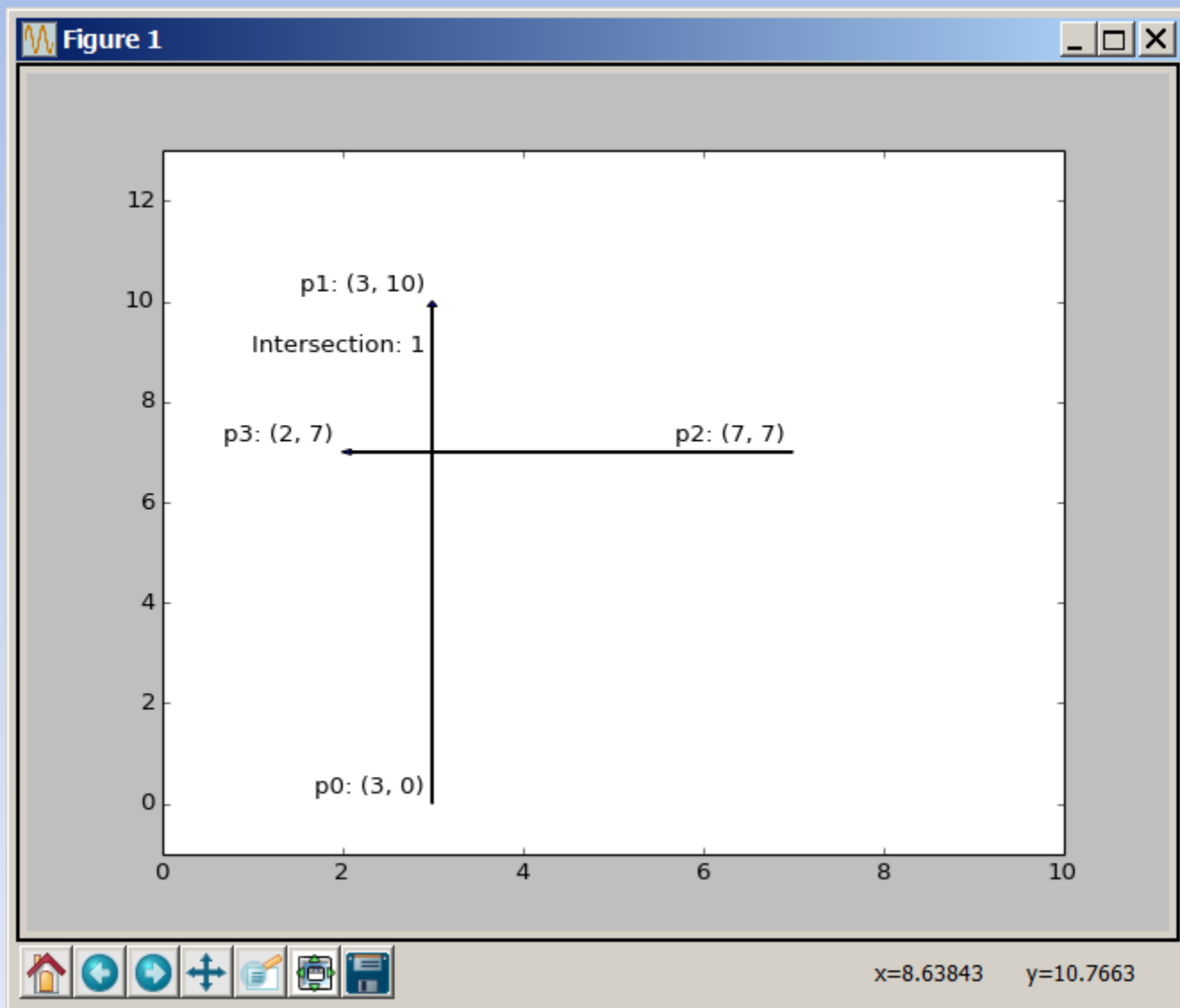
Next Steps

- Now we want to use this clockwise/counterclockwise info to determine line segment intersection!
- Determining if Line Segs Intersect define **STRADLE**.

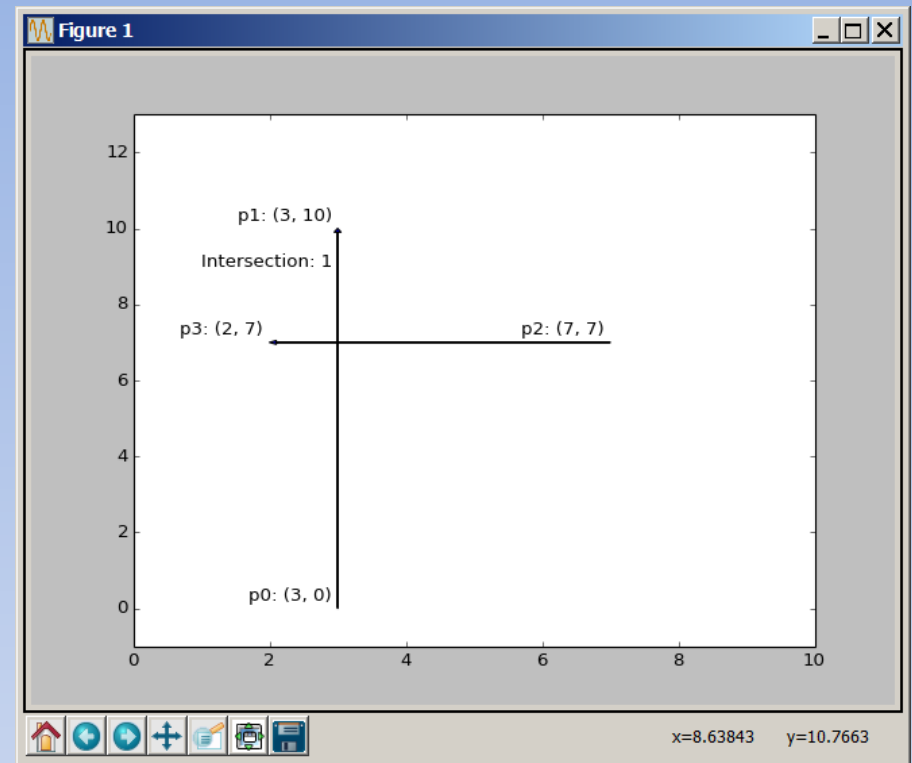
Straddling & Intersections

- Determining if Line Segs Intersect define **STRADLE**.
- STRADLE: A segment $\overline{p_1p_2}$ **straddles** a line if :
 - Point p_1 lies on one side of the line
 - Point p_2 lies on the other side of the line
- Two line segments INTERSECT if and only if either (or both) of the following conditions holds :
 1. Each segment straddles the line containing the other.
 2. An endpoint of one segment lies on the other segment.

Straddling

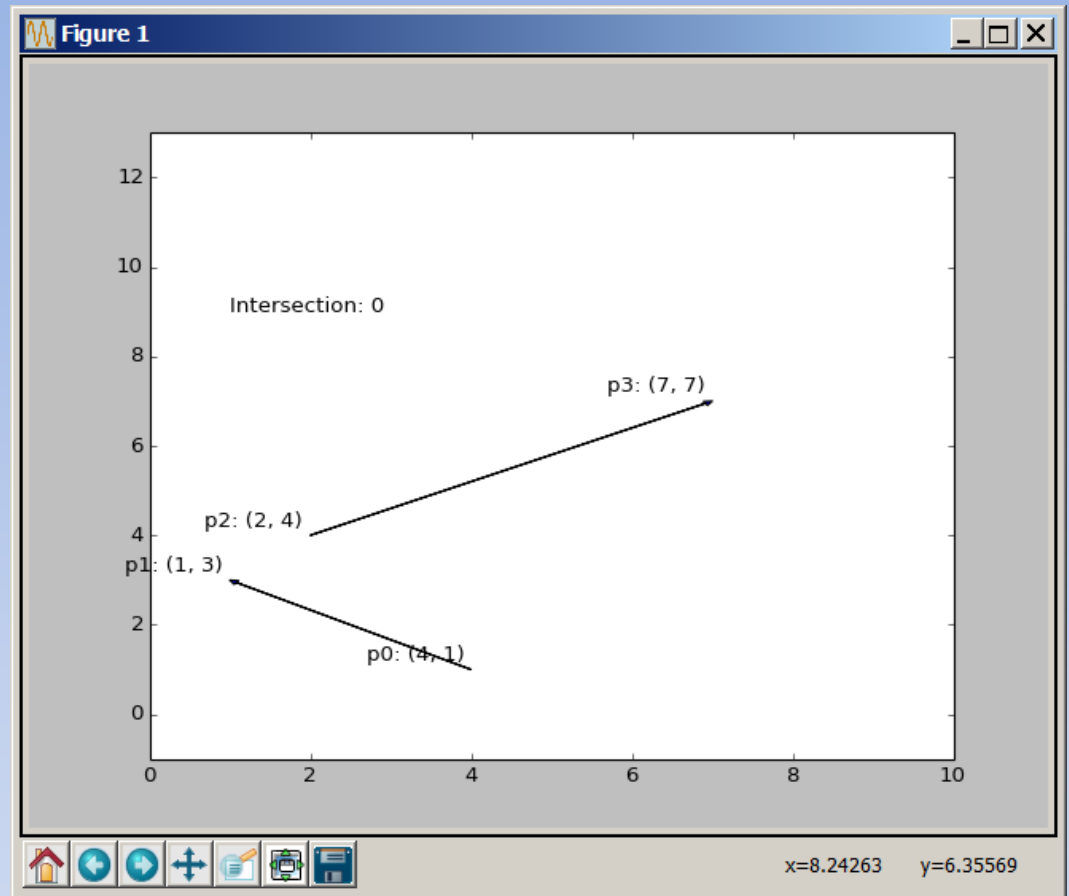


Straddling



- Line 1: p3: (2, 7), p2: (7, 7)
- Line 2: p0: (3, 0), p1: (3, 10)

Not Straddling



- Line 1: p2:(2,4), p3:(7,7)
- Line 2: p0:(4,1), p1:(1,3)
 - Points p2 & p3 are both clockwise of line segment p0,p1!

Algorithm

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)

```
SEGMENTS-INTERSECT( $p_1, p_2, p_3, p_4$ )sm
1   $d_1 = \text{DIRECTION}(p_3, p_4, p_1)$ 
2   $d_2 = \text{DIRECTION}(p_3, p_4, p_2)$ 
3   $d_3 = \text{DIRECTION}(p_1, p_2, p_3)$ 
4   $d_4 = \text{DIRECTION}(p_1, p_2, p_4)$ 
5  if  $((d_1 > 0 \text{ and } d_2 < 0) \text{ or } (d_1 < 0 \text{ and } d_2 > 0)) \text{ and}$   

       $((d_3 > 0 \text{ and } d_4 < 0) \text{ or } (d_3 < 0 \text{ and } d_4 > 0))$ 
6    return TRUE
7  elseif  $d_1 == 0 \text{ and } \text{ON-SEGMENT}(p_3, p_4, p_1)$ 
8    return TRUE
9  elseif  $d_2 == 0 \text{ and } \text{ON-SEGMENT}(p_3, p_4, p_2)$ 
10   return TRUE
11 elseif  $d_3 == 0 \text{ and } \text{ON-SEGMENT}(p_1, p_2, p_3)$ 
12   return TRUE
13 elseif  $d_4 == 0 \text{ and } \text{ON-SEGMENT}(p_1, p_2, p_4)$ 
14   return TRUE
15 else return FALSE
```

Algorithm

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)

DIRECTION(p_i, p_j, p_k)

1 **return** $(p_k - p_i) \times (p_j - p_i)$

ON-SEGMENT(p_i, p_j, p_k)

1 **if** $\min(x_i, x_j) \leq x_k \leq \max(x_i, x_j)$ and $\min(y_i, y_j) \leq y_k \leq \max(y_i, y_j)$

2 **return** TRUE

3 **else return** FALSE

4 **if** $d_1 == 0$ and ON-SEGMENT(p_3, p_4, p_1)

5 **return** TRUE

6 **elseif** $d_2 == 0$ and ON-SEGMENT(p_3, p_4, p_2)

7 **return** TRUE

8 **elseif** $d_3 == 0$ and ON-SEGMENT(p_1, p_2, p_3)

9 **return** TRUE

10 **elseif** $d_4 == 0$ and ON-SEGMENT(p_1, p_2, p_4)

11 **return** TRUE

12 **else return** FALSE

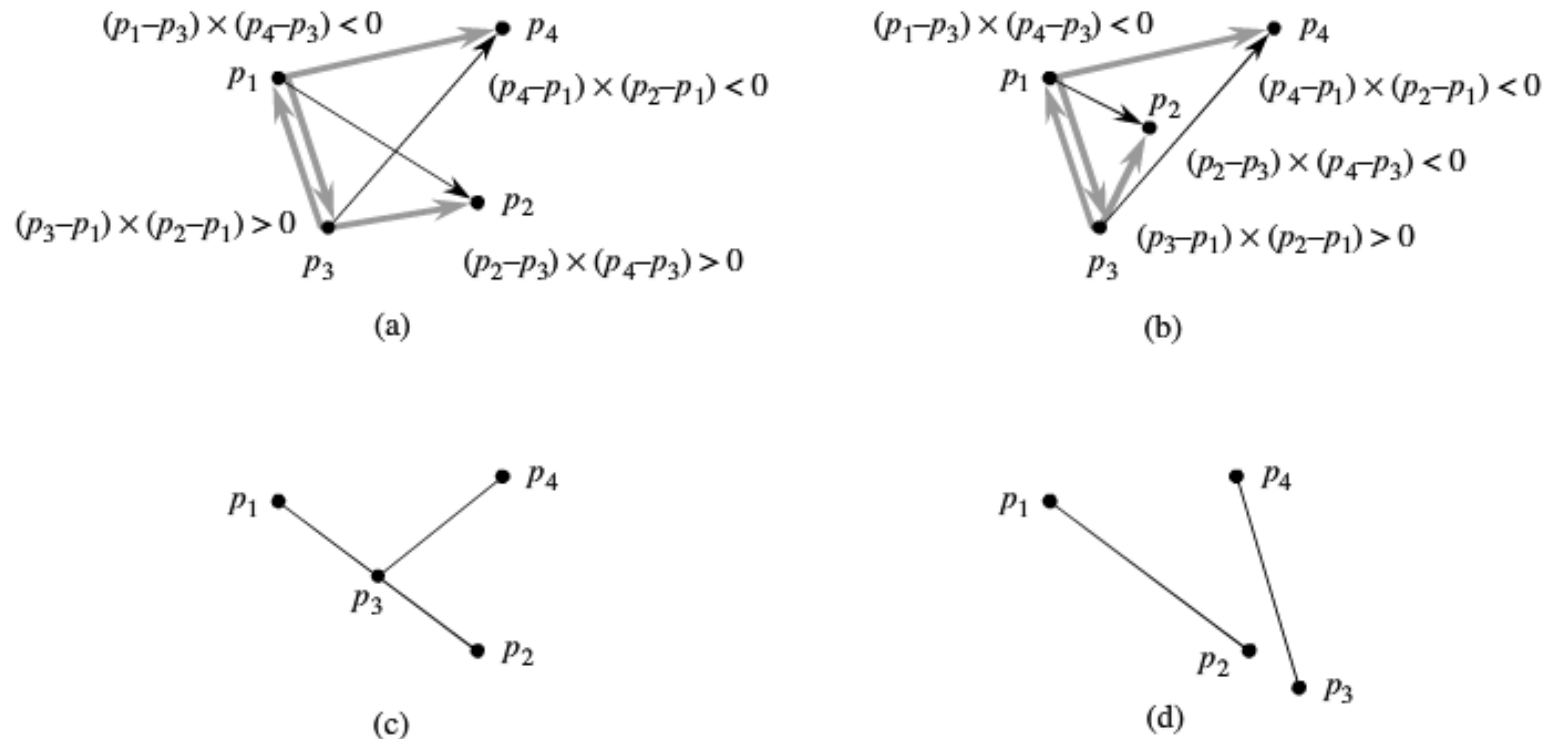
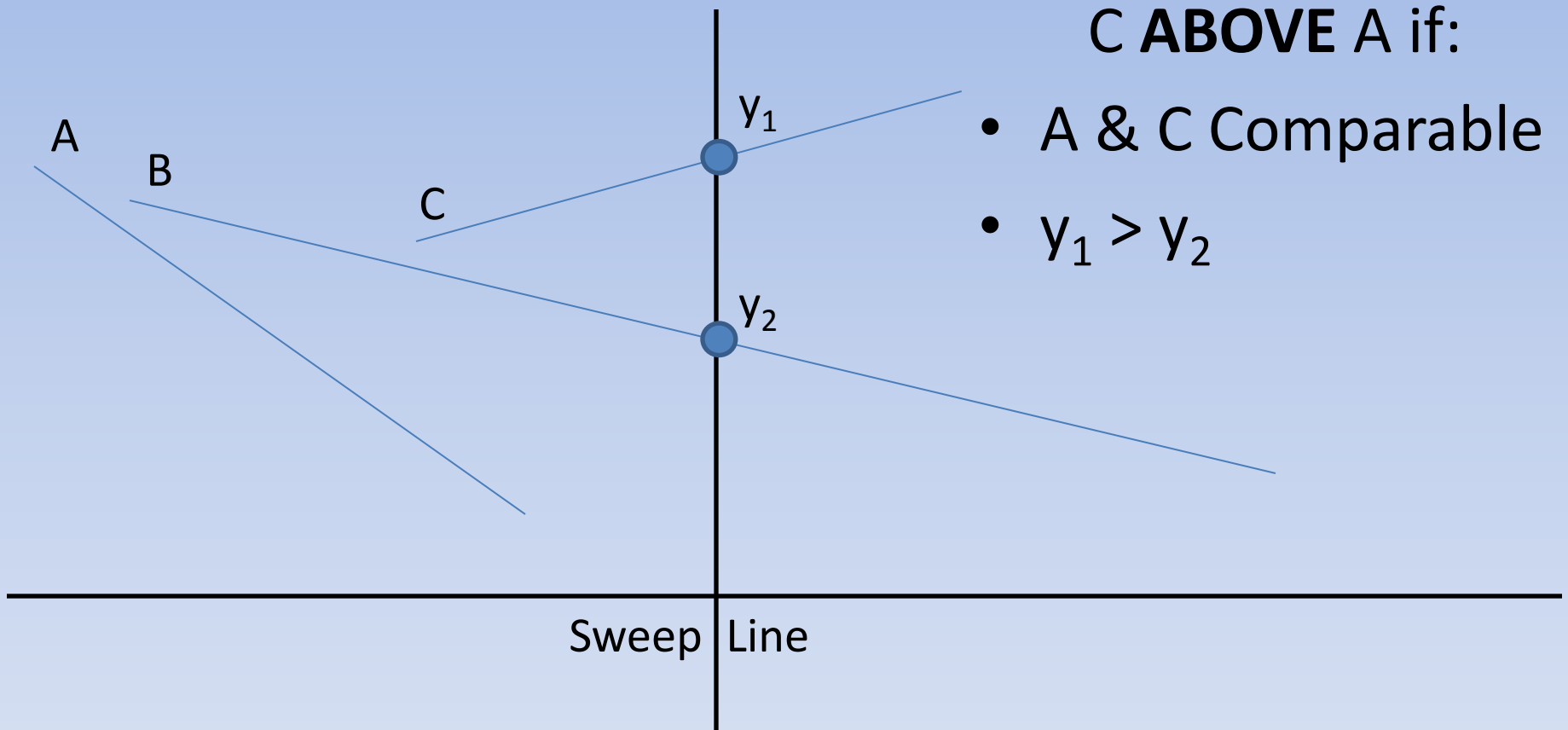


Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. (a) The segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ straddle each other's lines. Because $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1 p_2}$ straddles the line containing $\overline{p_3 p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. (b) Segment $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, but $\overline{p_1 p_2}$ does not straddle the line containing $\overline{p_3 p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. (c) Point p_3 is colinear with $\overline{p_1 p_2}$ and is between p_1 and p_2 . (d) Point p_3 is colinear with $\overline{p_1 p_2}$, but it is not between p_1 and p_2 . The segments do not intersect.

Query Any pair of segments Intersects w/ **Sweeping**

- Determines If ANY Intersecting line segments in $O(N \lg N)$ time.
 - N is the number of segments.
- **Sweeping:**
 - Imaginary vertical **Sweep Line** passes through objects.
 - Sweep dimension treated as time dimension
- Sweeping allows ordering the geometric objects.

Ordering Segments



- No Vertical Lines
- Line Segments **Comparable** if both intersect Sweep Line.

Sweep Line

33.2 Determining whether any pair of segments intersects

1023

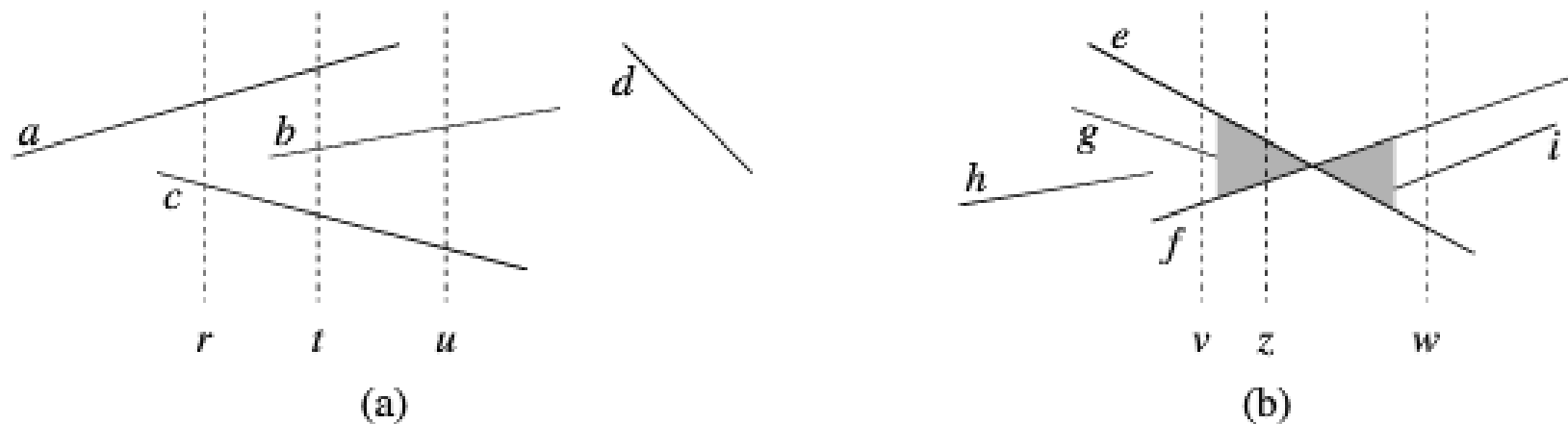


Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a \succ_r c$, $a \succ_t b$, $b \succ_t c$, $a \succ_t c$, and $b \succ_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, they reverse their orders: we have $e \succ_v f$ but $f \succ_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in the ordering given by the relation \succ_z .

Moving a Sweep Line

- Manage Two Sets of Data:
 - **Sweep-Line Status:** Ordering induced among objects intersecting sweep line.
 - **Event-Point Schedule:** This sequence of **event-points** are ordered left to right according to the x-coordinates, and mark the points where sweeping halts and processing takes place.

Maintaining Sweep-Line Status

- Sweep-Line Status Data Structure maintains a complete preorder of a set of line segments.
- Sweep-Line Status Data Structure Operations
 - $\text{Insert}(T, s)$: inserts segment s into T .
 - $\text{Delete}(T, s)$: delete segment s from T .
 - $\text{Above}(T, s)$: returns the segment immediately above segment s in T .
 - $\text{Below}(T, s)$: return the segment immediately below segment s in T .
- A balanced binary tree (avl, red-black) can implements ops in $O(\ln N)$.

ANY-SEGMENTS-INTERSECT(S)

```
1   $T = \emptyset$ 
2  sort the endpoints of the segments in  $S$  from left to right,
   breaking ties by putting left endpoints before right endpoints
   and breaking further ties by putting points with lower
   y-coordinates first
3  for each point  $p$  in the sorted list of endpoints
4      if  $p$  is the left endpoint of a segment  $s$ 
5          INSERT( $T, s$ )
6          if (ABOVE( $T, s$ ) exists and intersects  $s$ )
              or (BELOW( $T, s$ ) exists and intersects  $s$ )
7              return TRUE
8      if  $p$  is the right endpoint of a segment  $s$ 
9          if both ABOVE( $T, s$ ) and BELOW( $T, s$ ) exist
              and ABOVE( $T, s$ ) intersects BELOW( $T, s$ )
10             return TRUE
11             DELETE( $T, s$ )
12 return FALSE
```

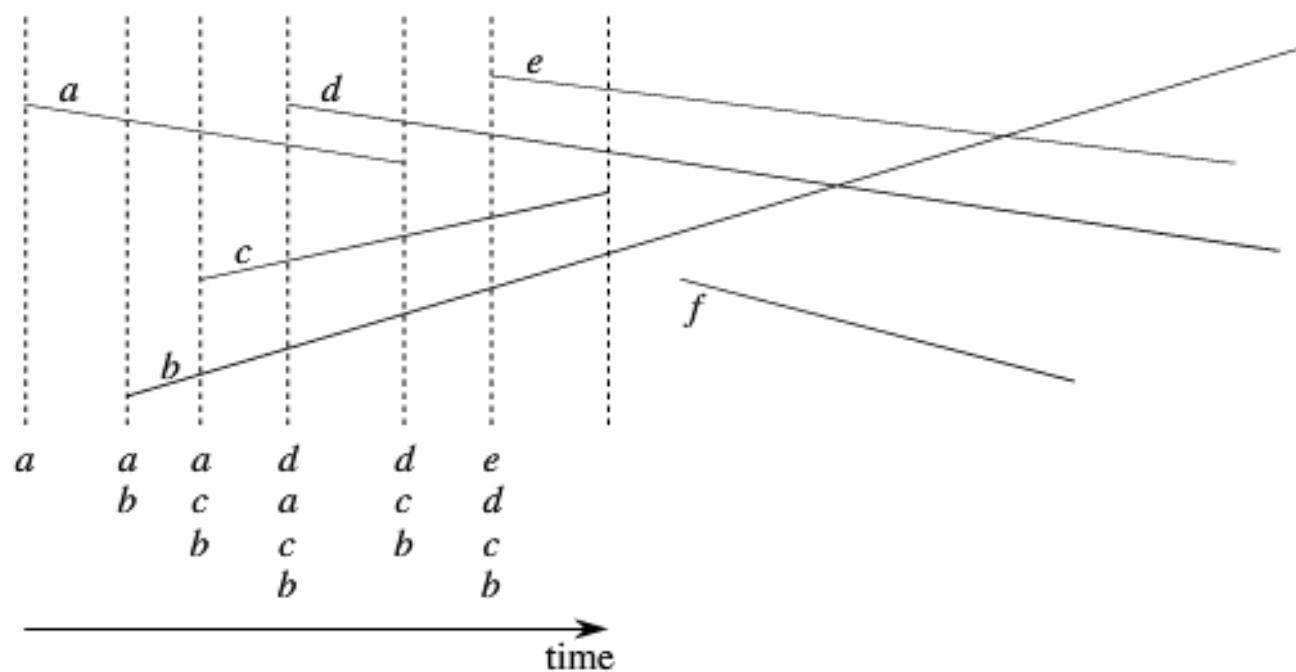
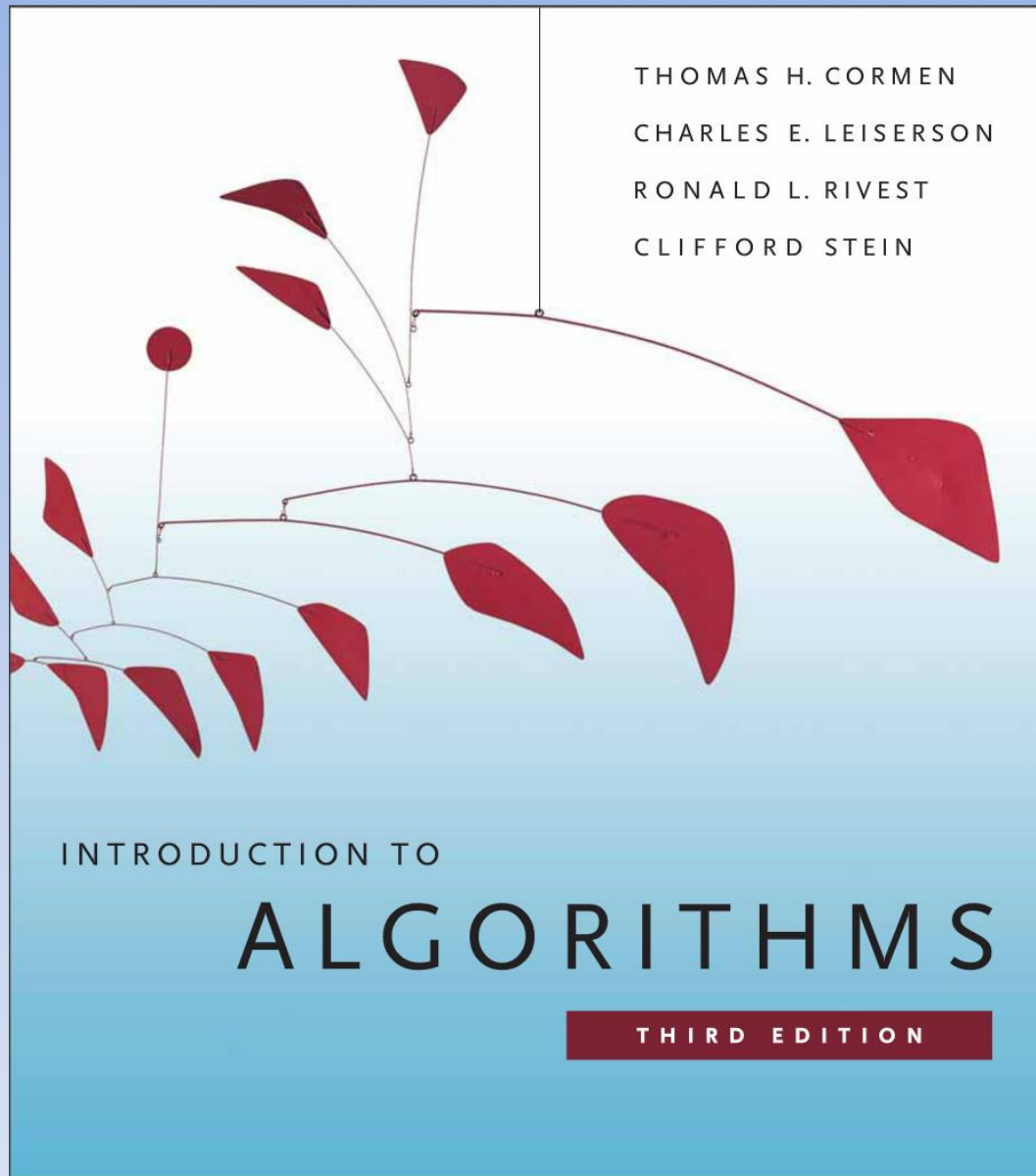


Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point. Except for the rightmost sweep line, the ordering of segment names below each sweep line corresponds to the total preorder T at the end of the **for** loop processing the corresponding event point. The rightmost sweep line occurs when processing the right endpoint of segment c ; because segments d and b surround c and intersect each other, the procedure returns TRUE.

	Introduction	3
1	The Role of Algorithms in Computing	5
1.1	Algorithms	5
1.2	Algorithms as a technology	11
2	Getting Started	16
2.1	Insertion sort	16
2.2	Analyzing algorithms	23
2.3	Designing algorithms	29
3	Growth of Functions	43
3.1	Asymptotic notation	43
3.2	Standard notations and common functions	53
4	Divide-and-Conquer	65
4.1	The maximum-subarray problem	68
4.2	Strassen's algorithm for matrix multiplication	75
4.3	The substitution method for solving recurrences	83
4.4	The recursion-tree method for solving recurrences	88
4.5	The master method for solving recurrences	93
★	4.6 Proof of the master theorem	97
5	Probabilistic Analysis and Randomized Algorithms	114
5.1	The hiring problem	114
5.2	Indicator random variables	118
5.3	Randomized algorithms	122
★	5.4 Probabilistic analysis and further uses of indicator random variables	130

	Introduction	229
10	Elementary Data Structures	232
	10.1 Stacks and queues	232
	10.2 Linked lists	236
	10.3 Implementing pointers and objects	241
	10.4 Representing rooted trees	246
11	Hash Tables	253
	11.1 Direct-address tables	254
	11.2 Hash tables	256
	11.3 Hash functions	262
	11.4 Open addressing	269
★	11.5 Perfect hashing	277
12	Binary Search Trees	286
	12.1 What is a binary search tree?	286
	12.2 Querying a binary search tree	289
	12.3 Insertion and deletion	294
★	12.4 Randomly built binary search trees	299
13	Red-Black Trees	308
	13.1 Properties of red-black trees	308
	13.2 Rotations	312
	13.3 Insertion	315
	13.4 Deletion	323
14	Augmenting Data Structures	339
	14.1 Dynamic order statistics	339
	14.2 How to augment a data structure	345
	14.3 Interval trees	348



174 : Chapter 11

Hashing!

