

Algorithms

Chapter 1&Chapter 2

Dr. Mohammed Salem Atoum

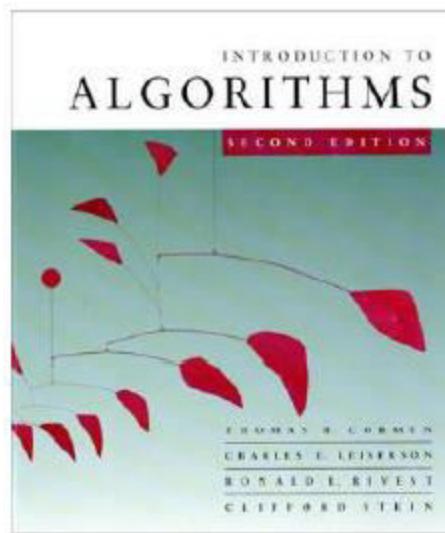
M.Atoum@inu.edu.jo

References

Introduction to Algorithms, Thomas H. Cormen, 2ed edition, MIT Press

Chapter 1

The Role of Algorithms in Computing



What are the Algorithms?

- Informally, an *algorithm* is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*. An algorithm is thus a sequence of computational steps that transform the input into output.
- It is a *tool* for *solving* a well-specified *computational problem*.



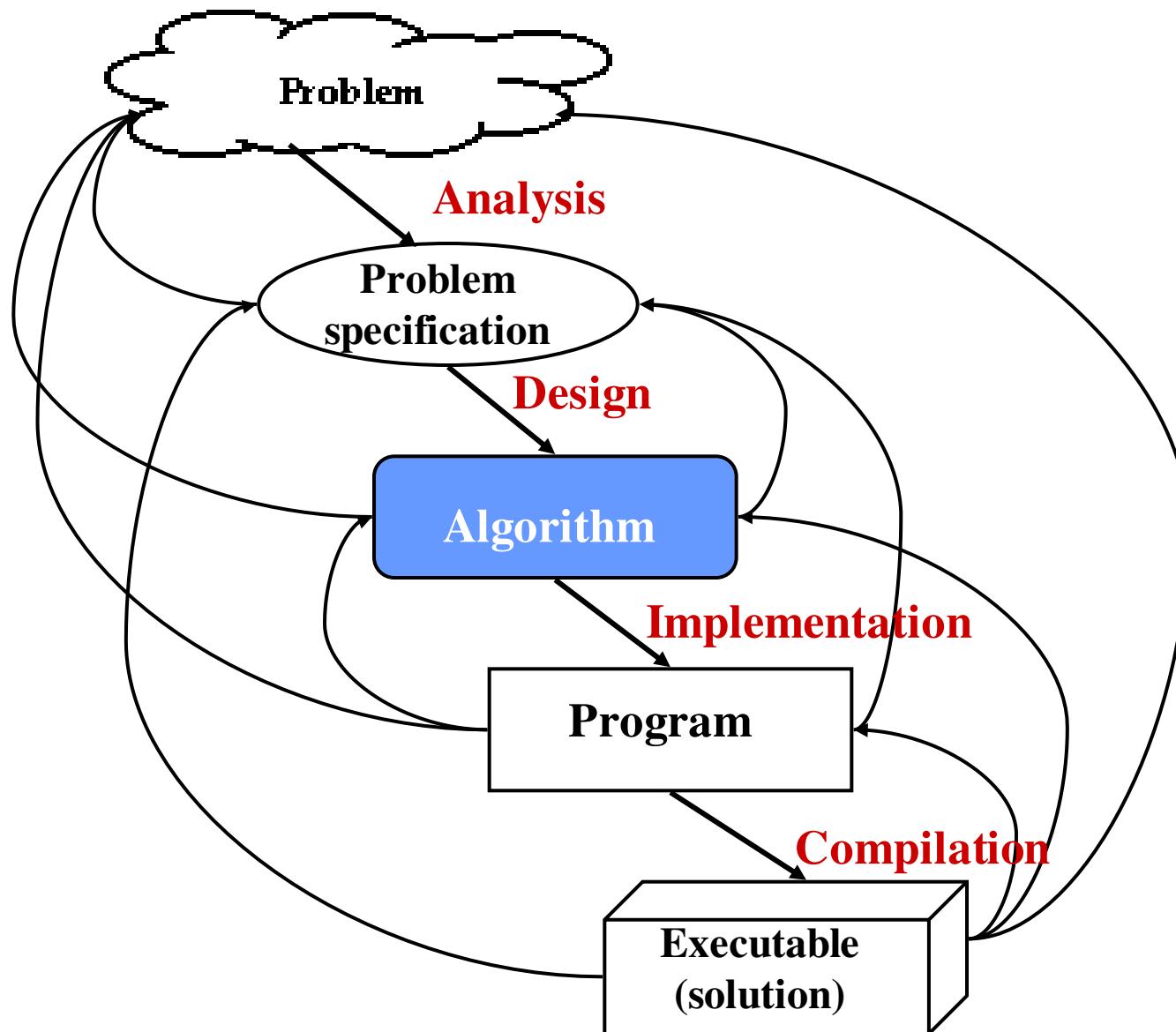
Correct and incorrect algorithms

- An algorithm is said to be **correct** if, for *every* input instance, it *halts* with the *correct* output
- An incorrect algorithm might:
 - **not halt** at all on some input, or
 - halt with an answer other than the desired one
- Algorithms must be:
 - **Correct**: For each input produce an appropriate output
 - **Efficient**: run as *quickly* as possible, and use as *little memory* as possible

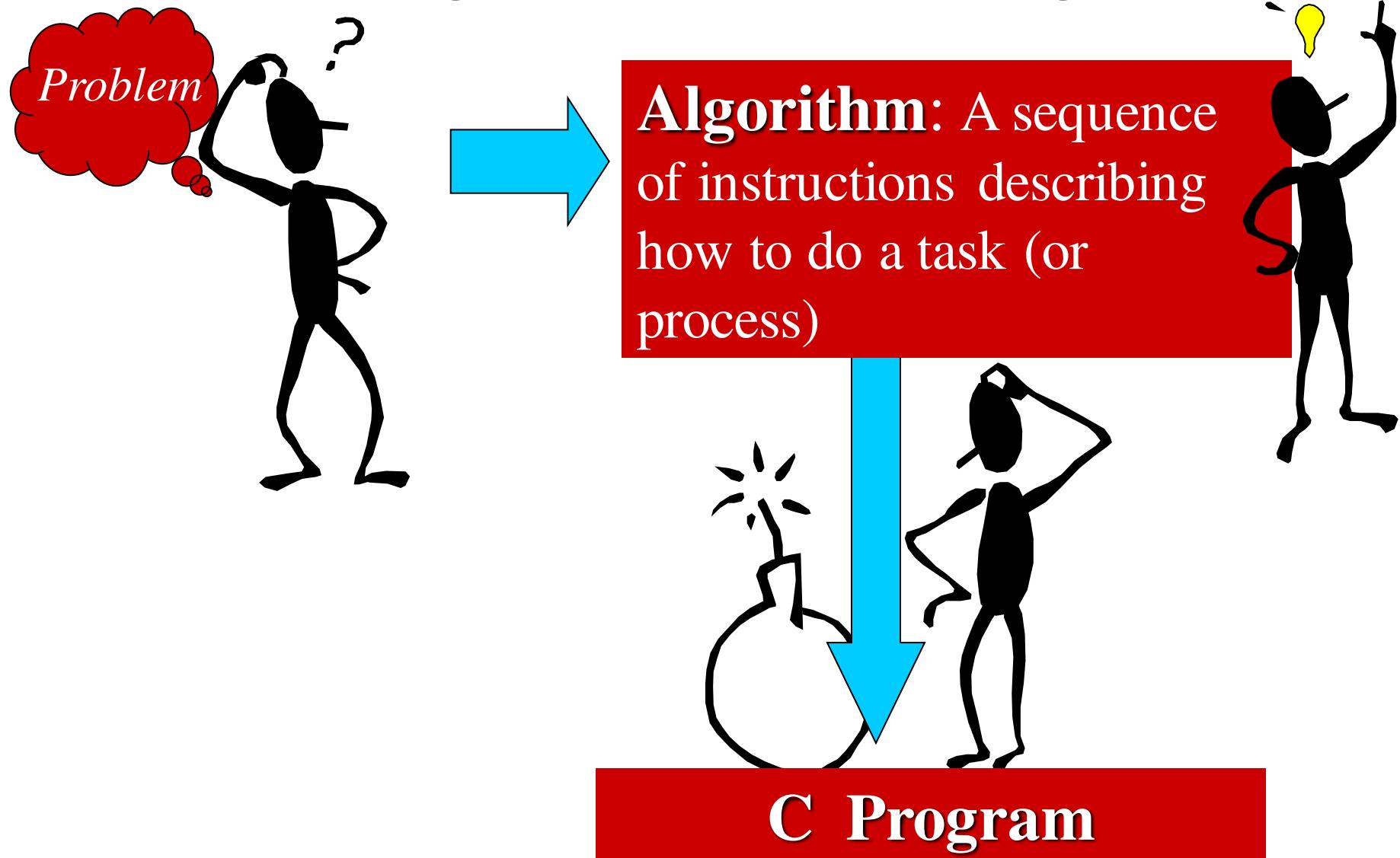
Example: sorting numbers.