Dictionary Operations



- Insert: Insert item
- Search: Find item
- Delete: Delete an item
- Hash Tables are great w/ Dictionaries
- Look @ Analysis of Hashing!
- First : Revisit Hashing!
 - If you've seen it before (in another context), Then the Text description should flow more easily
 - Opportunity to measure current understanding

Dictionary w/ Python

```
>>> OED = {}  
>>> OED["Algorithm"]="2. Math. and Computing. A procedure or set of rules used i  
n calculation and problem-solving; (in later use spec.) a precisely defined set  
of mathematical or logical operations for the performance of a particular task."
```

- Insert item: $O(1)$

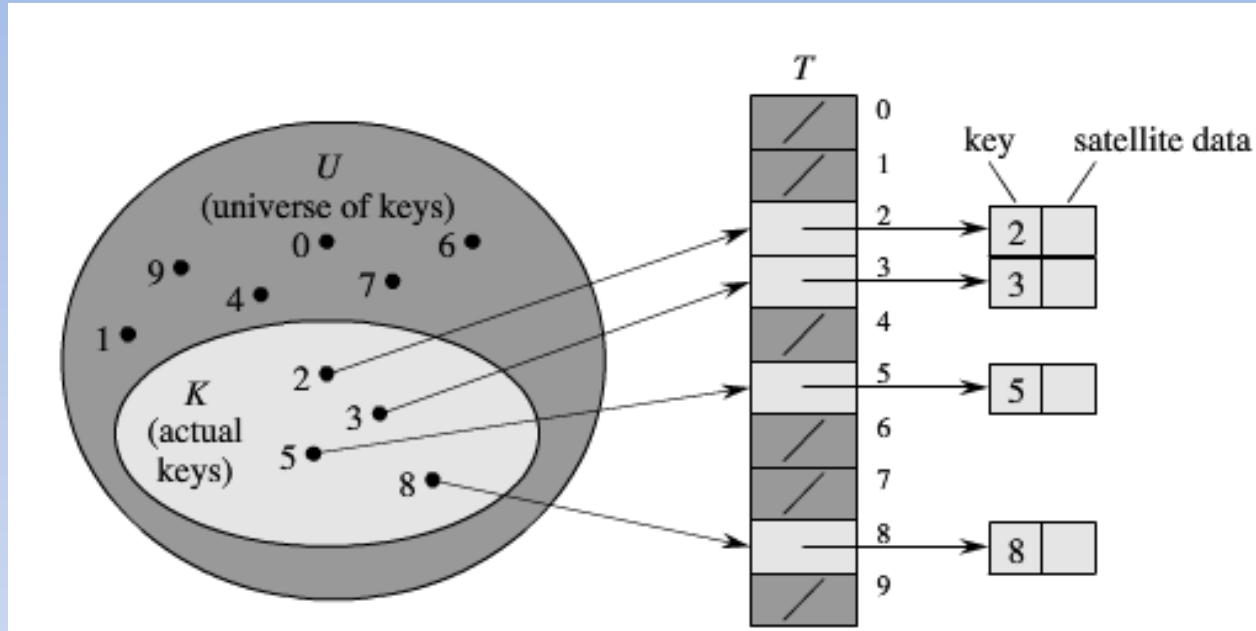
```
>>> print OED["Algorithm"]  
2. Math. and Computing. A procedure or set of rules used in calculation and prob  
lem-solving; (in later use spec.) a precisely defined set of mathematical or log  
ical operations for the performance of a particular task.
```

- Find item: $O(1)$

Direct Addressing versus Hashing

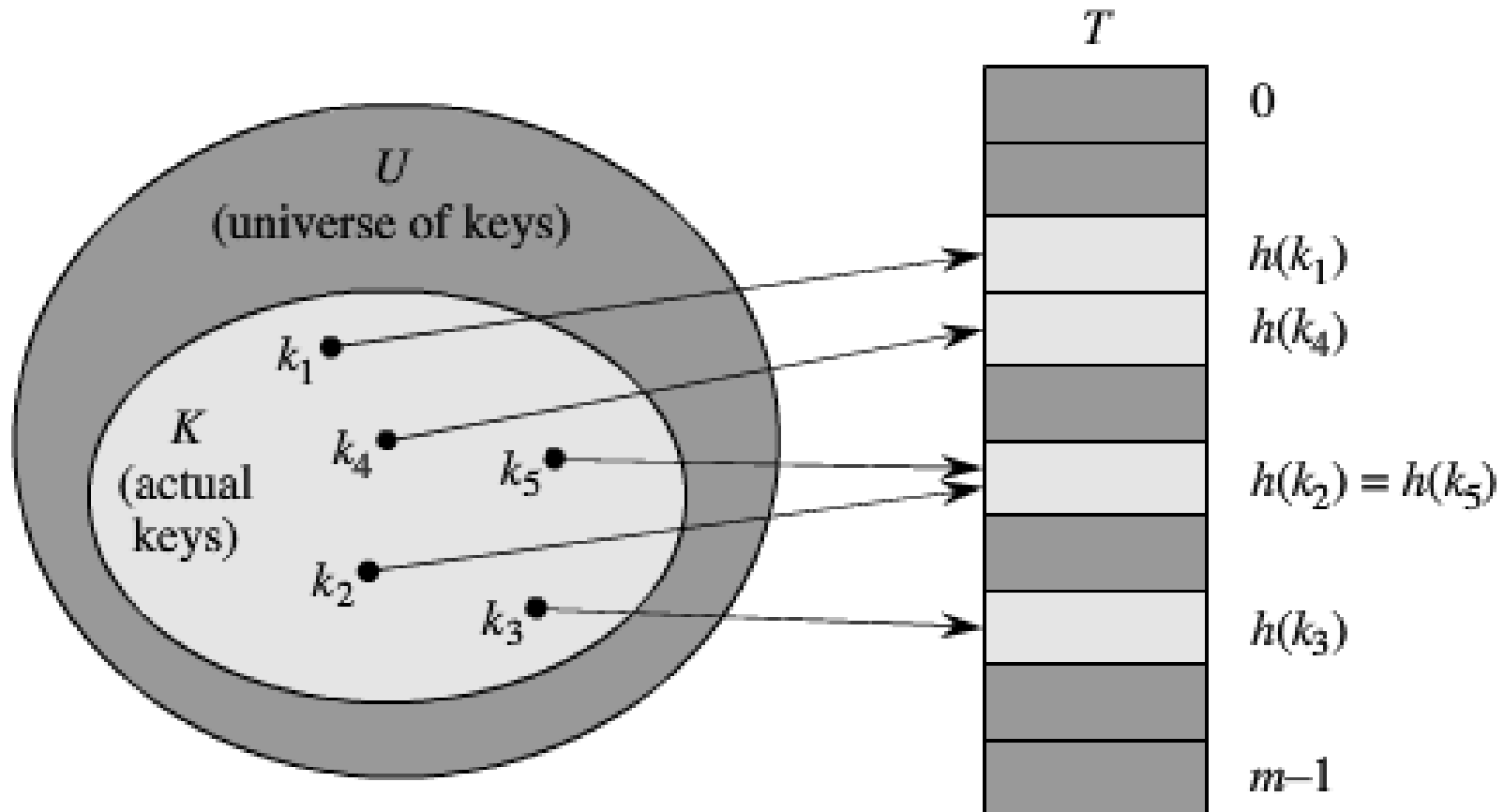
- Ordinary array uses direct addressing:
 - Offsetting by an integer index into a fixed length area of memory.
- Hash Table generalizes direct addressing:
 - Key range is large relative to the number of keys stored.
 - Hash tables use space proportional to the number of keys (not key range).
- Perfect Hashing can support searches in $O(1)$ worst-case time
 - where the set of keys being stored is static

Direct Addressing

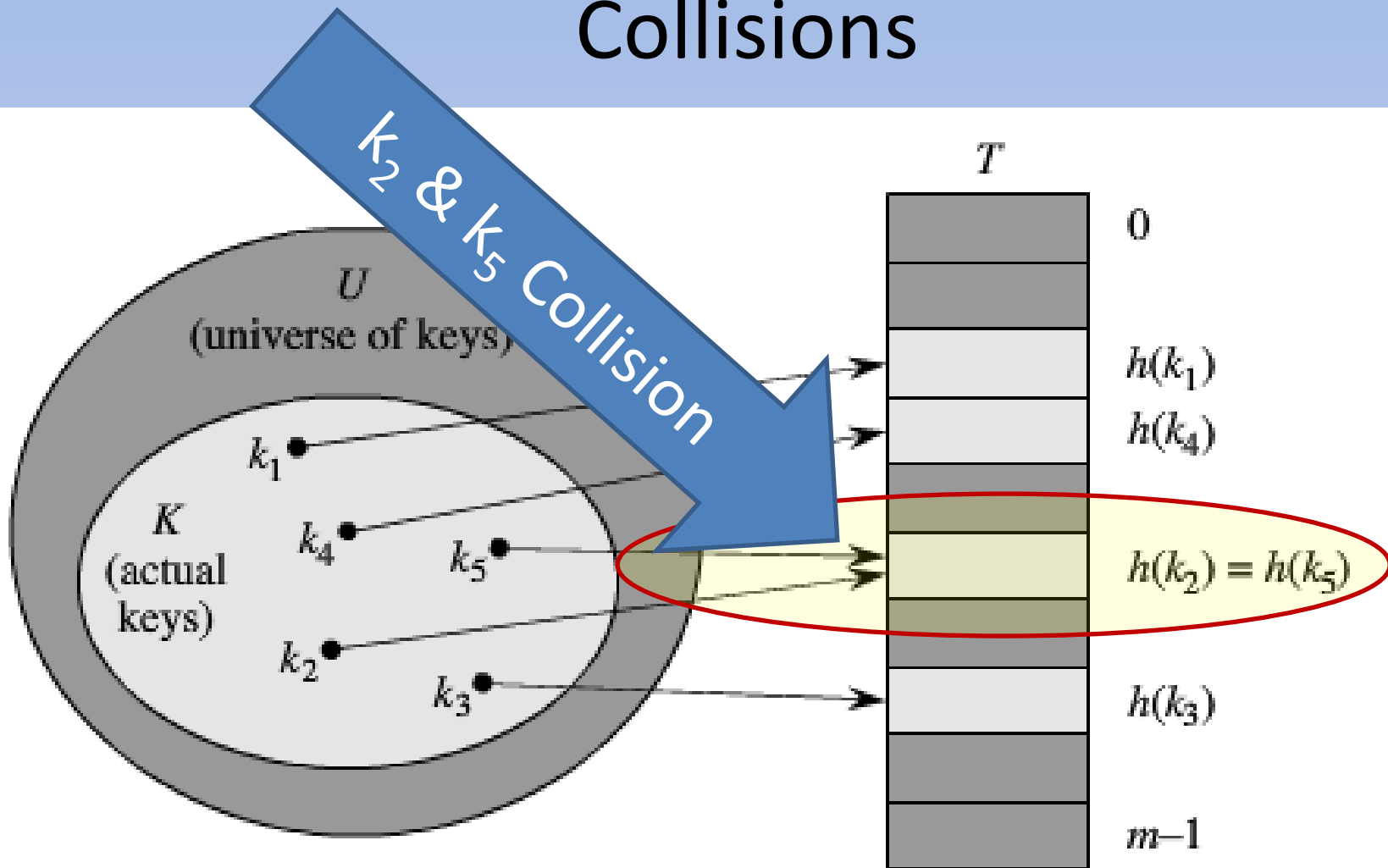


- Search, Insert, Delete: $O(1)$!

Hash Tables w/ Hash Function: $h(k)$



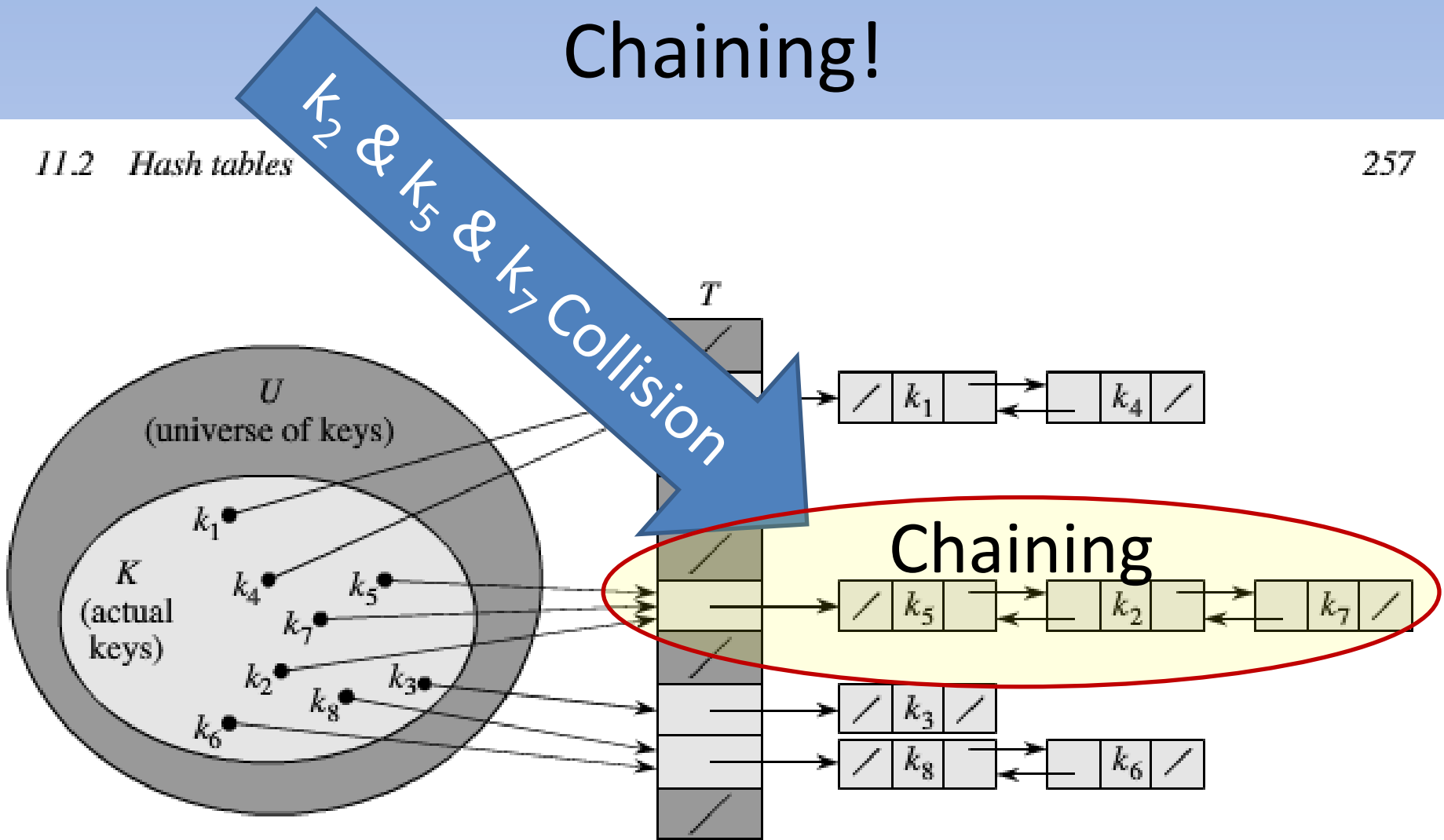
Hash Tables w/ Collisions



Hash Tables w/ Chaining!

11.2 Hash tables

257



How good is Hashing w/ Chaining ??

- How well does hashing perform w/ chaining ??
- How long to search for element w/ given key ??

Analyzing Hashing

- Hash Table T w/ m slots and n elements!
- Load factor α for T as n/m
 - average number of elements stored in a chain
 - Analysis in terms of α
 - Which can be less than, equal to, or greater than 1
- Worst-Case behavior Horrible w/ Hashing!
 - $\Theta(n)$ + time to produce Hash
- Average Performance depends on how well the hash spreads out the keys!

Analyzing Average w/ Hashing

- Average Performance depends on how well the hash spreads out the keys!
 - But for now ignore this!
- Assume any given element is equally likely to hash into any of the m slots!
 - **Simple Uniform Hashing!**

Chains w/ Expected Length

- Our performance will depend on the length of the chains hanging off the m slots!
- $n_j = \text{len}(T[j])$ for $j = 0, 1, \dots, m - 1$ denote the length of the chain hanging off slot j
- What is the expected value of n_j under our Simple Uniform Hashing assumption ??

Chains w/ Expected Length

- What is the expected value of n_j under our Simple Uniform Hashing assumption ??
- $E[n_j] = \alpha = n/m$
- Now our questions is given an element :
 - What's the TIME to FIND it?
- Two cases to consider :
 1. In the first case, we fail to find it..
 - It's not IN the Hash table!
 2. In the second case, we find it!
 - In this case, the key is IN the Hash table!

Theorem 11.1

- In a Hash Table, In which Collisions are Resolved by Chaining...
- An Unsuccessful Search takes average-case Time $\Theta(1 + \alpha)$,
 - UNDER the assumption of SIMPLE UNIFORM HASHING!

Theorem 11.2

- In a Hash Table, In which Collisions are Resolved by Chaining...
- Successful Searches take average-case Time $\Theta(1 + \alpha)$
 - UNDER the Uniform Hashing Assumption

Hash Functions

- You can probably guess some of what's on our Hashing wish list?
- KEY: Each key is equally likely to Hash to any of the m slots
 - Independently of where any other keys has hashed to!
- We usually do not know enough about the distribution of the keys!