- Input: A sequence of n numbers $\{a_3, a_1, a_2, \dots, a_n\}$
- Output: A reordered sequence of the input $\{a_1, a_2, a_3, \dots, a_n\}$ such that $a_1 \leq a_2 \leq a_3 \dots \leq a_n$.
- Input instance: $\{5, 2, 4, 1, 6, 3\}$.
- Output : $\{1, 2, 3, 4, 5, 6\}$.

The Problem-solving Process



From Algorithms to Programs,,,



A Simple Algorithm

- INPUT: a sequence of n numbers,
 - T is an array of n elements
 - $T[1], T[2], \dots, T[n]$
- OUTPUT: the smallest number among them

```
min = T[1]
for i = 2 to n do
{
    if T[i] < min
        min = T[i]
}
Output min
```

- Performance of this algorithm is a function of n

Describing Algorithms

- Algorithms can be implemented in any programming language
- Usually we use “pseudo-code” to describe algorithms
- Testing whether input n is prime:

```
for j = 2 to n-1
{
    if mod(n, j) = 0
        output "n is non prime"
        and halt
}
output "n is prime"
```

Why do we need the algorithms?

- One always need **to proof** that **his solution** method **terminate** and does the **correct answer**.
- Computers may be **fast** but they are not infinitely fast, and **memory** may be cheap but it is **not free**. This **resources** should be used wisely.

As an example:

- There are two algorithms for sorting; **merge** sort algorithm, and **insertion** sort algorithm
- Insertion sort takes an execution time equal $c_1 * n^2$ to sort n items.
- Merge sort takes an execution time equal $c_2 * n \log_2 n$ to sort n items.

Why do we need algorithms?- cont.

- c_1 and c_2 are constants.
- Insertion sort usually has a **smaller** constant factor than merge sort, so that $c_1 < c_2$
- Two computers; computer A running insertion sort, and computer B running merge sort.
- Assuming Computer A executes one billion instructions per second and computer B executes only ten million instructions per second, so that computer A is 100 times faster than computer B.

Why do we need algorithms?- cont.

- Suppose ($c_1 = 2$, and $c_2 = 50$).

-Algorithm Efficiency

- To sort one million number, computer A takes,
 $2 \times (10^6)^2$ instruction / 10^9 instruction/sec = 2000 sec.
- While computer B takes, $50 \times 10^6 \times \log_2 10^6$ instruction / 10^7 instruction/sec = 100 sec.
- By using algorithm whose running time grows more slowly, Computer B runs 20 times faster than Computer A
- **For ten million numbers**
 - **Insertion sort takes ≈ 2.3 days**
 - **Merge sort takes ≈ 20 minutes**
- Remark: We can **identify the Efficiency** of an algorithm from its **speed** (how long does the algorithm take to produce the result)

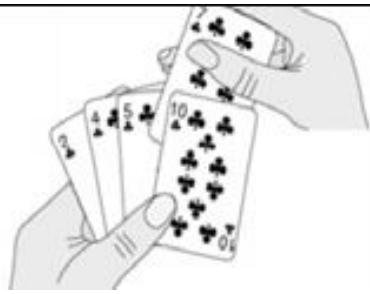
Chapter 2

Getting Started

Getting Started

- We will use the **insertion sort** algorithm to:
 - Define the pseudo code we are use.
 - we analyze its running time.
- Introduce the **divide-and conquer** approach to the design of algorithms and use it to develop an algorithm called **merge sort**.

Insertion sort



- Input: A sequence of n numbers $\{a_3, a_1, a_2, \dots, a_n\}$
- Output: A reordered sequence of the input $\{a_1, a_2, a_3, \dots, a_n\}$ such that $a_1 \leq a_2 \leq a_3 \dots \leq a_n$

Insertion Sort

Sorted array/list is by inserting one item at a time

- Simple to implement
- Efficient on small data sets
- Efficient on already almost ordered data sets

Insertion Sort

- Start with a sequence of size 1
- Repeatedly insert remaining elements



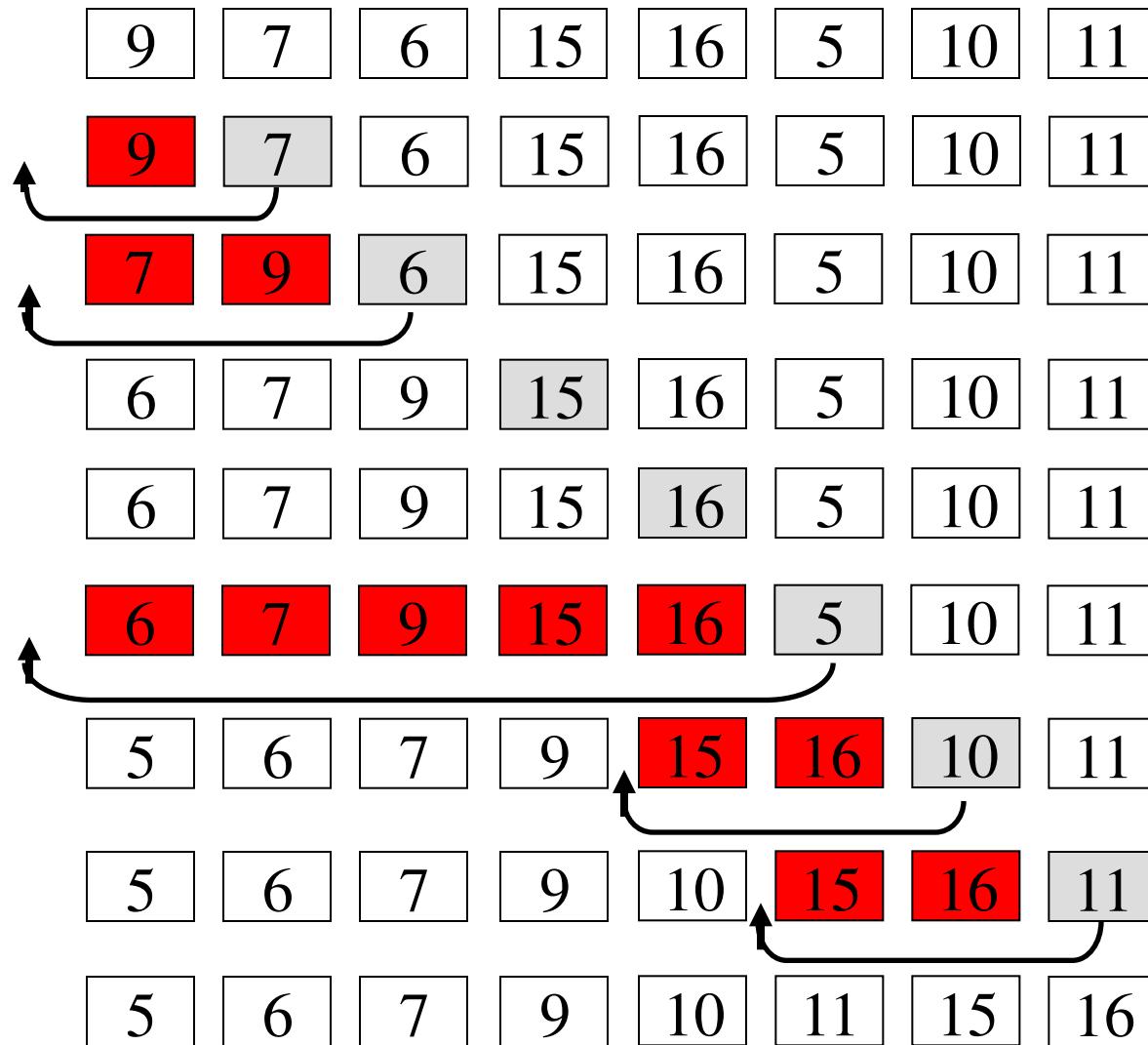
Insertion sort – pseudo code.