Hash Functions w/ Known Distribution

- IF we know the distribution of keys:
 - LIKE: keys are random real numbers independent and uniformly distribution in the range:

$$0 \leq k < 1$$

- SO, in this case the condition of simple uniform hashing is satisfied by :

$$h(k) = \lfloor km \rfloor$$

First Assumption: Keys are Natural Numbers!

- If keys are really NOT naturals, we just try to find a way to look at them as natural!
- For Example: Character String
 - Assume characters in string are represented by ASCII code.
 - Treat string of characters a big int each each characters representing a digit in a radix-128 integer.

- “David” =

$$\begin{aligned} &68 * 128^4 + 97 * 128^3 + 118 * 128^2 + 105 * 128 + 100 \\ &\quad * 128^0 \\ &= 18458981604 \end{aligned}$$

Division Method

- Basic method for mapping keys from natural numbers into some m fixed number of slots
 - $h(k) = k \bmod m$
- Put some constraints on our choice of m to improve hash:
 - Do not use Power of 2!
 - Mod'ing to the power of 2 amounts to grabbing last bits
 - Many cases low-order bits not randomly distributed
 - Like function to operate on all bits
 - Permuting characters mod (radix-1) does not change value!

Permuting characters mod (radix-1) does not change value!

- $h(\text{"David"}) = 18458981604 \% (128-1) = 107$
- $h(\text{"avidD"}) = 26287436356 \% (128-1) = 107$
- $h(\text{"vidDa"}) = 31897231969 \% (128-1) = 107$

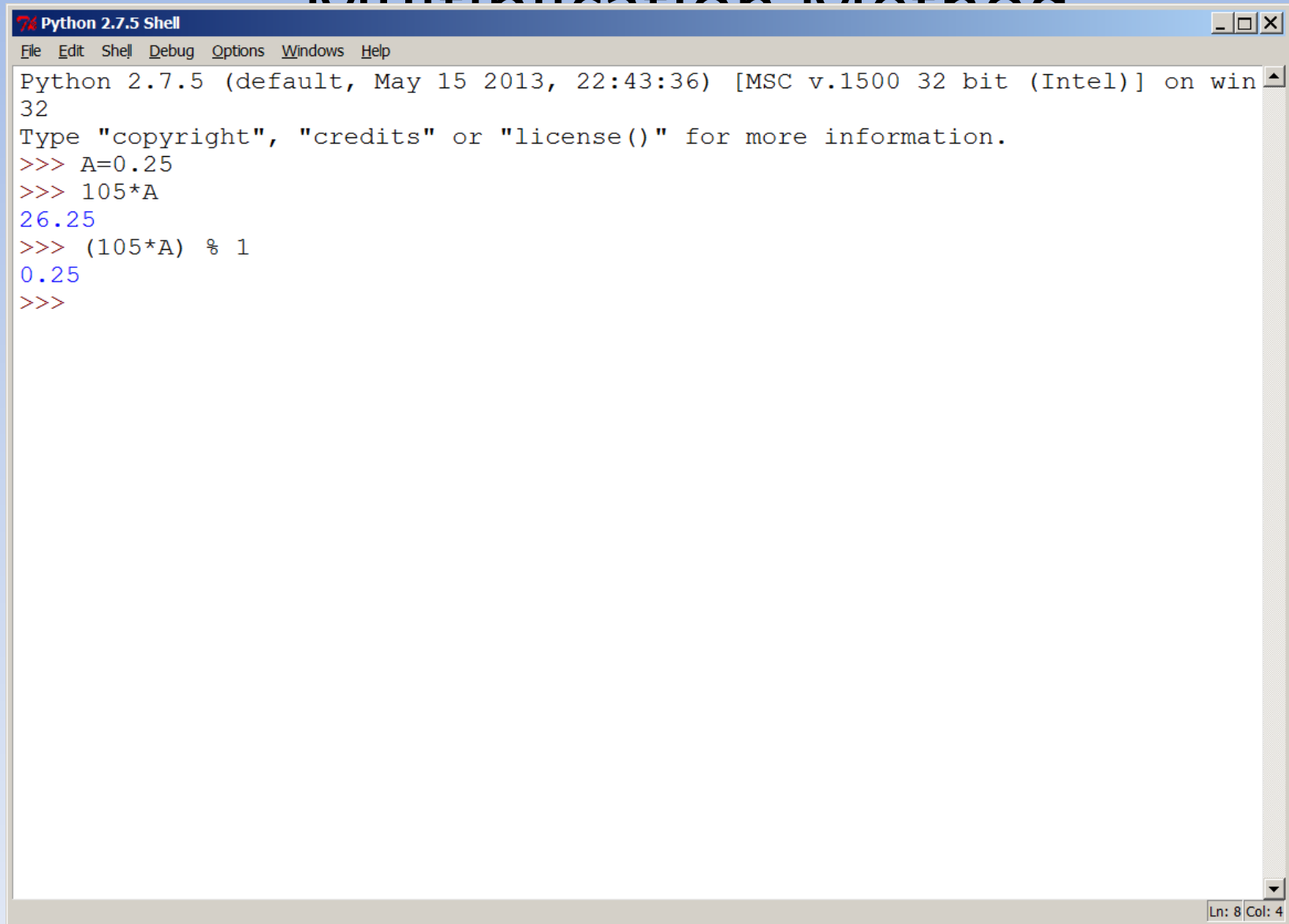
```
h=lambda x : sum([ord(x[i])*128**(len(x)-i-1) for i in range(len(x))])%127
```

- You can prove this!
 - Exercise: 11.3-3
- A prime not too close to an exact power of 2 is often a good choice for m.

Hashing w/ Multiplication Method

- Here we introduce A , a number between 0 and 1.
- We'll multiply our key k by A giving us a real number.
- We'll mod that with one, getting the fractional part of kA .
 - $A=0.25$
 - $(105 * A) \bmod 1 = 0.25$
 - $(106 * A) \bmod 1 = 0.5$
 - $(107 * A) \bmod 1 = 0.75$
- mA gives a value between 0 and $m-1$

Hashing w/ Multiplication Method



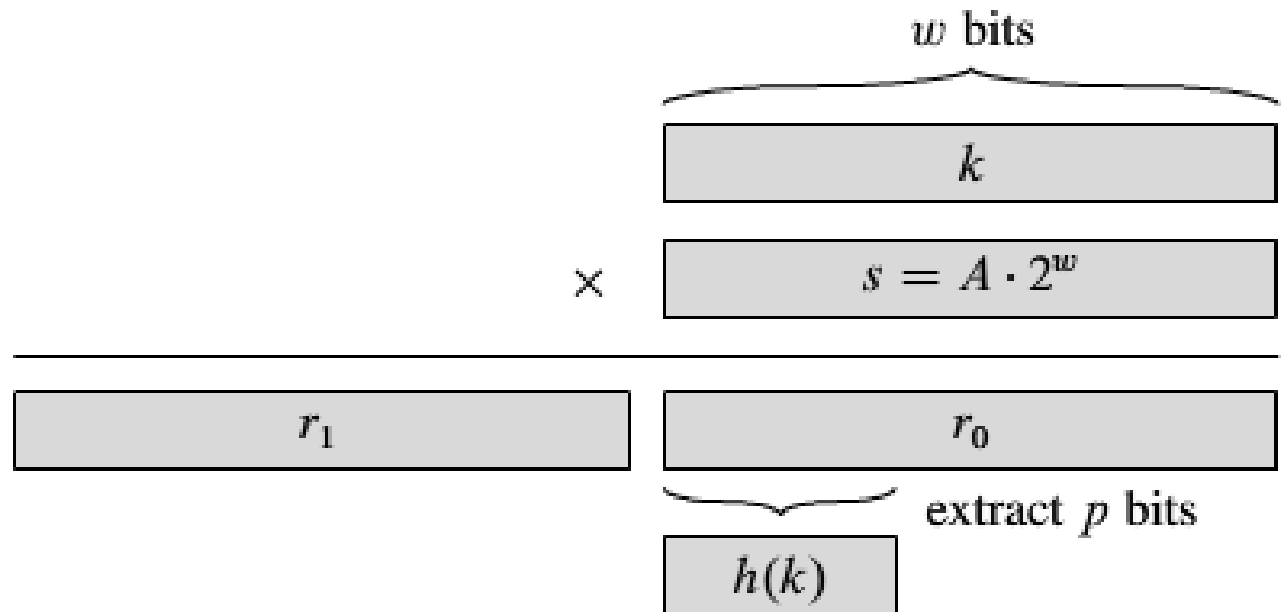
```
Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> A=0.25
>>> 105*A
26.25
>>> (105*A) % 1
0.25
>>>
```

1.

Hashing w/ Multiplication Method

264

Chapter 11 Hash Tables



- Now m doesn't matter:
 - Use $m=2^p$ for some p .
- Value of A does matter:
 - Knuth recommends $(\sqrt{5} - 1)/2 = 0.6280339887\dots$

Hashing w/ Multiplication Method

- Use $m=2^p$ for some p
 - $p=7$ so $2^p=128$
- Use $A=(\sqrt{5} - 1)/2 = 0.6280339887...$
- $h(\text{"David"}) = 128 * (18458981604 * A \% 1) = 125$
- $h(\text{"avidD"}) = 128 * (26287436356 * A \% 1) = 13$
- $h(\text{"vidDa"}) = 128 * (31897231969 * A \% 1) = 112$

Improving Worst-Case w/ Universal Hashing

- In the worst-case, bad luck (or malicious adversary) conspire to provide the worst possible sequence of keys to hash.
 - All keys hash to same slot!
- Whenever the hash function is unchanging, then this is always a possibility!
- Only effective defense is to select the hash function randomly and independent from keys!
- Universal Hashing is one approach!

Universal Hashing w/ Universal Collection of Hash Functions

- Assume we have :
 - two distinct keys k, l
 - A collection of hash function mapping to a range $0..m-1$
- Our collection of hash functions is universal IF:
 - The size of the collection of all hash functions that map both k, l to the same slot is $(1/m)^{\text{th}}$ of the total number of functions in the collection.

Corollary 11.4 w/ Theorem 11.3

- Corollary 11.4 uses 11.3 to show that :
 - Search operation with n total Inserts & Searches & Deletes
 - but $O(m)$ Inserts out of total
- is $O(1)$ for each search,
- so w/ Linearity of Expectation is $O(n)$ for the collection of operations.

Universal Class of Hash Functions w/ a little number theory

- We'll return to this w/ RSA Encryption
- Choose prime number p so all keys are in range $0..p-1$ inclusive!
- Now Define:
 - $\mathbb{Z}_p = \{0, 1, \dots, p-1\}$ More on this set later!
 - $\mathbb{Z}_p^* = \{1, \dots, p-1\}$
- Now we can define a whole collection of hash functions as:
 - $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$
 - $\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ which contains $p(p-1)$ functions

Universal Hashing

- Now a given our large prime number p
- We have a large class of hash functions to choose from
 - $p^{*}p-1$

Open Addressing

- Currently we have dealt with keys colliding at same slots w/ chains!
 - Chains hang off hash table!
- Open Addressing is an alternative w/ all elements kept IN hash table!
- Slots in Hash Table are examined systematically until item is found or deemed absent!
- All slots can get “filled up” w/ Open Addressing

Open Addressing w/ NO POINTERS!!

- Could keep chain in Hash Table, but we don't
- Freeing pointers provides more storage space.
 - Just use empty space in hash table!
- Insert key by probing slots until finding an empty one to put key in!
- Slots probed are ordered based on the inserted key!

Probing Slots

- Probe sequence is determined by expanding Hash function
- Hash function includes additional parameter Probe Number (starting w/ 0)
- Our probe sequence must be a permutation of the set of slots (so all slots get probed!):

$\langle h(k,0), h(k,1), \dots, h(k, m-1) \rangle$

= Permutation ($\langle 0, 1, \dots, m-1 \rangle$)

Inserting & Finding

HASH-INSERT(T, k)

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == \text{NIL}$ 
5           $T[j] = k$ 
6          return  $j$ 
7      else  $i = i + 1$ 
8  until  $i == m$ 
9  error “hash table overflow”
```

HASH-SEARCH(T, k)

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == k$ 
5          return  $j$ 
6       $i = i + 1$ 
7  until  $T[j] == \text{NIL}$  or  $i == m$ 
8  return  $\text{NIL}$ 
```

- Finding follows same probe sequence as Inserting.
- Removal can be tricky!

Open Addressing w/ Key Deletion

- Cannot just remove the item, marking it nil
 - May not know to continue probing sequence while searching
- Instead, mark item DELETED.
 - Insertion treats slots deleted as empty.
 - Search walks over deleted slots
- BUT, w/ deleted items:
 - searching no longer depends on load factor n/m !
 - Since slots with deleted items still potentially searched.
 - Frequently avoided when deletion needed!

Probing Techniques w/ Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing!
- BUT: None satisfy assumptions of Uniform Hashing
 - Limited in the number of probe sequences generated
 - m^2 versus $m!$
 - Double Hashing generating the most probe sequences

Linear Probing

- Auxiliary Hash Function:

$$h': U \rightarrow \{0, 1, \dots, m - 1\}$$

- Hash Function

$$h(k, i): (h'(k) + i) \bmod m$$

- How many distinct probes??

Linear Probing

- Auxiliary Hash Function:

$$h': U \rightarrow \{0, 1, \dots, m - 1\}$$

- Hash Function

$$h(k, i): (h'(k) + i) \bmod m$$

- How many distinct probes?? m
- Suffers from Primary Clustering
 - Long runs of occupied slots
 - Increased search time

Quadratic Probing

$$h(k, i): (h'(k) + c_1i + c_2i^2) \bmod m$$

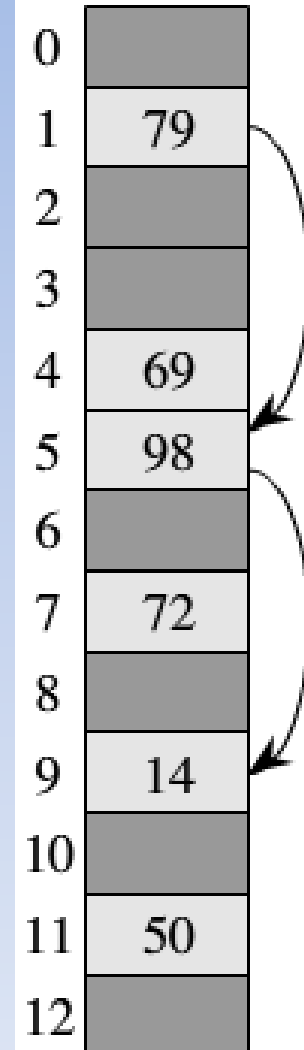
- c_1 , c_2 , and m are constrained
- Still suffers from clustering (secondary clustering).
- Still just m distinct probes.

Double Hashing

- One of the best for Open Addressing

$$h(k, i): (h_1(k) + ih_2(k)) \bmod m$$

- $h_1(k) = k \bmod 13$
- $h_2(k) = 1 + (k \bmod 11)$
- $k=14$
 - $h(14,0) = 1$
 - $h(14,1) = 1 + (1+3) = 5$
 - $h(14,2) = 1 + 2*(1+3) = 9$



Double Hashing

- One approach is :

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod m')$$

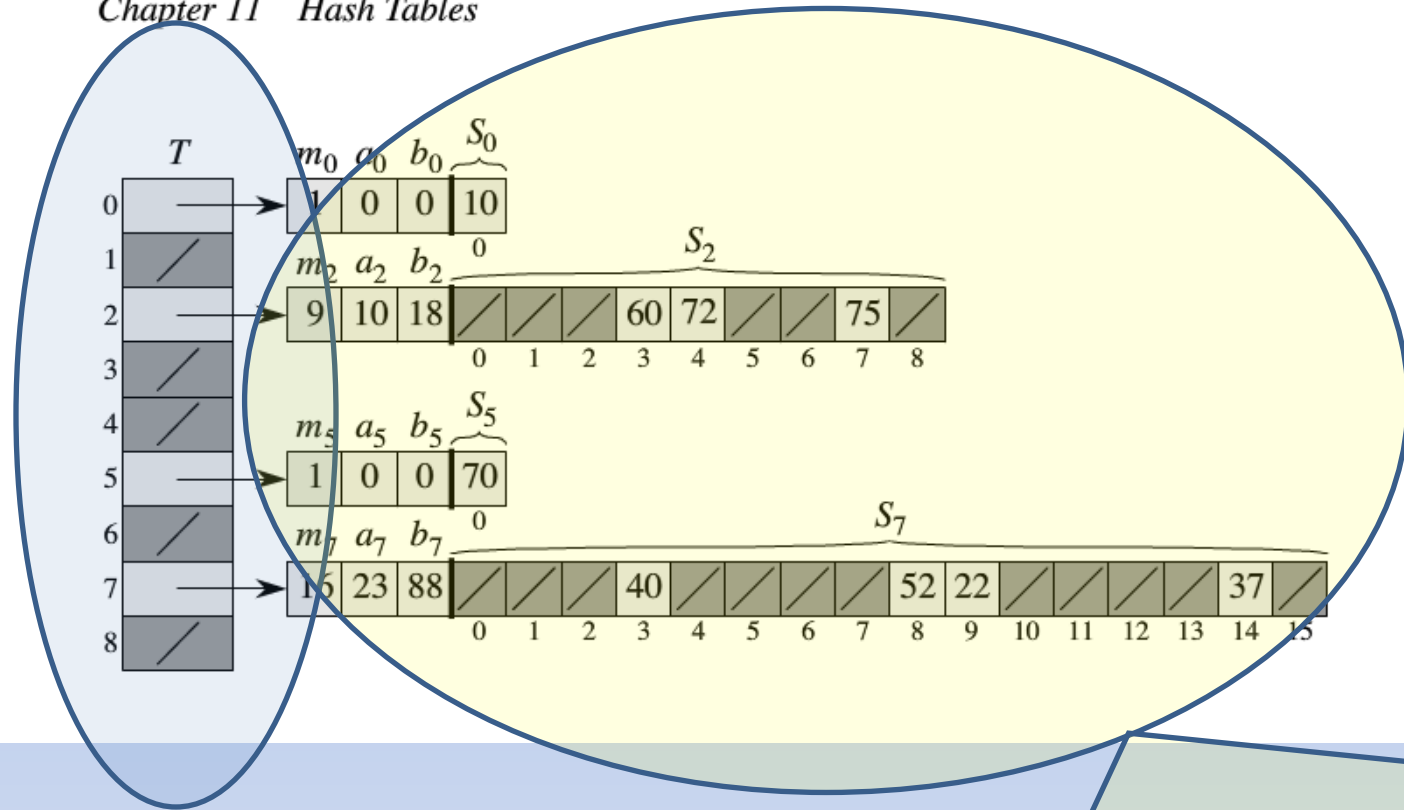
- Where
 - m = Prime Number
 - m' = is slightly less, like $(m-1)$
- Double hashing provides for m^2 probe sequences
 - $h_1(k)$ and $h_2(k)$ yield distinct probe sequences!

Perfect Hashing

- So far we've seen Hashing has GREAT average-case performance.
- But did you know...???
 - Hashing can also provide excellent worst-case performance...
 - IF: Keys are static!
 - Once stored the set of keys never changes!

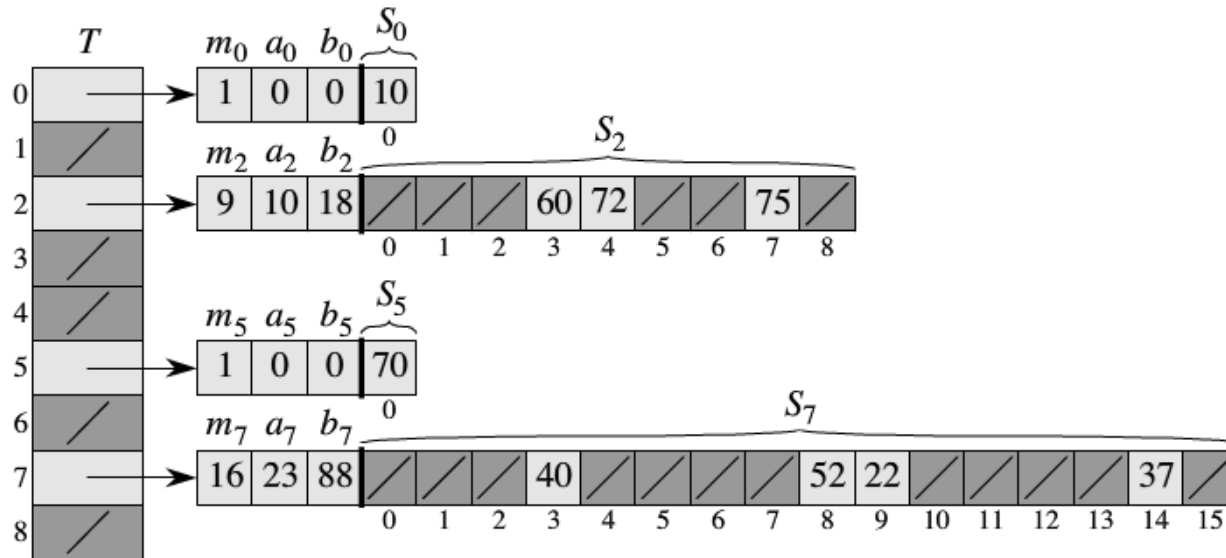
Perfect Hashing

- Perfect Hashing uses Universal Hash Function initially
 - like hashing with chaining
- BUT : Instead of Chaining, a second hash function is used!
- TWO Levels of hashing combine to provide $O(1)$ worst-case behavior!



First Level

Second Level



- The size of the second level hash table is :

$$S_j \text{ is } m_j = n_j^2$$

- No collisions occur with secondary hashing!
- The hash function for secondary hashing for each hash table is h_j and is chosen from a set of universal hash functions!

Theorem 11.9

- If we store n keys in a hash table of size $m = n^2$ then the probability of ANY collisions is $\frac{1}{2}$!
- This is a great result!
- BUT, in some cases n^2 is too large for our table size.

Theorem 11.10

- If we store n keys in a hash table size $m=n$,
- THEN we are going to need second level hash tables of size n_j^2
 - Here n_j is the number of keys that hash to slot j .
- THEN we want the Expected sum of all of these second level hash tables to be less than $2n!!$

Binary Search Trees

12.1 What is a binary search tree?

287

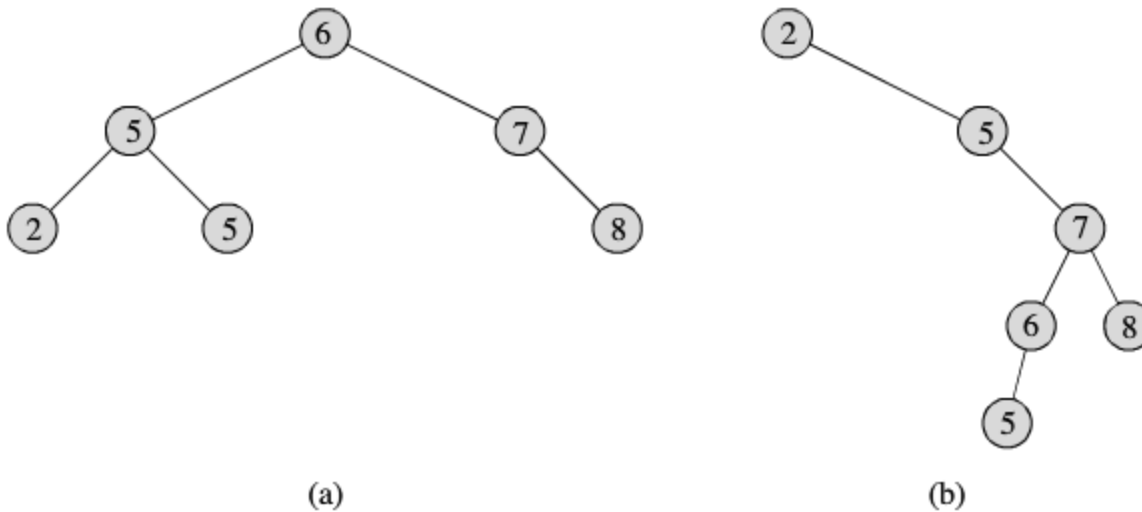


Figure 12.1 Binary search trees. For any node x , the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efficient binary search tree with height 4 that contains the same keys.

Binary Search Trees

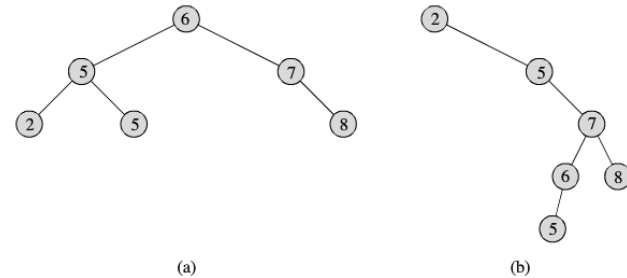


Figure 12.1 Binary search trees. For any node x , the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efficient binary search tree with height 4 that contains the same keys.

- Linked Data Structure
- Each node has pointers (along with key value and satellite data):
 - p: Parent
 - left: Left Subtree
 - right: Right Subtree

Binary-SearchTree Property

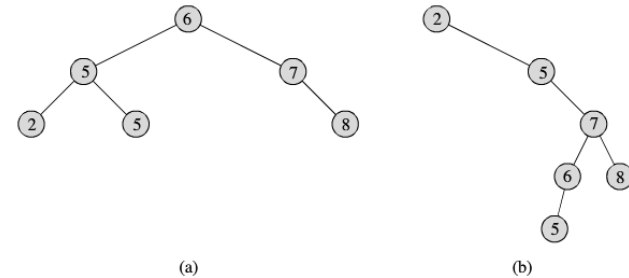


Figure 12.1 Binary search trees. For any node x , the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efficient binary search tree with height 4 that contains the same keys.

- Let x be a node in a binary search tree.
- IF y is a node in the left subtree of x ,
 - THEN $y.key \leq x.key$.
- IF y is a node in the right subtree of x ,
 - THEN $y.key \geq x.key$.

Binary-SearchTree Property

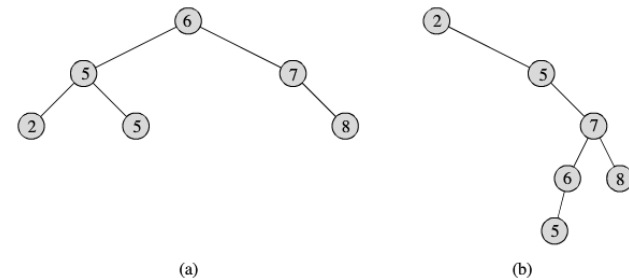


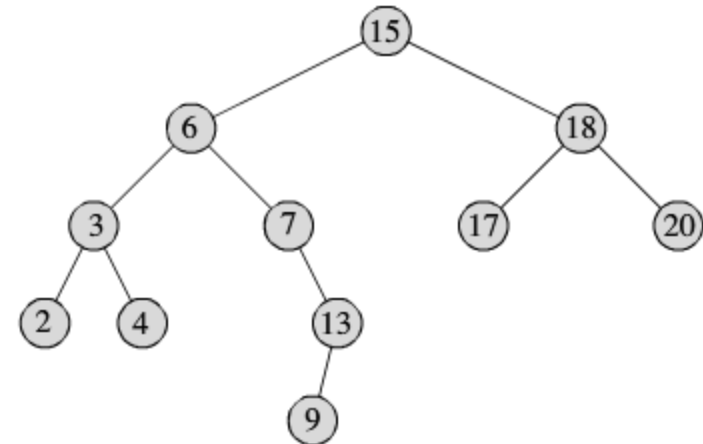
Figure 12.1 Binary search trees. For any node x , the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efficient binary search tree with height 4 that contains the same keys.

- Inorder Walk allows printing out key value in sorted order:

INORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )
```

Searching



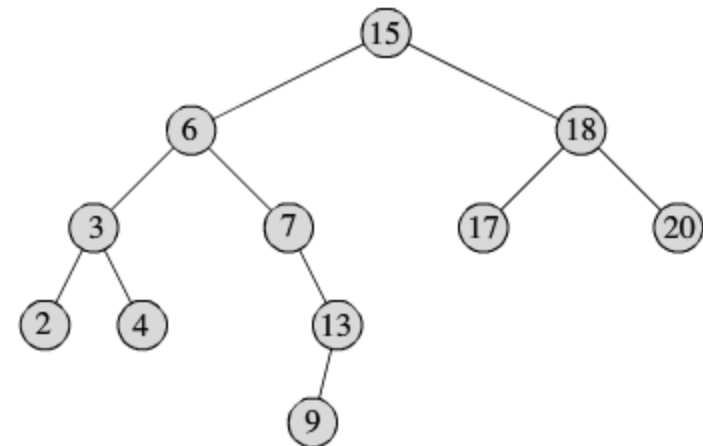
TREE-SEARCH(x, k)

- 1 **if** $x == \text{NIL}$ or $k == x.key$
- 2 **return** x
- 3 **if** $k < x.key$
- 4 **return** TREE-SEARCH($x.left, k$)
- 5 **else return** TREE-SEARCH($x.right, k$)

Iterative Tree Searching

290

Chapter 12 Binary Search Trees

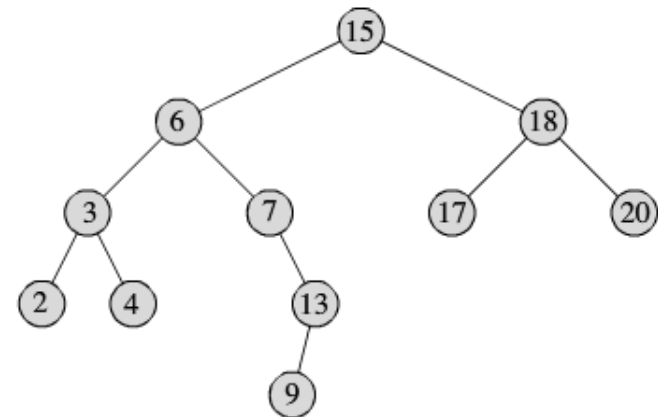


12.2 Querying a binary search tree

291

ITERATIVE-TREE-SEARCH(x, k)

```
1  while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2      if  $k < x.\text{key}$ 
3           $x = x.\text{left}$ 
4      else  $x = x.\text{right}$ 
5  return  $x$ 
```

TREE-MINIMUM(x)

```

1  while  $x.left \neq \text{NIL}$ 
2       $x = x.left$ 
3  return  $x$ 
  
```

TREE-MAXIMUM(x)

```

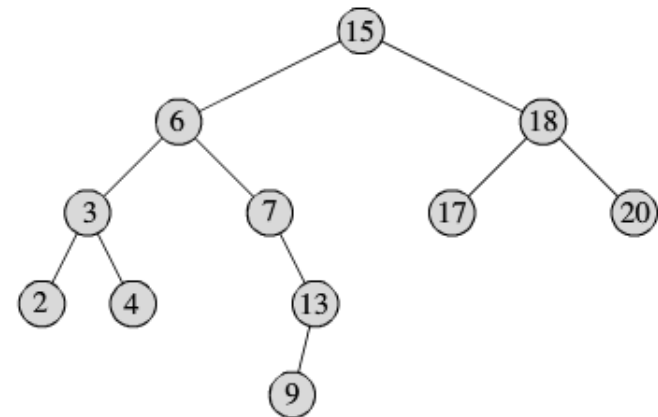
1  while  $x.right \neq \text{NIL}$ 
2       $x = x.right$ 
3  return  $x$ 
  
```

TREE-SUCCESSOR(x)

```

1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.p$ 
4  while  $y \neq \text{NIL}$  and  $x == y.right$ 
5       $x = y$ 
6       $y = y.p$ 
7  return  $y$ 
  
```

IF Node has a Right
Succ is Min of Right



TREE-MINIMUM(x)

```

1  while  $x.left \neq \text{NIL}$ 
2       $x = x.left$ 
3  return  $x$ 
  
```

TREE-MAXIMUM(x)

```

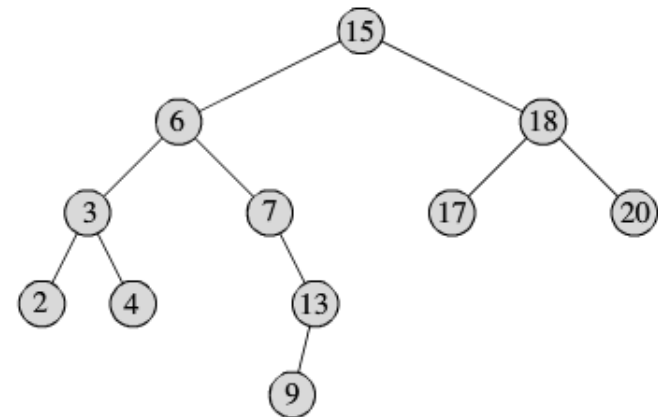
1  while  $x.right \neq \text{NIL}$ 
2       $x = x.right$ 
3  return  $x$ 
  
```

TREE-SUCCESSOR(x)

```

1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MAXIMUM( $x.right$ )
3   $y = x.p$ 
4  while  $y \neq \text{NIL}$  and  $x == y.right$ 
5       $x = y$ 
6       $y = y.p$ 
7  return  $y$ 
  
```

ELSE: Look To Parent



TREE-MINIMUM(x)

```

1  while  $x.left \neq \text{NIL}$ 
2       $x = x.left$ 
3  return  $x$ 
  
```

TREE-MAXIMUM(x)

```

1  while  $x.right \neq \text{NIL}$ 
2       $x = x.right$ 
3  return  $x$ 
  
```

TREE-SUCCESSOR(x)