- The insertion algorithm

```
INSERTION-SORT(A)
1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2      do  $\text{key} \leftarrow A[j]$ 
3           $\triangleright$  Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ 
4           $i \leftarrow j - 1$ 
5          while  $i > 0$  and  $A[i] > \text{key}$ 
6              do  $A[i + 1] \leftarrow A[i]$ 
7                   $i \leftarrow i - 1$ 
8               $A[i + 1] \leftarrow \text{key}$ 
```

This algorithm takes as parameter an array of the sequence numbers to be sorted $A[1 \dots n]$, the number of elements in the array is denoted by $\text{length}[A]$. When this algorithm finish, the input array will contains the sorted sequence numbers

Insertion Sort Execution Example



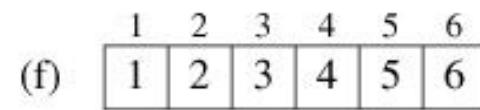
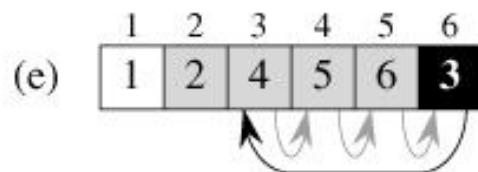
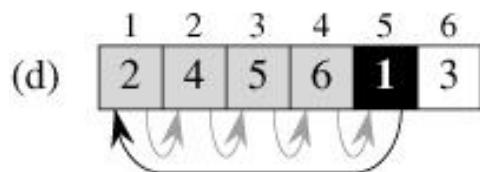
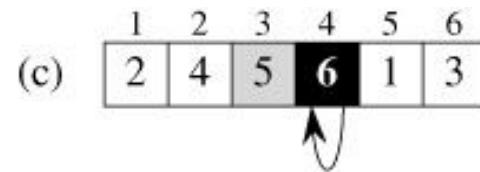
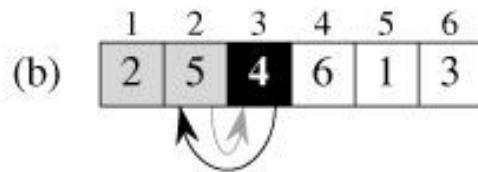
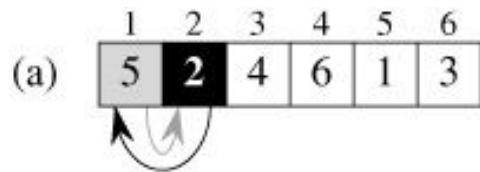
The insertion sort

We will use $A = \{5, 2, 4, 6, 1, 3\}$ as an input instance.

The index j indicate the element to be inserted.

$A[1\dots j-1]$ contains the sorted elements.

$A[j+1\dots n]$ contains the elements still to be sorted.



Pseudo-code conventions

-Algorithms are typically written in **pseudo-code** that is similar to **C/C++ and JAVA**.

We use the following conventions in our pseudocode

1. Indentation indicates block structure. For example, the body of the for loop that begins on line 1 consists of lines 2-8, and the body of the while loop that begins on line 5 contains lines 6-7 but not line 8.

2. The looping constructs **While**, **For** , and repeat and the conditional constructs

If, then and **else** .

5. Variables (such as i, j, and key) are **local** to the given procedure. We shall not use **global variables without** explicit indication.

Cont.....

6. Array elements are accessed by specifying the array name followed by the index in square brackets. For example, A[i] indicates the ith element of the array A. The notation "... is used to indicate a range of values within an array. Thus, A[1...j] indicates the sub-array of A consisting of the j elements A[1], A[2], . . . , A[j].
7. A particular attributes is accessed using the attributes name followed by the name of its object in square brackets.
For example, we treat an array as an object with the attribute length indicating how many elements it contains(length[A]).

Analysis of insertion sort

- The time taken by insertion sort depends on the input (sorting thousand items takes more time than sorting three items).
- In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input.
- Input size depends on the problem being studied

Analysis of insertion sort

- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.

A constant amount of time is required to execute each line of our pseudo-code. One line may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i , where c_i is a constant.

Analysis of insertion sort

- We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed.
- For each $j = 2, 3, \dots, n$, where $n = \text{length}[A]$, we let i be the number of times the while loop test in line 5 is executed for that value of j . When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.
- We assume that comments are not executable statements, and so they take no time.

Analysis of insertion sort

| INSERTION-SORT(A) | | cost times |
|-------------------|--|------------------------------------|
| 1 | for $j \leftarrow 2$ to $\text{length}[A]$ | $c_1 \quad n$ |
| 2 | do $\text{key} \leftarrow A[j]$ | $c_2 \quad n - 1$ |
| 3 | ▷ Insert $A[j]$ into the sorted sequence $A[1\dots j - 1]$. | $0 \quad n - 1$ |
| 4 | $i \leftarrow j - 1$ | $c_4 \quad n - 1$ |
| 5 | while $i > 0$ and $A[i] > \text{key}$ | $c_5 \quad \sum_{j=2}^n t_j$ |
| 6 | do $A[i + 1] \leftarrow A[i]$ | $c_6 \quad \sum_{j=2}^n (t_j - 1)$ |
| 7 | $i \leftarrow i - 1$ | $c_7 \quad \sum_{j=2}^n (t_j - 1)$ |
| 8 | $A[i + 1] \leftarrow \text{key}$ | $c_8 \quad n - 1$ |

Analysis of insertion sort – cont.

- The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and is executed n times will contribute $c_i n$ to the total running time.
- To compute $T(n)$, the running time of INSERTIONSORT, we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

Analysis of insertion sort – cont.

- For the INSERTION-SORT, the **best case occurs** if the array is already sorted. For each $j = 2, 3, \dots, n$, we then find that $A[i] \leq \text{key}$ in line 5 when i has its initial value of $j - 1$.
Thus $tj=1$ for $j = 2, 3, \dots, n$, and the best-case running time is:

(tj : is the number of time the while loop is executed for that value of j)

$$\begin{aligned}T(n) &= c1n + c2(n-1) + c4(n-1) + c5(n-1) + c8(n-1). \\&= (c1 + c2 + c4 + c5 + c8)n - (c2 + c4 + c5 + c8).\end{aligned}$$

- This running time can be expressed as $an + b$ for constants a and b that depend on the statement costs c_i ; it is thus a **linear function of n** .

Analysis of insertion sort – cont.

- If the array is in **reverse sorted** order—that is, in decreasing order—the **worst case** results. We must compare each element $A[j]$ with each element in the **entire sorted sub-array $A[1 \dots j - 1]$** , and so $t_j = j$ for $j = 2, 3, \dots, n$. Noting that

$$\sum_{j=2}^n j = \frac{n(n+1)}{2}, \quad \text{and} \quad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}.$$

This will result

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2}-1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

- This worst-case running time can be expressed as **$an^2 + bn + c$** for constants a , b , and c , that again depend on the statement **costs c_i** ; it is thus a **quadratic function of n** .

Worst-case and average-case analysis

- The worst-case running time of an algorithm is an **upper bound** on the running time for any input. Knowing it gives us a guarantee that the algorithm **will never take any longer**.
- The "**average case**" is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in sub-array $A[1\dots j-1]$ to insert element $A[j]$, On average, half the elements in $A[1\dots j-1]$ are less than $A[j]$, and half the elements are greater.

On average, therefore, we check half of the sub-array $A[1\dots j-1]$, so $t_j = j/2$. If we work out the resulting average-case running time, it turns out to be a **quadratic function** of the input size, just like the worst-case running time.

Algorithm Analysis

- Best Case
 - If A is sorted: $O(n)$ comparisons
- Worst Case
 - If A is reversed sorted: $O(n^2)$ comparisons
- Average Case
 - If A is randomly sorted: $O(n^2)$ comparisons

Designing algorithms

- There are many ways to design an algorithm.
- Insertion sort uses an **incremental approach**: having sorted the sub-array $A[1\dots j - 1]$, we insert the single element $A[j]$ into its proper place, yielding the sorted sub-array $A[1\dots j]$.
- Another approach to design is the **divide-and-conquer approach** which has a recursive structure to solve a given problem; they break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem.

Divide-and-Conquer

- Recursive in structure
 - *Divide* the problem into several smaller sub-problems that are similar to the original but smaller in size
 - *Conquer* the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
 - *Combine* the solutions to create a solution to the original problem

An Example: Merge Sort

- *Divide*: Divide the n -element sequence to be sorted into two subsequences of $n/2$ elements each
- *Conquer*: Sort the two subsequences recursively using merge sort.
- *Combine*: Merge the two sorted subsequences to produce the sorted answer.

Merge sort

- The procedure **MERGE-SORT(A, p, r)** sorts the elements in the sub-array A[p...r].

The divide step simply computes an index q that partitions A[p...r] into two sub-arrays: A[p...q], containing $n/2$ elements, and A[q + 1...r], containing $n/2$ elements.