```

1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.p$ 
4  while  $y \neq \text{NIL}$  and  $x == y.right$ 
5       $x = y$ 
6       $y = y.p$ 
7  return  $y$ 
  
```

IF: x is right child
Look To Parent

Return
First Left Child's Parent
or Nil

Insertion

- Example:
 - Inserting node z with key=13

TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$     // tree  $T$  was empty
11 elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```

12.3 Insertion and deletion

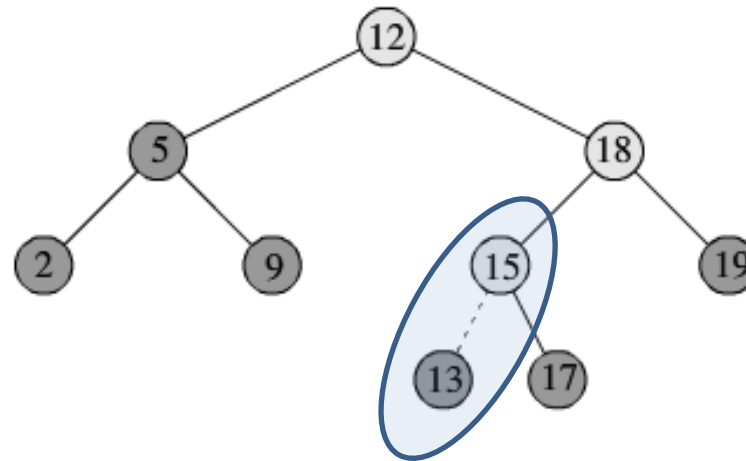


Figure 12.3 Inserting an item with key 13 into a binary search tree.

- y is trailing pointer
 - parent of x

Insertion

- Example:
 - Inserting node z with key=13

TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$     // tree  $T$  was empty
11 elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```

12.3 Insertion and deletion

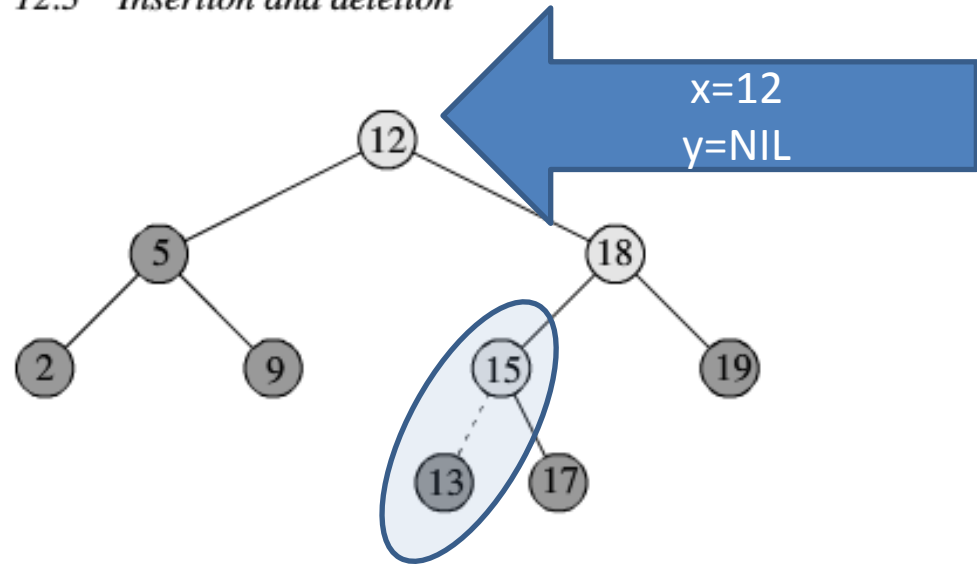


Figure 12.3 Inserting an item with key 13 into a binary search tree.

- $13 > 12$

Insertion

- Example:
 - Inserting node z with key=13

TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$     // tree  $T$  was empty
11 elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```

12.3 Insertion and deletion

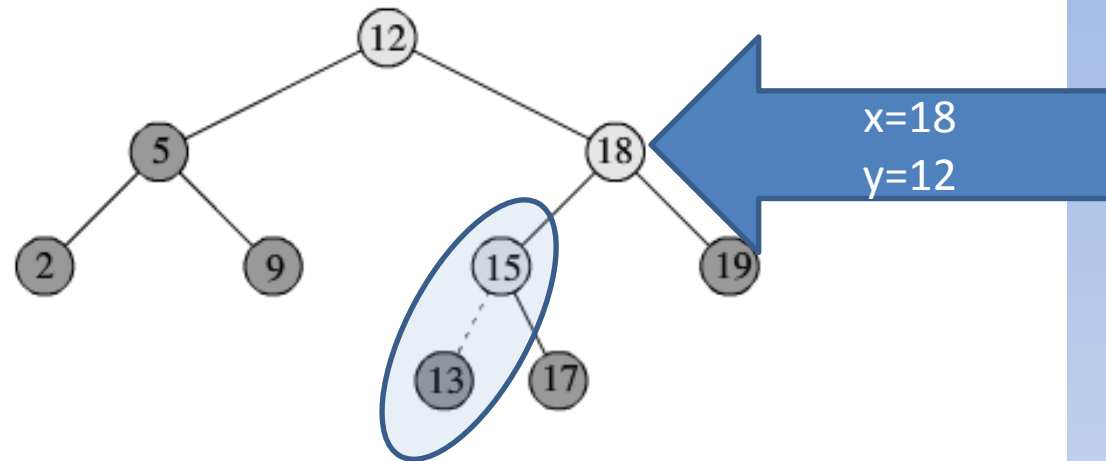


Figure 12.3 Inserting an item with key 13 into a binary search tree.

- $13 < 18$

Insertion

- Example:
 - Inserting node z with key=13

TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$     // tree  $T$  was empty
11 elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```

12.3 Insertion and deletion

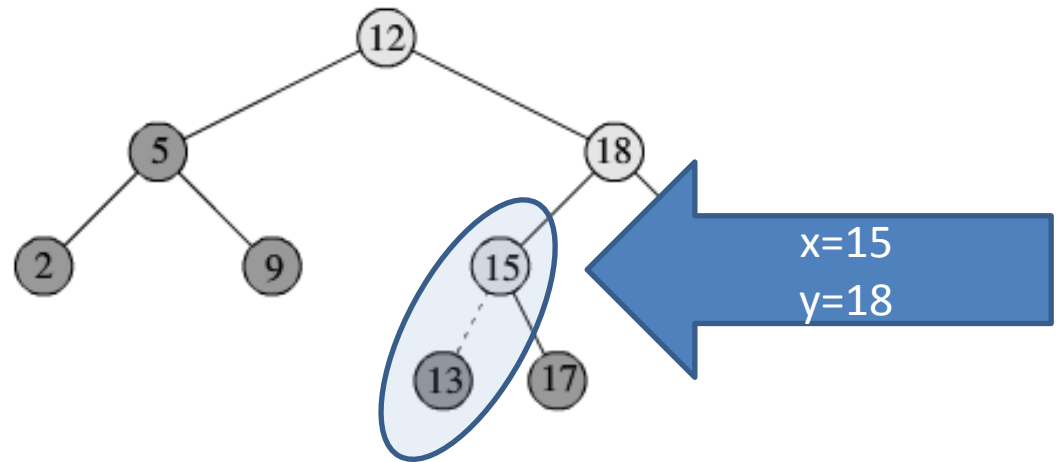


Figure 12.3 Inserting an item with key 13 into a binary search tree.

- $13 < 15$

Insertion

- Example:
 - Inserting node z with key=13

TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$       // tree  $T$  was empty
11 elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```

12.3 Insertion and deletion

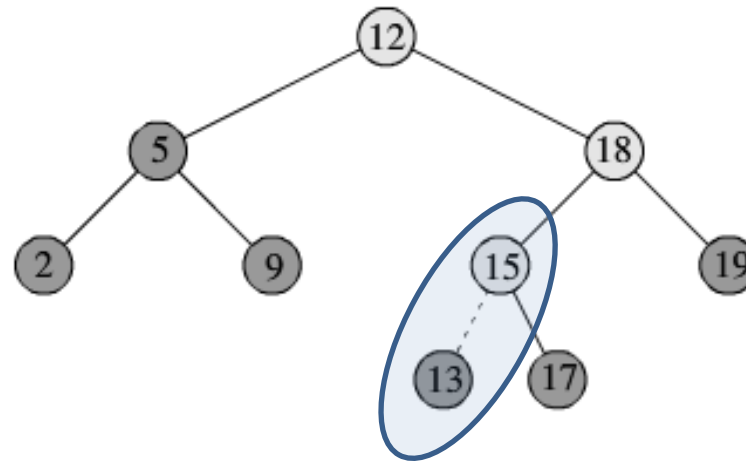


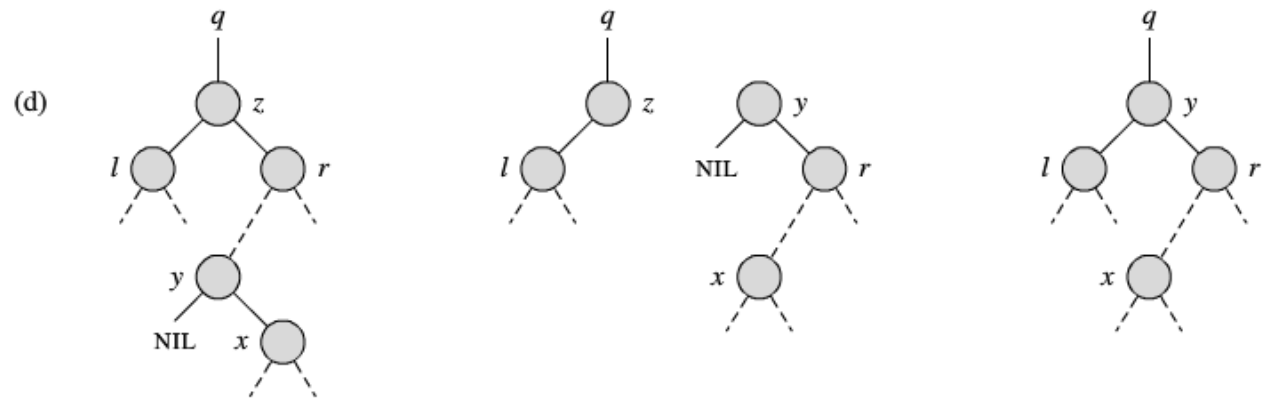
Figure 12.3 Inserting an item with key 13 into a binary search tree.

- $13 < 15$

z 's parent is y

z becomes y 's
left or right child

Deletion



TRANSPLANT(T, u, v)

```

1  if  $u.p == \text{NIL}$ 
2       $T.\text{root} = v$ 
3  elseif  $u == u.p.\text{left}$ 
4       $u.p.\text{left} = v$ 
5  else  $u.p.\text{right} = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
    
```

TREE-DELETE(T, z)

```

1  if  $z.\text{left} == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.\text{right}$ )
3  elseif  $z.\text{right} == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.\text{left}$ )
5  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
6      if  $y.p \neq z$ 
7          TRANSPLANT( $T, y, y.\text{right}$ )
8           $y.\text{right} = z.\text{right}$ 
9           $y.\text{right}.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.\text{left} = z.\text{left}$ 
12      $y.\text{left}.p = y$ 
    
```

	Introduction	229
10	Elementary Data Structures	232
	10.1 Stacks and queues	232
	10.2 Linked lists	236
	10.3 Implementing pointers and objects	241
	10.4 Representing rooted trees	246
11	Hash Tables	253
	11.1 Direct-address tables	254
	11.2 Hash tables	256
	11.3 Hash functions	262
	11.4 Open addressing	269
★	11.5 Perfect hashing	277
12	Binary Search Trees	286
	12.1 What is a binary search tree?	286
	12.2 Querying a binary search tree	289
	12.3 Insertion and deletion	294
★	12.4 Randomly built binary search trees	299
13	Red-Black Trees	308
	13.1 Properties of red-black trees	308
	13.2 Rotations	312
	13.3 Insertion	315
	13.4 Deletion	323
14	Augmenting Data Structures	339
	14.1 Dynamic order statistics	339
	14.2 How to augment a data structure	345
	14.3 Interval trees	348

Binary Search Trees

- Great average performance
 - Searching in $O(\lg n)$
- But worst-case performance is much worse!
 - Searching in $O(n)$!!
- Need to prevent tree from structures generating worst case performance!

Red-Black Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*T.nil*) is black.
4. If a node is red, then both its children are black.
 - No two reds in a row on a simple path from the root to a leaf.
5. All simple paths from a node to descendant leaves contain the same number of black nodes.

Black-Height

- **black-height** : the number of black nodes on any simple path from, but not including, a node x down to a leaf the of the node,
 - denoted $bh(x)$.
- By property 5,
 - the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes.
 - We define the black-height of a red-black tree to be the black-height of its root.

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.

Operations on Red-Black Trees

- Insertions and Deletion are not so easy
- If we insert, what color to make the new node?
 - Red? Might violate property 4
 - If a node is red, then both its children are black.
 - No two reds in a row on a simple path from the root to a leaf.
 - Black? Might violate property 5
 - All simple paths from a node to descendant leaves contain the same number of black nodes.

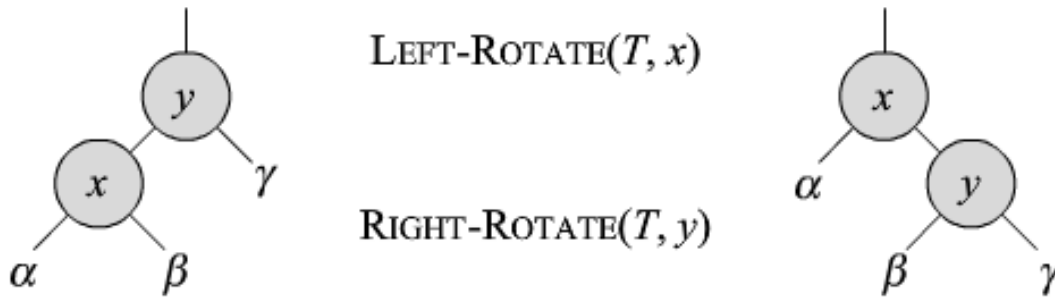
Rotations

- The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time.
- Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1.
- To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.
- We change the pointer structure through rotation, which is a local operation in a search tree that preserves the binary-search-tree property.

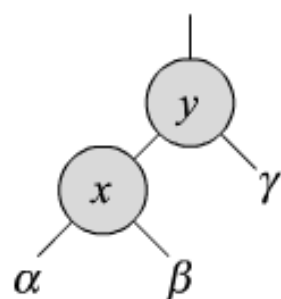
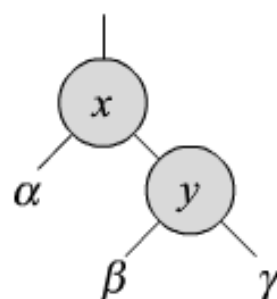
Rotations

13.2 Rotations

313



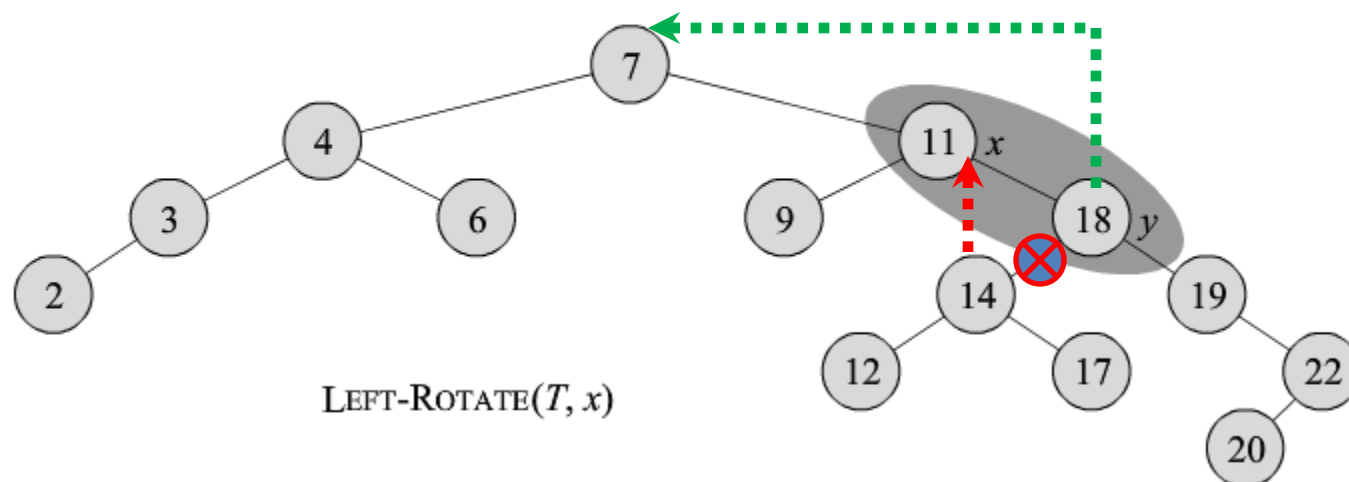
- Left-Rotate:
 - Shifts weight to the left.
- Right-Rotate:
 - Shifts weight to the right!

LEFT-ROTATE(T, x)RIGHT-ROTATE(T, y)LEFT-ROTATE(T, x)

```

1  y = x.right           // set y
2  x.right = y.left      // turn y's left subtree into x's right subtree
3  if y.left != T.nil
4      y.left.p = x
5  y.p = x.p             // link x's parent to y
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right = y
11 y.left = x            // put x on y's left
12 x.p = y

```



LEFT-ROTATE(T, x)

```

1   $y = x.right$                 // set  $y$ 
2   $x.right = y.left$             // turn  $y$ 's left subtree into  $x$ 's right subtree
3  if  $y.left \neq T.nil$ 
4       $y.left.p = x$  .....
5   $y.p = x.p$  ..... // link  $x$ 's parent to  $y$ 
6  if  $x.p == T.nil$ 
7       $T.root = y$ 
8  elseif  $x == x.p.left$ 
9       $x.p.left = y$ 
10 else  $x.p.right = y$ 
11  $y.left = x$                 // put  $x$  on  $y$ 's left
12  $x.p = y$ 

```

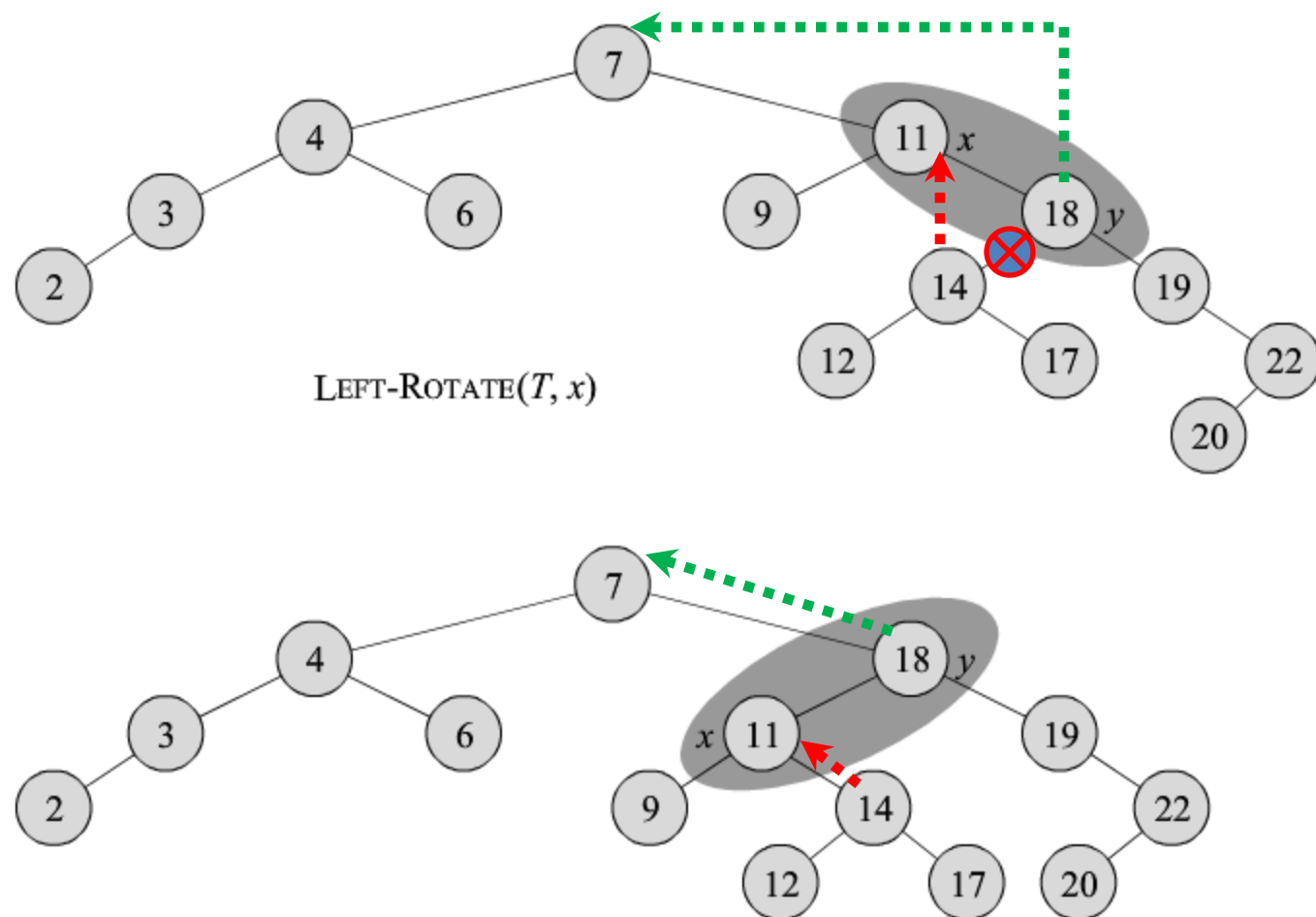


Figure 13.3 An example of how the procedure $\text{LEFT-ROTATE}(T, x)$ modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Rotations

- ***Time***
 - $O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.
- ***Notes***
 - Rotation is a very basic operation, also used in AVL trees and splay trees.
 - Some books talk of rotating on an edge rather than on a node.

Insertions

Start w/ BST Insertion

RB-INSERT(T, z)

$y = T.nil$

$x = T.root$

while $x \neq T.nil$

$y = x$

if $z.key < x.key$

$x = x.left$

else $x = x.right$

$z.p = y$

if $y == T.nil$

$T.root = z$

elseif $z.key < y.key$

$y.left = z$

else $y.right = z$

$z.left = T.nil$

$z.right = T.nil$

$z.color = RED$

RB-INSERT-FIXUP(T, z)

TREE-INSERT(T, z)

1 $y = NIL$

2 $x = T.root$

3 **while** $x \neq NIL$

4 $y = x$

5 **if** $z.key < x.key$

6 $x = x.left$

7 **else** $x = x.right$

8 $z.p = y$

9 **if** $y == NIL$

10 $T.root = z$ // tree T was empty

11 **elseif** $z.key < y.key$

12 $y.left = z$

13 **else** $y.right = z$

Insertions

Start w/ BST Insertion

- 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

Which Properties might be Violated?

- Property 1: OK
 - Every node is either red or black.
- Property 2: The root is black.
 - If z is the root, then there's a violation.
 - Otherwise, OK.
- Property 3: OK.
 - Every leaf ($T.nil$) is black.
- Property 4.
 - If a node is red, then both its children are black.
 - No two reds in a row on a simple path from the root to a leaf.
 - If $z.p$ is red, there's a violation: both z and $z.p$ are red.
- Property 5: OK.
 - All simple paths from a node to descendant leaves contain the same number of black nodes.
- Remove the violation by calling RB-INSERT-FIXUP:

RB-Insert-Fixup(T, z)

```
RB-INSERT-FIXUP( $T, z$ )
  while  $z.p.color == \text{RED}$ 
    if  $z.p == z.p.p.left$ 
       $y = z.p.p.right$ 
      if  $y.color == \text{RED}$ 
         $z.p.color = \text{BLACK}$  // case 1
         $y.color = \text{BLACK}$  // case 1
         $z.p.p.color = \text{RED}$  // case 1
         $z = z.p.p$  // case 1
      else if  $z == z.p.right$ 
         $z = z.p$  // case 2
        LEFT-ROTATE( $T, z$ ) // case 2
         $z.p.color = \text{BLACK}$  // case 3
         $z.p.p.color = \text{RED}$  // case 3
        RIGHT-ROTATE( $T, z.p.p$ ) // case 3
      else (same as then clause with “right” and “left” exchanged)
     $T.root.color = \text{BLACK}$ 
```

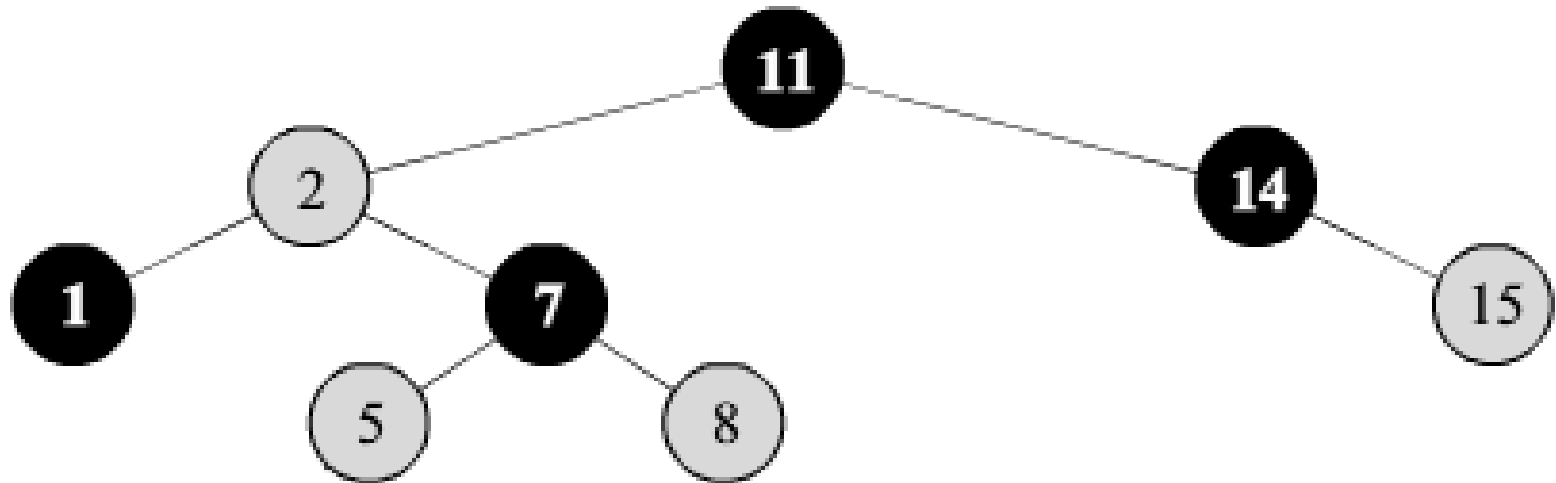
Loop Invariant

- At the start of each iteration of the **while** loop,
- z is red.
- There is at most one red-black violation:
 - Property 2: z is a red root, or
 - Property 4: z and $z.p$ are both red.

Loop Invariant

- **Initialization:** true from the insert.
- **Termination:** The loop terminates because $z.p$ 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 $z.p$ is the sentinel $T.nil$, 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 $z.p$ is a left child.
 - Let y be z 's uncle ($z.p$'s sibling).

Example: Insert (4)



- Insert 4
- Initialize Color=Red

Insertions

Start w/ BST Insertion

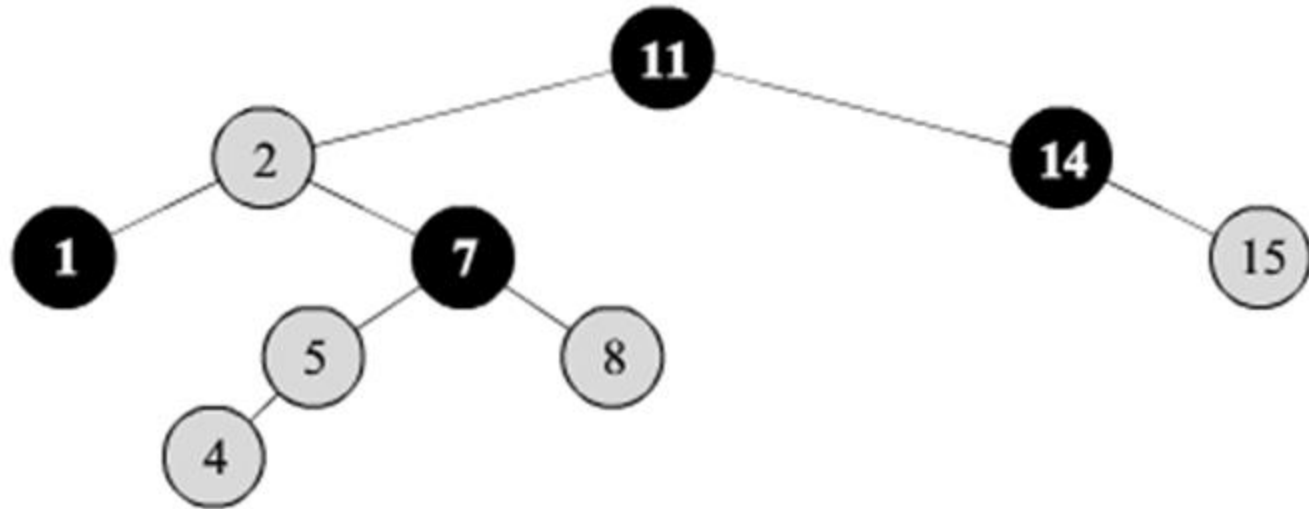
- RB-Insert ends by coloring the new node z RED
- Then it calls RB-Insert-Fixup because we could have violated a red-black property

Insertions

Start w/ BST Insertion

```
RB-INSERT(T, z)  
  y = T.nil  
  x = T.root  
  while x ≠ T.nil  
    y = x  
    if z.key < x.key  
      x = x.left  
    else x = x.right  
  z.p = y  
  if y == T.nil  
    T.root = z  
  elseif z.key < y.key  
    y.left = z  
  else y.right = z  
  z.left = T.nil  
  z.right = T.nil  
  z.color = RED  
  RB-INSERT-FIXUP(T, z)
```

Example: (4) Inserted



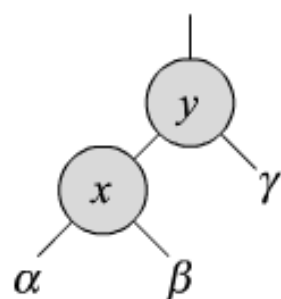
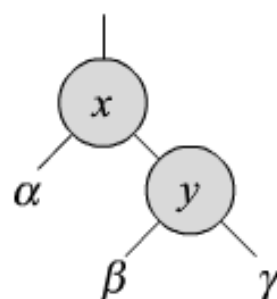
- 4 Inserted
- Color Initialized Red

Which Properties might be Violated?

- Property 1: OK
 - Every node is either red or black.
- Property 2: The root is black.
 - If z is the root, then there's a violation.
 - Otherwise, OK.
- Property 3: OK.
 - Every leaf ($T.nil$) is black.
- Property 4.
 - If a node is red, then both its children are black.
 - No two reds in a row on a simple path from the root to a leaf.
 - If $z.p$ is red, there's a violation: both z and $z.p$ are red.
- Property 5: OK.
 - All simple paths from a node to descendant leaves contain the same number of black nodes.
- Remove the violation by calling RB-INSERT-FIXUP:

RB-Insert-Fixup(T, z)

```
RB-INSERT-FIXUP( $T, z$ )
  while  $z.p.color == \text{RED}$ 
    if  $z.p == z.p.p.left$ 
       $y = z.p.p.right$ 
      if  $y.color == \text{RED}$ 
         $z.p.color = \text{BLACK}$  // case 1
         $y.color = \text{BLACK}$  // case 1
         $z.p.p.color = \text{RED}$  // case 1
         $z = z.p.p$  // case 1
      else if  $z == z.p.right$ 
         $z = z.p$  // case 2
        LEFT-ROTATE( $T, z$ ) // case 2
         $z.p.color = \text{BLACK}$  // case 3
         $z.p.p.color = \text{RED}$  // case 3
        RIGHT-ROTATE( $T, z.p.p$ ) // case 3
      else (same as then clause with “right” and “left” exchanged)
     $T.root.color = \text{BLACK}$ 
```

LEFT-ROTATE(T, x)RIGHT-ROTATE(T, y)LEFT-ROTATE(T, x)