- To sort the entire sequence $A = \{A[1], A[2], \dots, A[n]\}$, we make the initial call MERGE-SORT(A, 1, length[A]), where $\text{length}[A] = n$.

```
MERGE-SORT (A, p, r)
1 if p < r
2   then q ← ⌊(p + r)/2⌋
3     MERGE-SORT (A, p, q)
4     MERGE-SORT (A, q + 1, r)
5     MERGE (A, p, q, r)
```

Merge algorithm

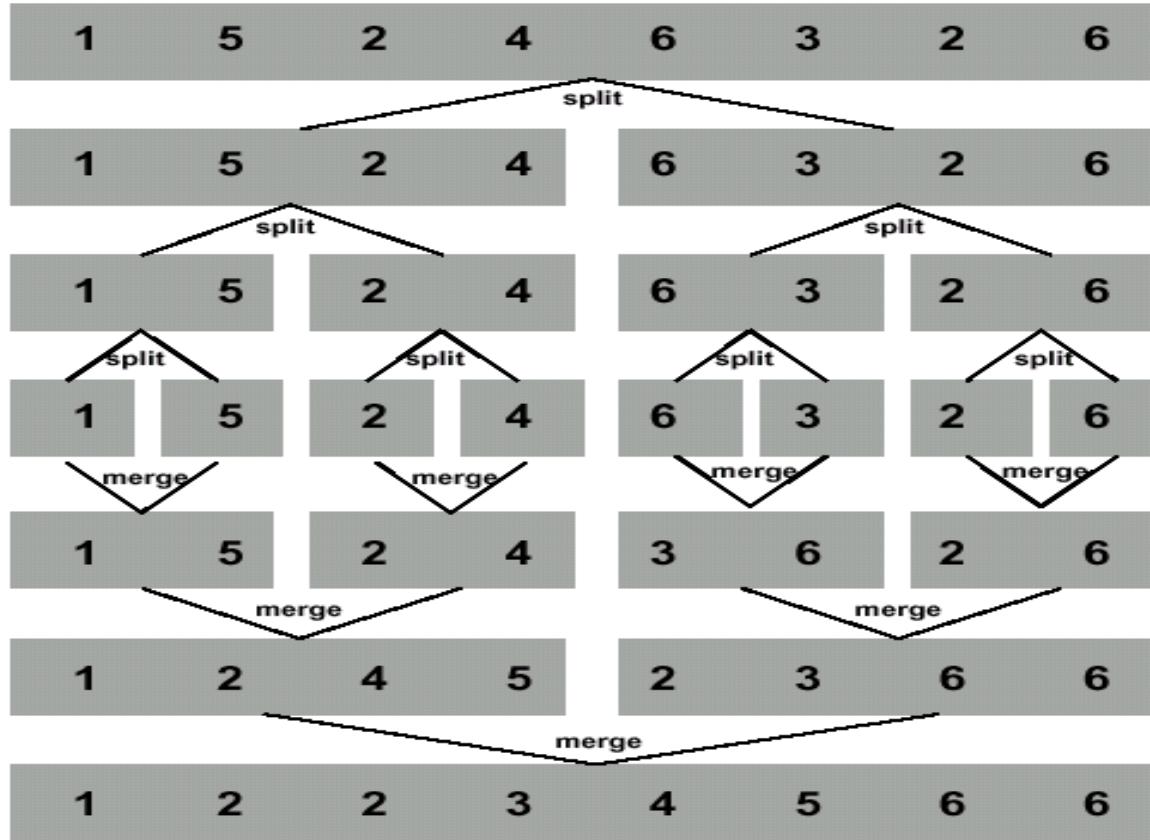
MERGE (A, p, q, r)

```
1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3   $\triangleright$  create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
4  for  $i \leftarrow 1$  to  $n_1$ 
5      do  $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7      do  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13     do if  $L[i] \leq R[j]$ 
14         then  $A[k] \leftarrow L[i]$ 
15              $i \leftarrow i + 1$ 
16         else  $A[k] \leftarrow R[j]$ 
17              $j \leftarrow j + 1$ 
```

Merge sort – cont.

- The operation of merge sort on the array $A = \{1, 5, 2, 4, 6, 3, 2, 6\}$.

Input:



Output.

Merge sort

- The key operation of the merge sort algorithm is the **merging of two sorted sequences in the "combine" step**. To perform the merging, we use an auxiliary procedure **MERGE(A, p, q, r)**, where A is an array and p, q, and r are **indices** numbering elements of the array such that $p \leq q < r$.
- The procedure assumes that the sub-arrays **A[p...q]** and **A[q + 1...r]** are in **sorted order**. It merges them to form a single sorted sub-array that replaces the current sub-array **A[p...r]**.

Merge algorithm – cont.

- The operation of lines 10-17 in the call MERGE(A, 9, 12, 16).

| | | | | | | | | | | | |
|---|-----|---|----|----|----|----|----|----|----|-----|-----|
| A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | ... |
| | ... | 2 | 4 | 5 | 7 | 1 | 2 | 3 | 6 | ... | |

k

| | | | | | | |
|---|---|---|---|---|----------|-----|
| L | 1 | 2 | 3 | 4 | 5 | ... |
| | 2 | 4 | 5 | 7 | ∞ | |

i

| | | | | | | |
|---|---|---|---|---|----------|-----|
| R | 1 | 2 | 3 | 4 | 5 | ... |
| | 1 | 2 | 3 | 6 | ∞ | |

j

(a)

| | | | | | | | | | | | |
|---|-----|---|----|----|----|----|----|----|----|-----|-----|
| A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | ... |
| | ... | 1 | 4 | 5 | 7 | 1 | 2 | 3 | 6 | ... | |

k

| | | | | | | |
|---|---|---|---|---|----------|-----|
| L | 1 | 2 | 3 | 4 | 5 | ... |
| | 2 | 4 | 5 | 7 | ∞ | |

i

| | | | | | | |
|---|---|---|---|---|----------|-----|
| R | 1 | 2 | 3 | 6 | ∞ | ... |
| | 1 | 2 | 3 | 6 | ∞ | |

j

(b)

| | | | | | | | | | | | |
|---|-----|---|----|----|----|----|----|----|----|-----|-----|
| A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | ... |
| | ... | 1 | 2 | 5 | 7 | 1 | 2 | 3 | 6 | ... | |

k

| | | | | | | |
|---|---|---|---|---|----------|-----|
| L | 1 | 2 | 3 | 4 | 5 | ... |
| | 2 | 4 | 5 | 7 | ∞ | |

i

| | | | | | | |
|---|---|---|---|---|----------|-----|
| R | 1 | 2 | 3 | 6 | ∞ | ... |
| | 1 | 2 | 3 | 6 | ∞ | |

j

(c)

| | | | | | | | | | | | |
|---|-----|---|----|----|----|----|----|----|----|-----|-----|
| A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | ... |
| | ... | 1 | 2 | 2 | 7 | 1 | 2 | 3 | 6 | ... | |

k

| | | | | | | |
|---|---|---|---|---|----------|-----|
| L | 1 | 2 | 3 | 4 | 5 | ... |
| | 2 | 4 | 5 | 7 | ∞ | |

i

| | | | | | | |
|---|---|---|---|---|----------|-----|
| R | 1 | 2 | 3 | 6 | ∞ | ... |
| | 1 | 2 | 3 | 6 | ∞ | |

j

(d)

Merge algorithm – cont.

- The operation of lines 10-17 in the call MERGE(A, 9, 12, 16)

| | | | | | | | | | | | | |
|-----|-----|-----|---|----|----|----------|-----|-----|----|----|-----|----------|
| | A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | |
| | | ... | 1 | 2 | 2 | 3 | 1 | 2 | 3 | 6 | ... | |
| | | | | | | | | | | | | k |
| L | | 1 | 2 | 3 | 4 | 5 | | | | | | |
| | | 2 | 4 | 5 | 7 | ∞ | | | | | | |
| | | i | | | | | R | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | j | | | | |

(e)

| | | | | | | | | | | | | |
|-----|-----|-----|---|----|----|----------|-----|-----|----|----|-----|----------|
| | A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | |
| | | ... | 1 | 2 | 2 | 3 | 4 | 2 | 3 | 6 | ... | |
| | | | | | | | | | | | | k |
| L | | 1 | 2 | 3 | 4 | 5 | | | | | | |
| | | 2 | 4 | 5 | 7 | ∞ | | | | | | |
| | | i | | | | | R | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | j | | | | |

(f)

| | | | | | | | | | | | | |
|-----|-----|-----|---|----|----|----------|-----|-----|----|----|-----|----------|
| | A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | |
| | | ... | 1 | 2 | 2 | 3 | 4 | 5 | 3 | 6 | ... | |
| | | | | | | | | | | | | k |
| L | | 1 | 2 | 3 | 4 | 5 | | | | | | |
| | | 2 | 4 | 5 | 7 | ∞ | | | | | | |
| | | i | | | | | R | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | j | | | | |