```

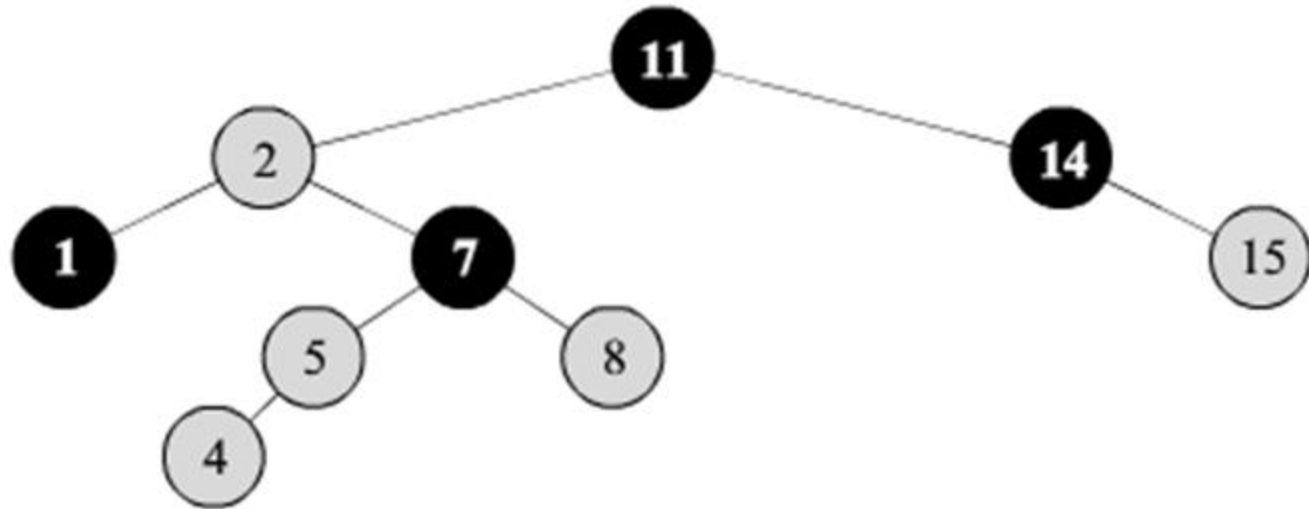
1  y = x.right           // set y
2  x.right = y.left      // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil
4      y.left.p = x
5  y.p = x.p             // link x's parent to y
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right = y
11 y.left = x            // put x on y's left
12 x.p = y

```

Loop Invariant

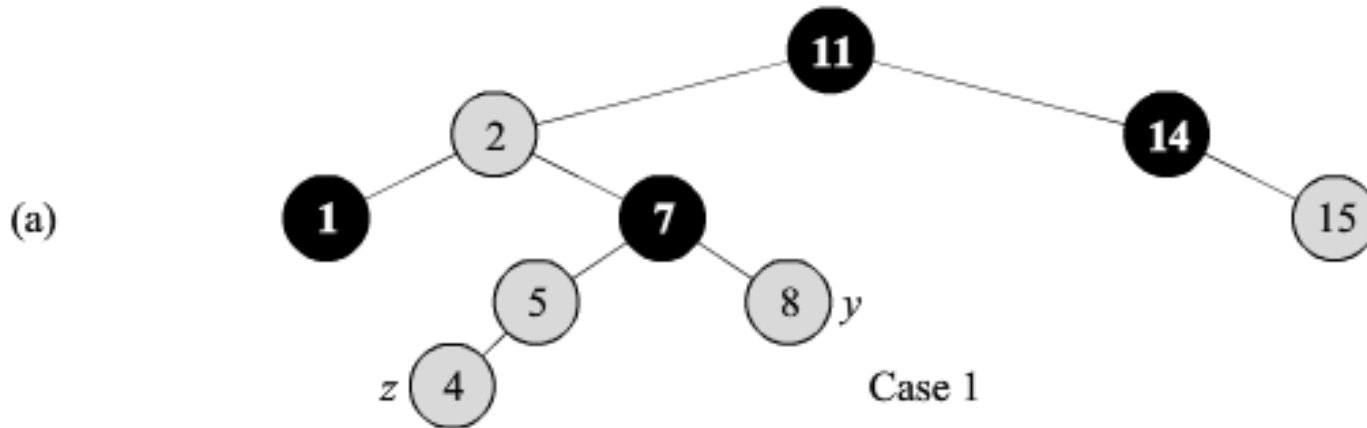
- At the start of each iteration of the **while** loop,
- z is red.
- There is at most one red-black violation:
 - Property 2: z is a red root, or
 - Property 4: z and $z.p$ are both red.

Example: (4) Inserted



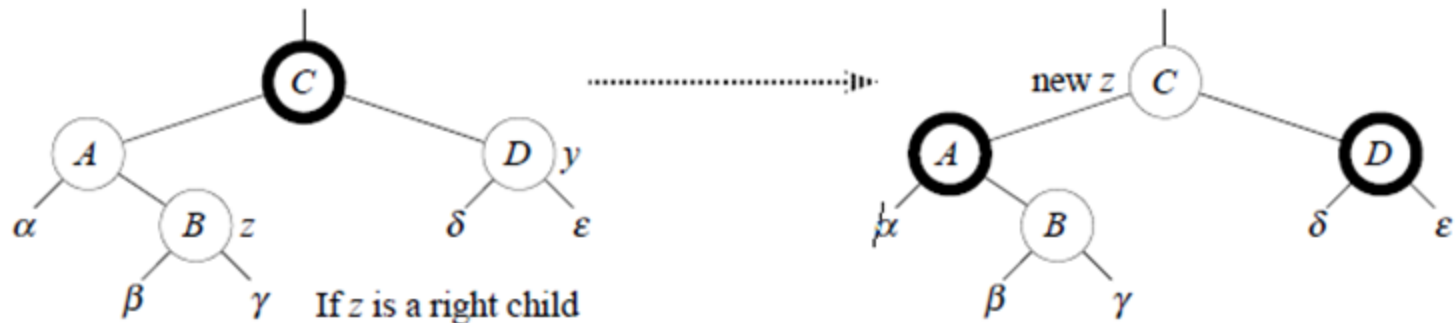
- 4 Inserted
- Color Initialized Red

Case 1: y is red



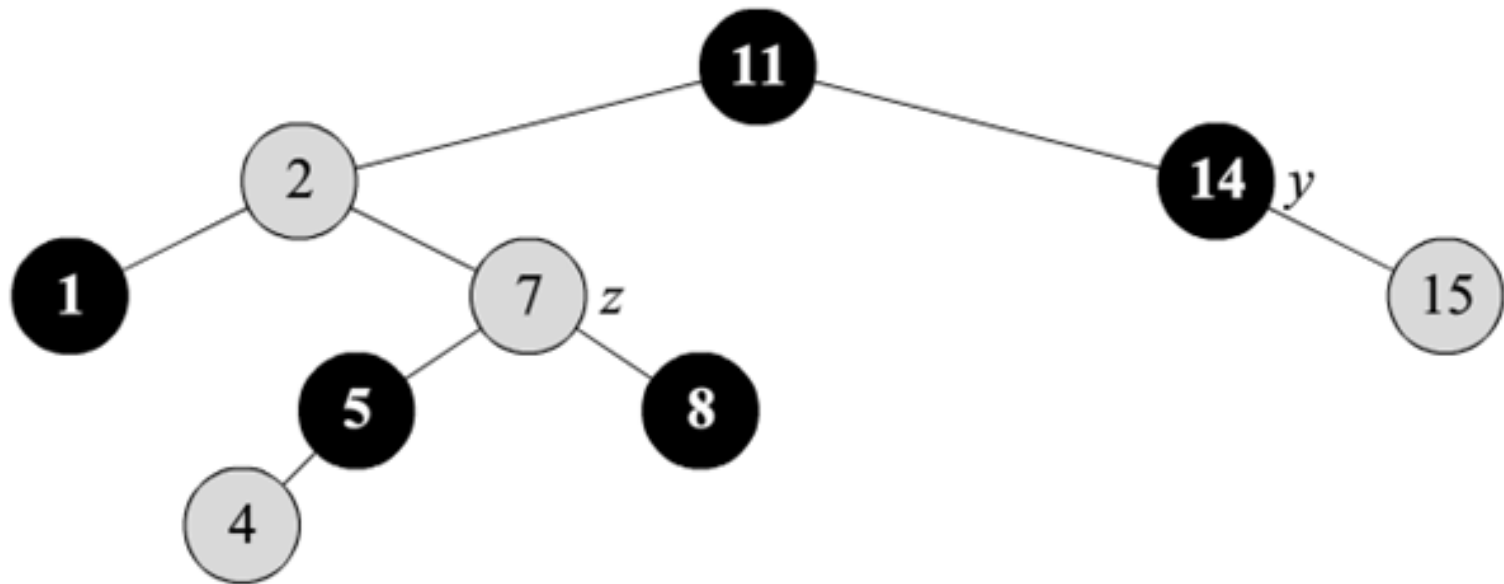
- y is z's red uncle!
- Plan:
 - Turn z's grandfather's children black!
 - Turn z's grandfather red – recurse from grandfather

Case 1: y is red

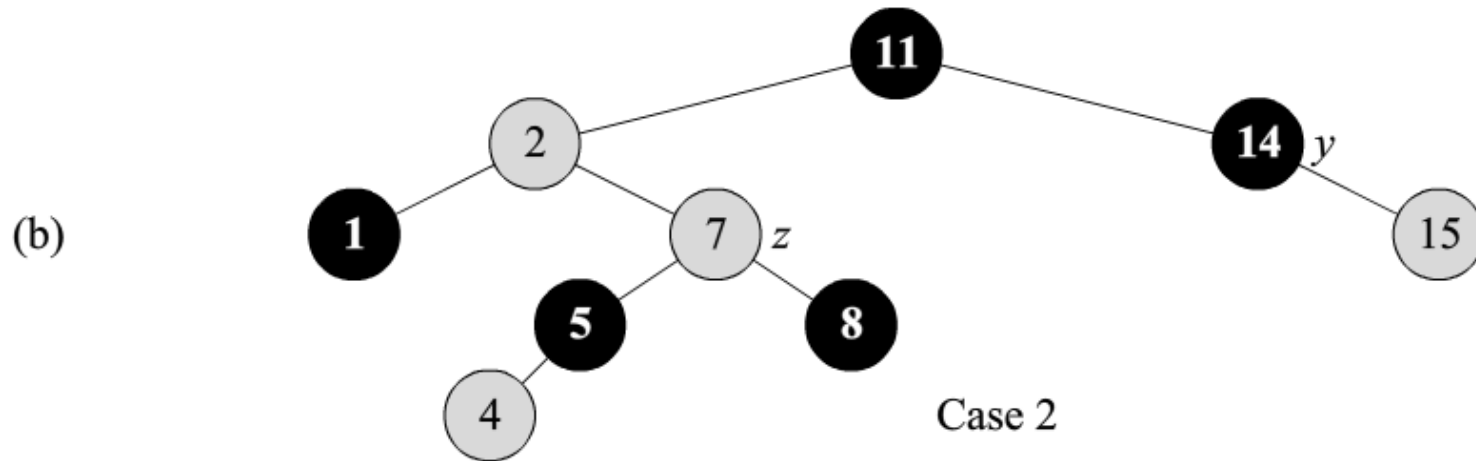


- $z.p.p$ (z 's grandparent) must be black, since z and $z.p$ are both red and there are no other violations of property 4.
- Make $z.p$ and y black \Rightarrow now z and $z.p$ are not both red. But property 5 might now be violated.
- Make $z.p.p$ red \Rightarrow restores property 5.
- The next iteration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

1 Iteration Completed

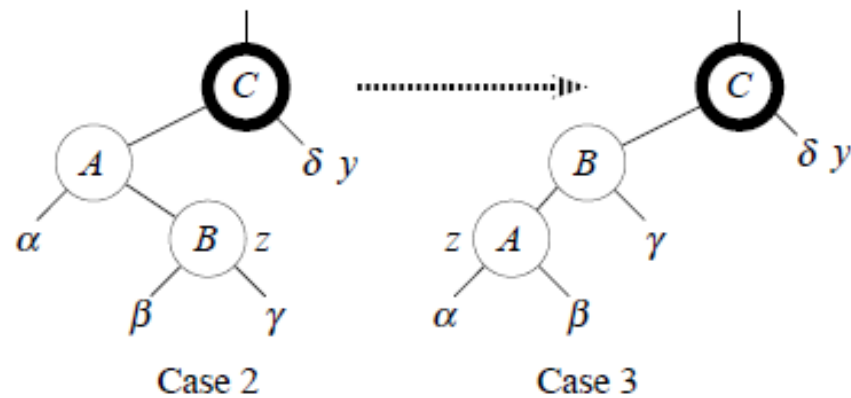


Now w/ Case 2: y is Black, z is Right Child



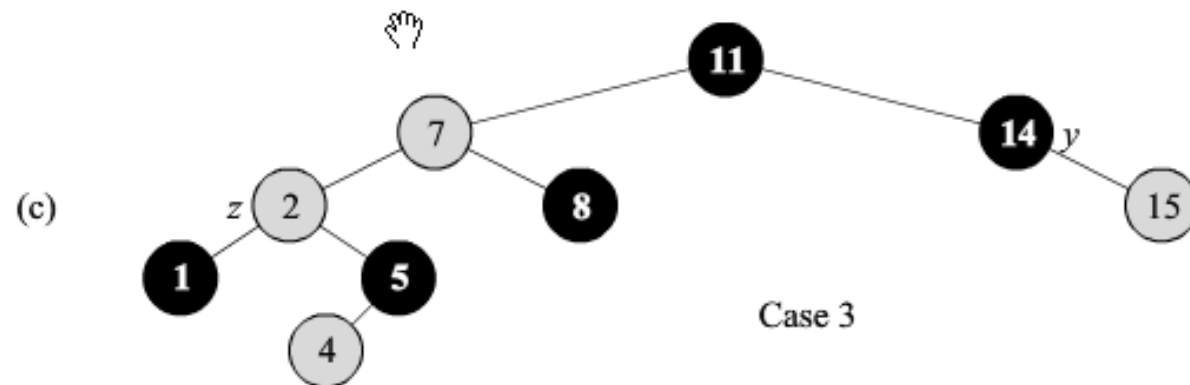
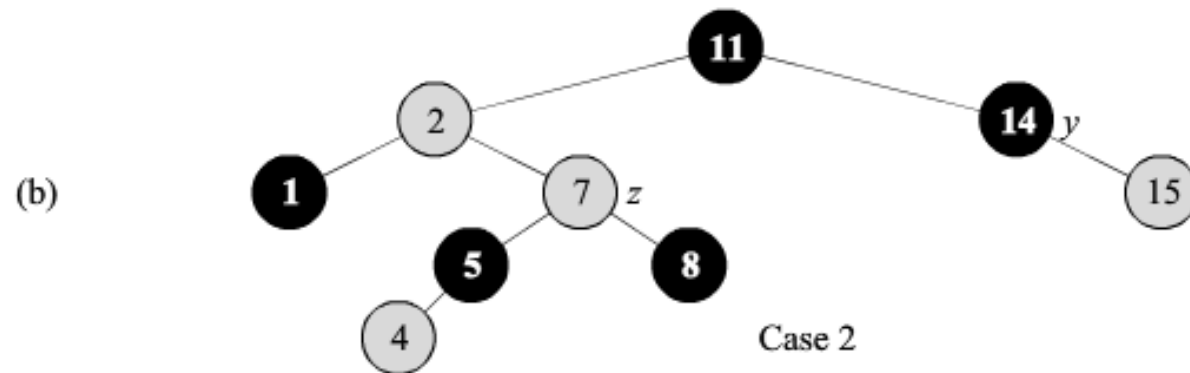
- z's uncle is black!
 - Need to adjust!

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

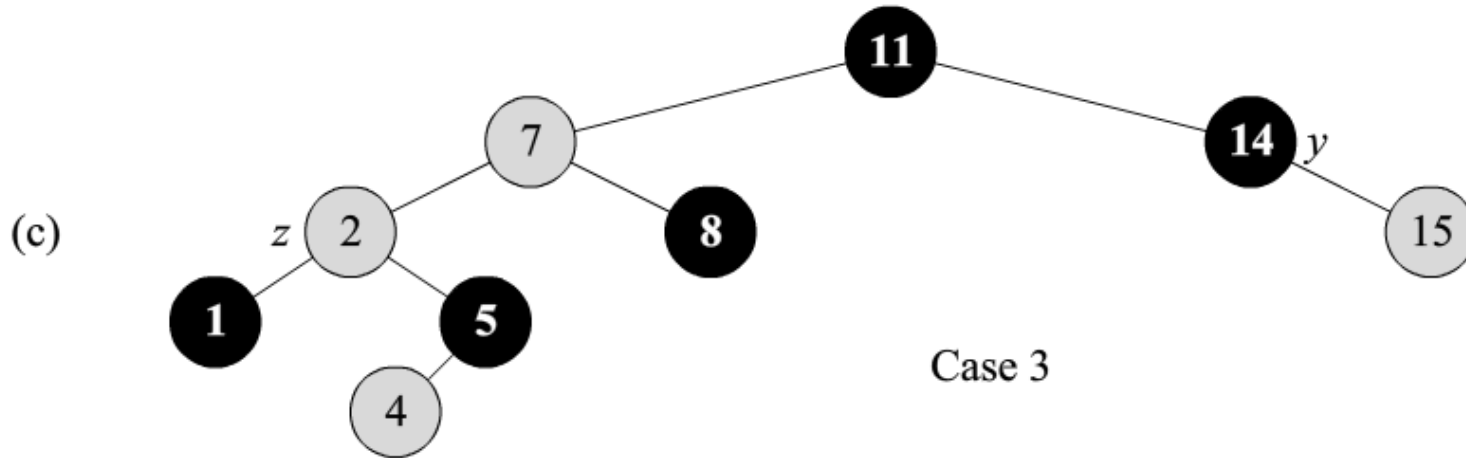


- Left rotate around $z.p \Rightarrow$ now z is a left child, and both z and $z.p$ are red.
- Takes us immediately to case 3.

Case 2: y is Black, z is Right Child



Case 3: y is Black, z is Left Child



- z's uncle is still black

Case 3

- **Case 3:** y is black, z is a left child
- Make $z.p$ black and $z.p.p$ red.
- Then right rotate on $z.p.p$.
- No longer have 2 reds in a row.
- $Z.p$ is now black \Rightarrow no more iterations.

Resulting Legal Red-Black Tree