(g)

| | | | | | | | | | | | | |
|-----|-----|-----|---|----|----|----------|-----|-----|----|----|-----|----------|
| | A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | |
| | | ... | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 6 | ... | |
| | | | | | | | | | | | | k |
| L | | 1 | 2 | 3 | 4 | 5 | | | | | | |
| | | 2 | 4 | 5 | 7 | ∞ | | | | | | |
| | | i | | | | | R | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | j | | | | |

(h)

| | | | | | | | | | | | | |
|-----|-----|-----|---|----|----|----------|-----|-----|----|----|-----|----------|
| | A | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | |
| | | ... | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | ... | |
| | | | | | | | | | | | | k |
| L | | 1 | 2 | 3 | 4 | 5 | | | | | | |
| | | 2 | 4 | 5 | 7 | ∞ | | | | | | |
| | | i | | | | | R | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | 1 | 2 | 3 | 6 | ∞ |
| | | | | | | | | j | | | | |

(i)

Analysis of Merge Sort

- Divide: computing the middle takes $O(1)$
- Conquer: solving 2 sub-problem takes $2T(n/2)$
- Combine: merging n -element takes $O(n)$
- Total:

$$\begin{aligned} T(n) &= O(1) && \text{if } n = 1 \\ T(n) &= 2T(n/2) + O(n) + O(1) && \text{if } n > 1 \end{aligned}$$

$$\Rightarrow T(n) = O(n \lg n)$$

- Solving this recurrence (*how?*) gives $T(n) = O(n \lg n)$
 - This expression is a *recurrence*
- In chapter 4 we shall see how to solve common recurrences**

Algorithms and Computations Complexity

Chapter 3

Growth of Functions

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

How fast will your program run?

- The **running time** of your program will depend upon:
 - The algorithm
 - **The input**
 - Your implementation of the algorithm in a programming language
 - The compiler you use
 - The OS on your computer
 - Your computer hardware

Complexity

- Complexity is the number of steps required to solve a problem.
- The goal is to find the best algorithm to solve the problem with a less number of steps
- Complexity of Algorithms
 - The size of the problem is a measure of the quantity of the input data n
 - The **time** needed by an algorithm, expressed as a function of the size of the problem (it solves), is called the (**time**) **complexity** of the algorithm $T(n)$

Measures of Algorithm Complexity

- Often $T(n)$ depends on the input, in such cases one can talk about
 - **Worst-case** complexity,
 - **Best-case** complexity,
 - **Average-case** complexity of an algorithm
- Alternatively, one can determine bounds (**upper or lower**) on $T(n)$

Measures of Algorithm Complexity

- **Worst-Case Running Time:** the **longest** time for any input size of **n**
 - provides an **upper bound** on running time for any input
- **Best-Case Running Time:** the **shortest** time for any input size of **n**
 - provides **lower bound** on running time for any input
- **Average-Case Behavior:** the expected performance **averaged** over all possible inputs
 - it is generally **better** than worst case behavior, but **sometimes** it's roughly as bad as worst case
 - **difficult to compute**

Order of Growth

- For very large input size, it is the *rate of grow*, or *order of growth* that matters asymptotically
- We can ignore the *lower-order terms*, since they are relatively insignificant for very large n
- We can also ignore *leading term's constant coefficients*, since they are not as important for the rate of growth in computational efficiency for very large n
- Higher order functions of n are normally considered less efficient

Asymptotic Notation

- Θ, O, Ω
- Used to **describe the running times** of algorithms
- Instead of exact running time, say $\Theta(n^2)$
- Defined for functions whose domain is the set of natural numbers, N

Asymptotic Notation

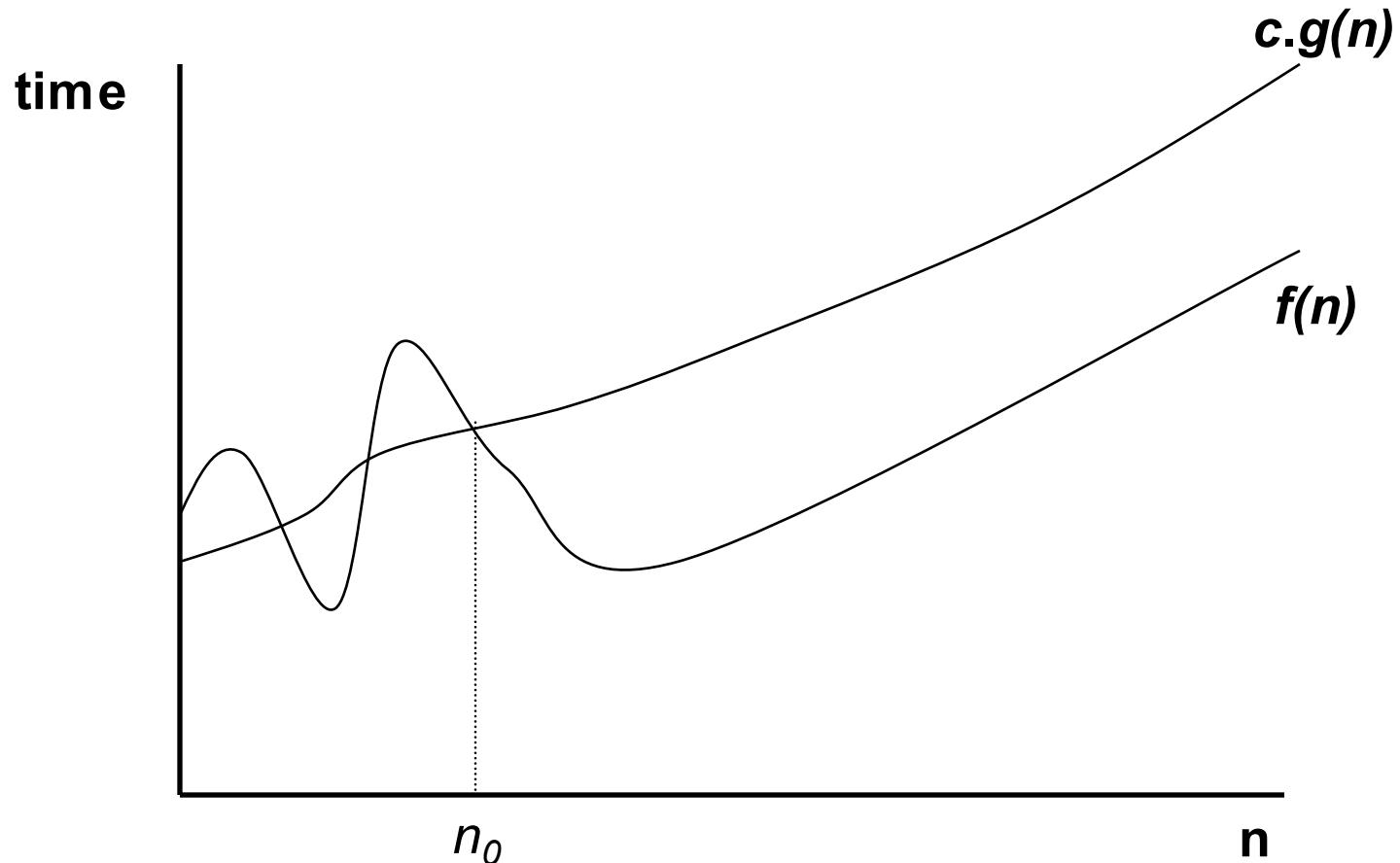
- Asymptotic (**big-O**) notation:
 - *What does $O(n)$ running time mean? $O(n^2)$? $O(n \log n)$?*

Big-O notation

(Upper Bound – Worst Case)

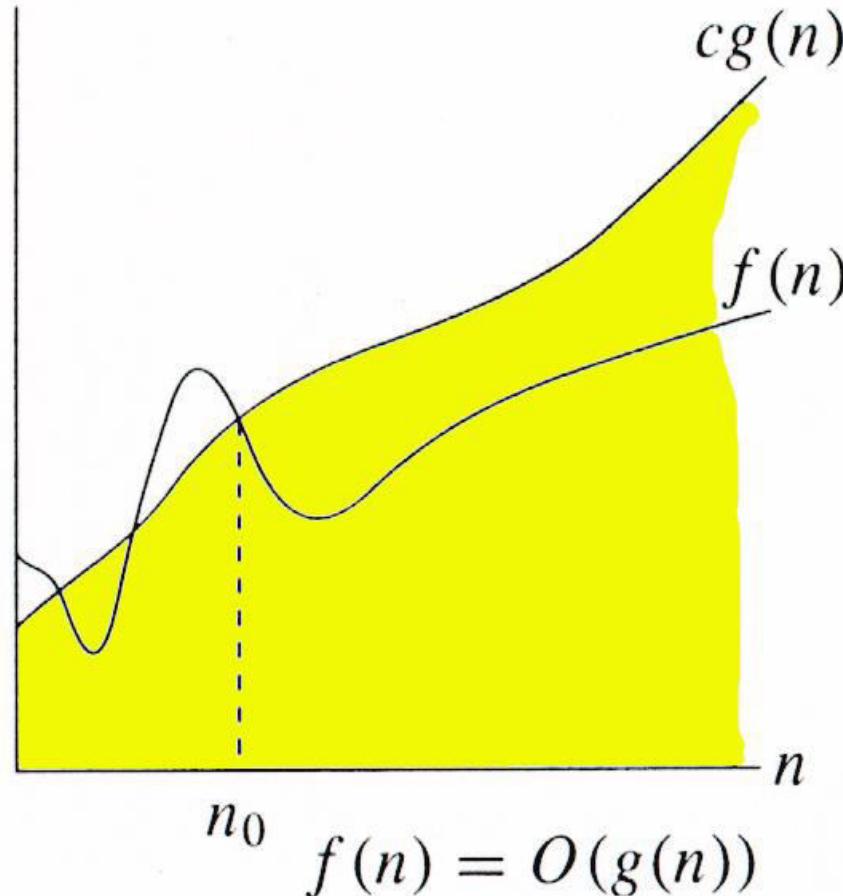
- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions
 - $O(g(n)) = \{f(n) : \text{there exist positive constants } c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$
- $O(g(n))$ means that as $n \rightarrow \infty$, the execution time $f(n)$ is at most $c.g(n)$ for some constant c

Big-O notation (Upper Bound – Worst Case)



$$f(n) = O(g(n))$$

O -notation



We say $g(n)$ is an *asymptotic upper bound* for $f(n)$

Big-O notation (Upper Bound – Worst Case)

- Example1: Is $2n + 7 = O(n)$?
- Let
 - $T(n) = 2n + 7$
 - $T(n) = n(2 + 7/n)$
 - Note for $n=7$;
 - $2 + 7/n = 2 + 7/7 = 3$
 - $T(n) \leq 3n$; $\forall n \geq 7$  
 - Then $T(n) = O(n)$
 - $\lim_{n \rightarrow \infty} [T(n) / n] = 2 \geq 0 \rightarrow T(n) = O(n)$

Big-O notation (Upper Bound – Worst Case)

○ Example2: Is $5n^3 + 2n^2 + n + 10^6 = O(n^3)$?

○ Let

□ $T(n) = 5n^3 + 2n^2 + n + 10^6$

□ $T(n) = n^3 (5 + 2/n + 1/n^2 + 10^6/n^3)$

□ Note for $n=100$;

➤ $5 + 2/n + 1/n^2 + 10^6/n^3 =$

➤ $5 + 2/100 + 1/10000 + 1 = 6.05$

□ $T(n) \leq 6.05 n^3 ; \quad \forall n \geq 100$ n_0



↑
c

□ Then $T(n) = O(n^3)$

□ $\lim_{n \rightarrow \infty} [T(n) / n^3] = 5 \geq 0 \rightarrow T(n) = O(n^3)$

Big-O notation

(Upper Bound – Worst Case)

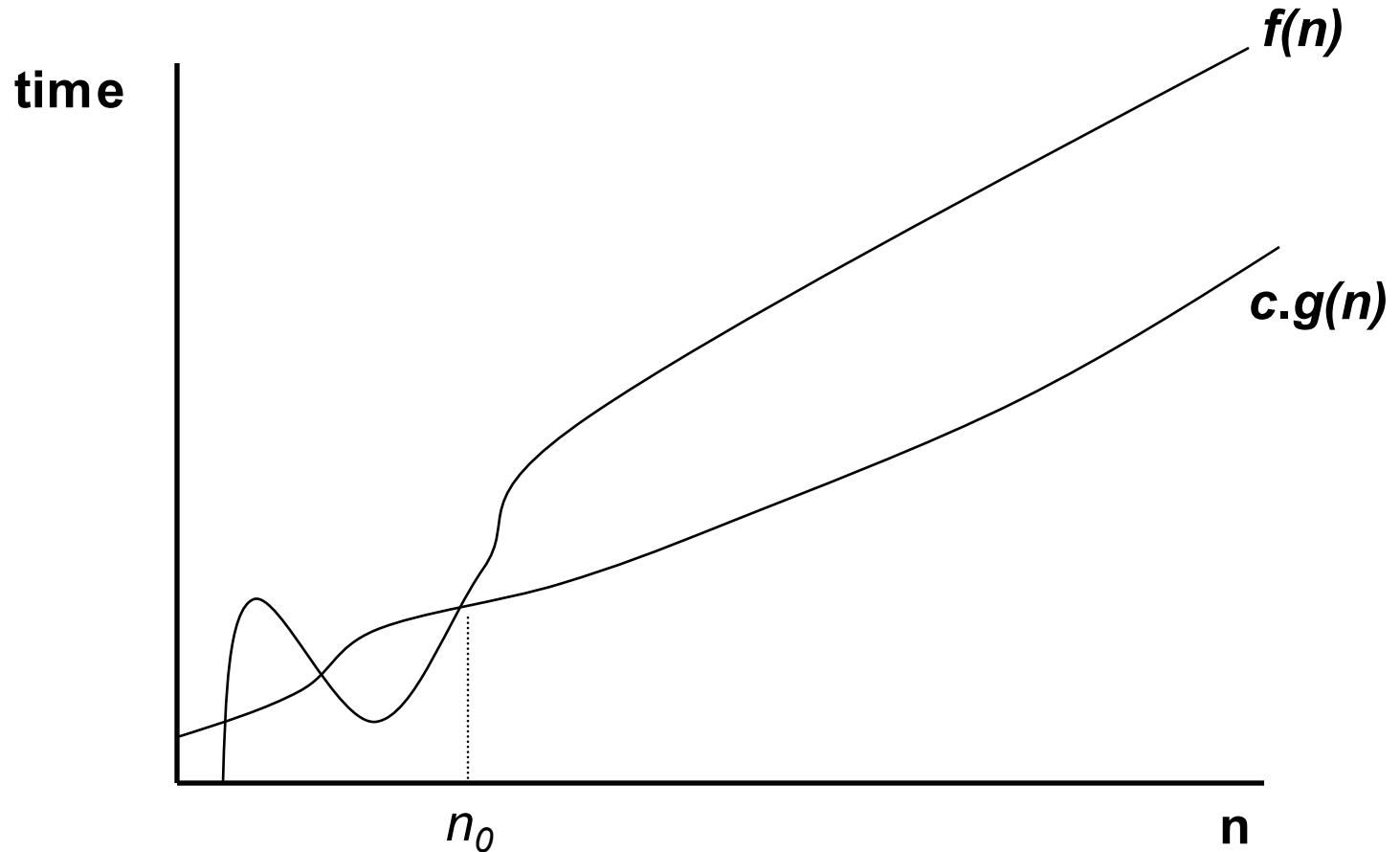
- Express the execution time as a function of the input size n
- Since only the growth rate matters, we can ignore the multiplicative **constants** and the **lower order terms**, e.g.,
 - $n, n+1, n+80, 40n, n+\log n$ is $O(n)$
 - $n^{1.1} + 10000000000n$ is $O(n^{1.1})$
 - n^2 is $O(n^2)$
 - $3n^2 + 6n + \log n + 24.5$ is $O(n^2)$
- $O(1) < O(\log n) < O((\log n)^3) < O(n) < O(n^2) < O(n^3) < O(n^{\log n}) < O(2^{\sqrt{n}}) < O(2^n) < O(n!) < O(n^n)$
- Constant < Logarithmic < Linear < Quadratic < Cubic < Polynomial < Factorial < Exponential

Ω -notation (Omega)

(Lower Bound – Best Case)

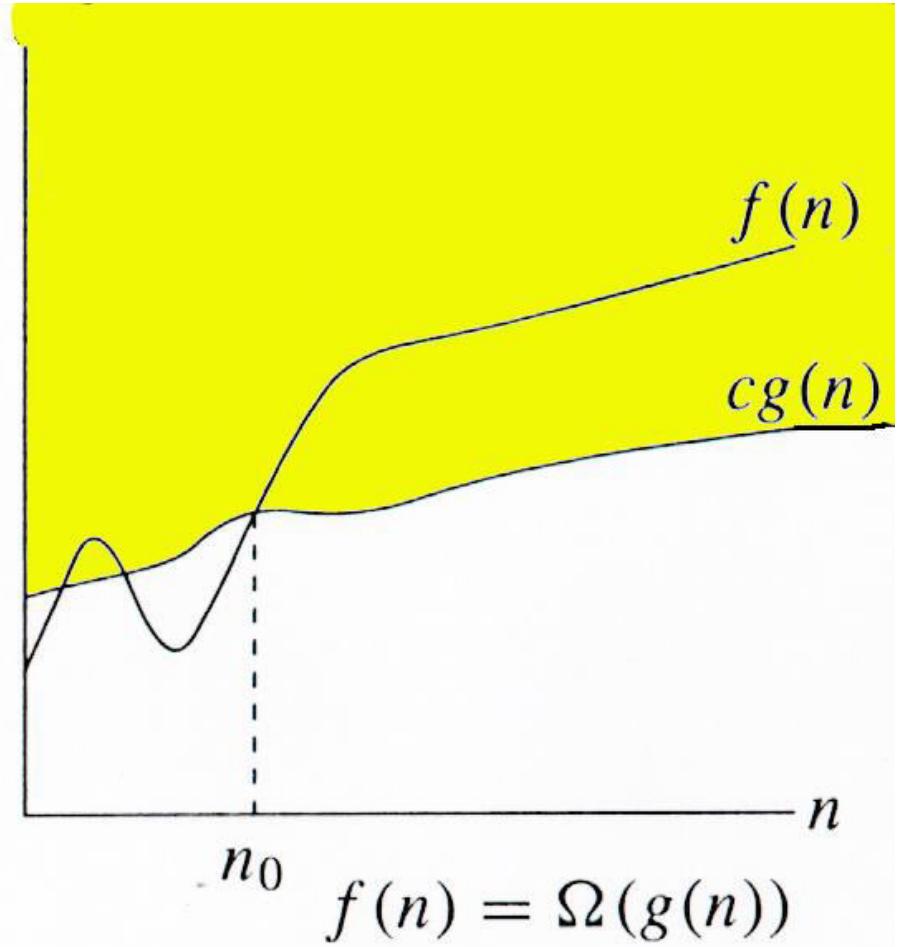
- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions
 - $\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$
- We say $g(n)$ is an ***asymptotic lower bound*** for $f(n)$.
- $\Omega(g(n))$ means that as $n \rightarrow \infty$, the execution time $f(n)$ is at **least $c.g(n)$** for some constant **c**

Ω -notation (Lower Bound – Best Case)



$$f(n) = \Omega(g(n))$$

Ω -notation



We say $g(n)$ is an **asymptotic lower bound** for $f(n)$

Θ notation (Theta)

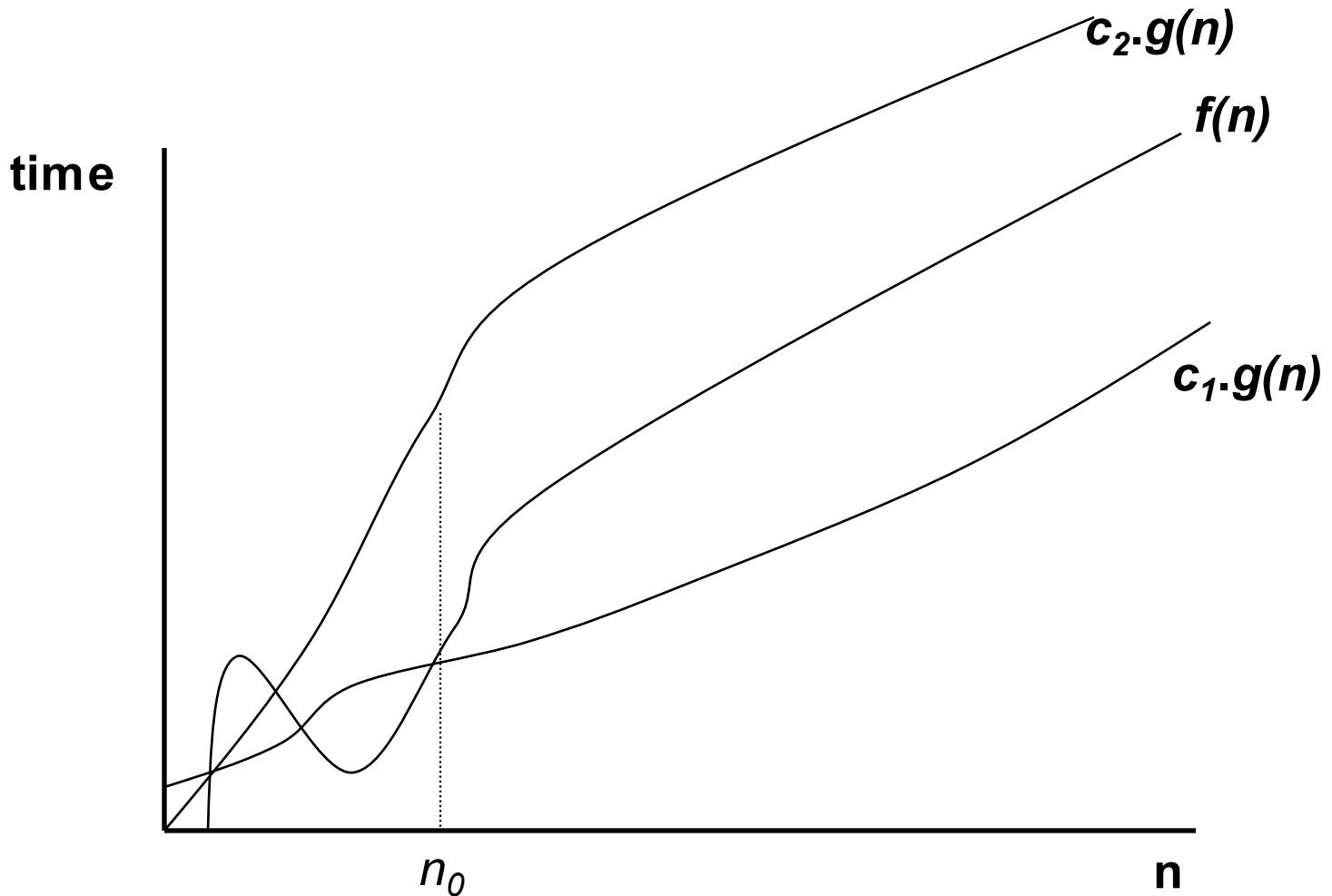
(Tight Bound)

- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions
 - $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1 > 0, c_2 > 0 \text{ and } n_0 > 0 \text{ such that}$
 - $c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}$

Θ notation (Theta) (Tight Bound)

- We say $g(n)$ is an *asymptotic tight bound* for $f(n)$.
- Theta notation
 - $\Theta(g(n))$ means that as $n \rightarrow \infty$, the execution time $f(n)$ is at **most** $c_2.g(n)$ and at **least** $c_1.g(n)$ for some constants c_1 and c_2 .

Θ notation (Theta) (Tight Bound)

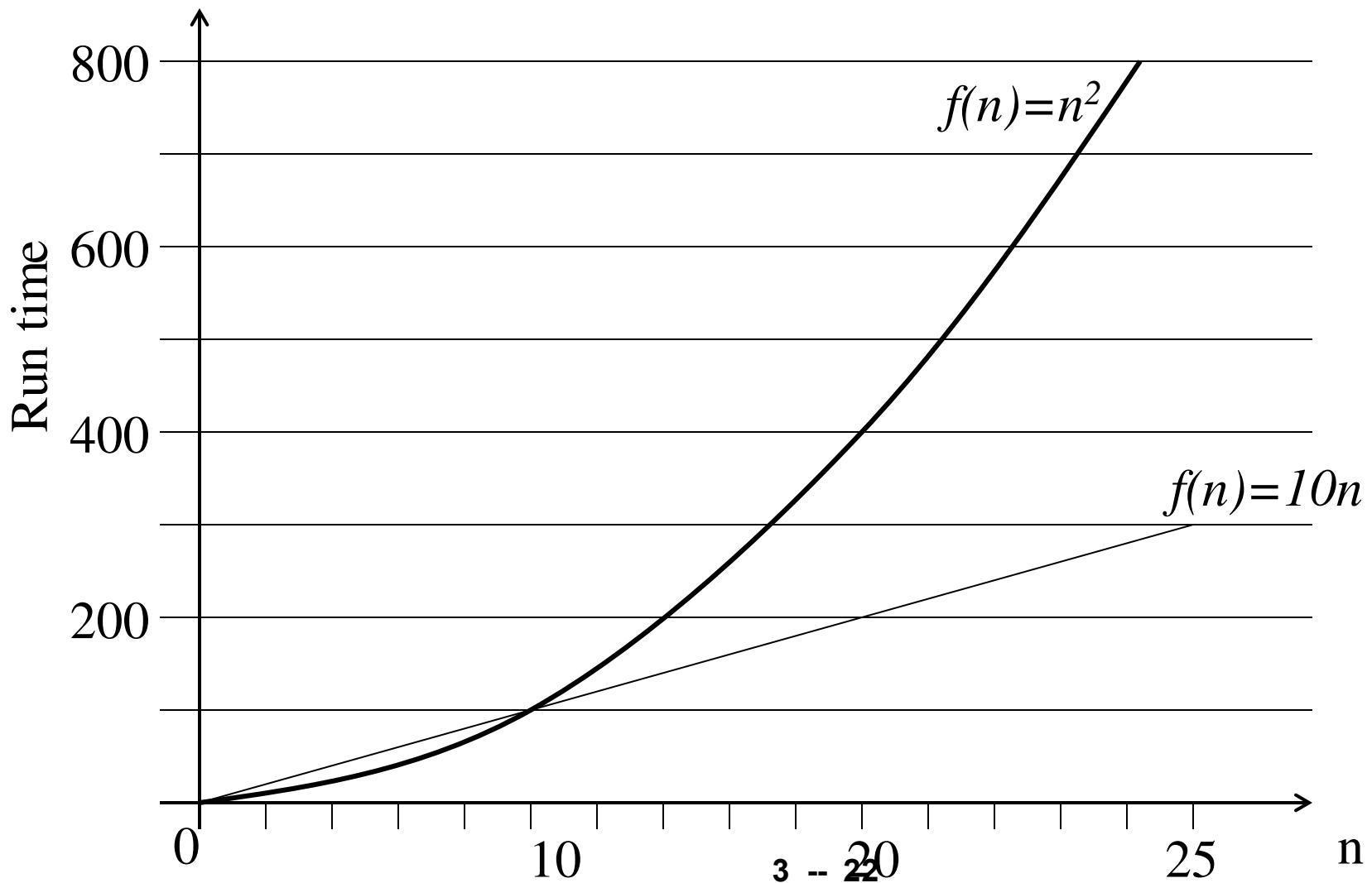


$$f(n) = \Theta(g(n))$$

Some Common Name for Complexity

| | |
|-----------------------|------------------|
| $O(1)$ | Constant time |
| $O(\log n)$ | Logarithmic time |
| $O(\log^2 n)$ | Log-squared time |
| $O(n)$ | Linear time |
| $O(n^2)$ | Quadratic time |
| $O(n^3)$ | Cubic time |
| $O(n^i)$ for some i | Polynomial time |
| $O(2^n)$ | Exponential time |

Effect of Multiplicative Constant



Floors & Ceilings

- For any **real** number x , we denote the **greatest integer** *less than or equal* to x by $\lfloor x \rfloor$
 - read “the floor of x ”
- For any **real** number x , we denote the **least integer greater than or equal** to x by $\lceil x \rceil$
 - read “the ceiling of x ”
- For all real x , (example for $x=4.2$)
 - $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- For any integer n ,
 - $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$

Polynomials

- Given a positive **integer d** , a **polynomial** in n of **degree d** is a function $P(n)$ of the form
 - $P(n) = \sum_{i=0}^d a_i n^i$
 - where a_0, a_1, \dots, a_d are coefficient of the polynomial
 - $a_d \neq 0$
- A polynomial is **asymptotically positive** iff $a_d > 0$
 - Also $P(n) = \Theta(n^d)$

Exponents

- $x^0 = 1$
- $x^1 = x$
- $x^{-1} = 1/x$

- $x^a \cdot x^b = x^{a+b}$

- $x^a / x^b = x^{a-b}$

- $(x^a)^b = (x^b)^a = x^{ab}$

- $x^n + x^n = 2x^n \neq x^{2n}$

- $2^n + 2^n = 2 \cdot 2^n = 2^{n+1}$

Logarithms (1)

- In computer science, all logarithms are to **base 2** unless specified otherwise
- $x^a = b \text{ iff } \log_x(b) = a$
- $\lg(n) = \log_2(n)$
- $\lg^k(n) = (\lg(n))^k$
- $\log_a(b) = \log_c(b) / \log_c(a) ; \quad c > 0$
- $\lg(ab) = \lg(a) + \lg(b)$
- $\lg(a/b) = \lg(a) - \lg(b)$
- $\lg(a^b) = b \cdot \lg(a)$

Logarithms (2)

- $a = b^{\log_b(a)}$
- $a^{\log_b(n)} = n^{\log_b(a)}$
- $\lg(1/a) = -\lg(a)$
- $\log_b(a) = 1/\log_a(b)$
- $\lg(n) < n$ for all $n > 0$
- $\log_a(a) = 1$
- $\lg(1) = 0, \lg(2) = 1, \lg(1024=2^{10}) = 10$
- $\lg(1048576=2^{20}) = 20$

Summation

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = n(n+1)/2 = \Theta(n^2)$$

$$\sum_{k=0}^n x^k = 1 + x + x^2 \dots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

$$\sum_{k=1}^n (ca_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$$

$$\sum_{k=1}^n (a_k - a_{k-1}) = a_n - a_0, \text{ for } a_0, a_1, \dots, a_n$$

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n, \text{ for } a_0, a_1, \dots, a_n$$

Summation

- Constant Series: For $a, b \geq 0$,

$$\sum_{i=a}^b 1 = b - a + 1$$

Series

$$\sum_{i=0}^N 2^i = 2^{N+1} - 1$$

$$\sum_{i=0}^N A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=1}^N i = \frac{N(N + 1)}{2} \approx \frac{N^2}{2}$$

Proof of Geometric series

Proofs for geometric series are done by cancellation, as demonstrated.

Proof

$$S = 1 + \cancel{A} + \cancel{A^2} + \cancel{A^3} + \cancel{A^4} + \cancel{A^5} + \dots$$

$$AS = \cancel{A} + \cancel{A^2} + \cancel{A^3} + \cancel{A^4} + \cancel{A^5} + \dots$$

$$S - AS = 1$$

$$S = \frac{1}{1 - A}$$

Factorials

- $n!$ (“n factorial”) is defined for integers $n \geq 0$ as
- $n! = \begin{cases} 1 & \text{if } n=0, \\ n.(n-1)! & \text{if } n>0 \end{cases}$
- $n! = 1 \cdot 2 \cdot 3 \dots n$
- $n! < n^n$ for $n \geq 2$

Proofs by Counterexample & Contradiction

الإثبات عن طريق التجربة و التناقض

- There are several ways to **prove** a theorem:
 - **Counterexample:**
 - By providing an example of in which the theorem **does not hold**, you prove the theory to be **false**.
 - For example: All multiples of 5 are even. However 3×5 is 15, which is odd. The theorem is false.
 - **Contradiction:**
 - Assume the theorem to be true. If the assumption **implies** that some known property is **false**, then the theorem **CANNOT** be **true**.

الاستقراء Proof by Induction

- Proof by **induction** has three standard parts:
 - The first step is proving a **base case**, that is, establishing that a theorem is true for some small value(s), this step is almost always **trivial**.
 - Next, an **inductive hypothesis** is assumed. Generally this means that the theorem is assumed to be true for all cases up to some limit n .
 - Using this assumption, the theorem is then shown to be true for the next **value**, which is typically $n+1$ (**induction step**). This proves the theorem (as long as n is finite).

Example: Proof By Induction

- Claim: $S(n)$ is true for all $n \geq k$
 - Basis:
 - Show formula is true when $n = k$
 - Inductive hypothesis:
 - Assume formula is true for an arbitrary n
 - Induction Step:
 - Show that formula is then true for $n+1$

Induction Example: Gaussian Closed Form

- Prove $1 + 2 + 3 + \dots + n = n(n+1) / 2$

- Basis:

- If $n = 0$, then $0 = 0(0+1) / 2 = 0$

- Inductive hypothesis:

- Assume $1 + 2 + 3 + \dots + n = n(n+1) / 2$

- Step (show true for $n+1$):

$$1 + 2 + \dots + n + n+1 = (1 + 2 + \dots + n) + (n+1)$$

$$= n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2$$

$$= (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2$$

Induction Example: Geometric Closed Form

- Prove
- $a^0 + a^1 + \dots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
 - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$
 $a^0 = 1 = (a^1 - 1)/(a - 1) = 1$
 - Inductive hypothesis:
 - Assume $a^0 + a^1 + \dots + a^n = (a^{n+1} - 1)/(a - 1)$
 - Step (show true for $n+1$):
$$\begin{aligned} a^0 + a^1 + \dots + a^{n+1} &= a^0 + a^1 + \dots + a^n + a^{n+1} \\ &= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1) \end{aligned}$$

Heap Sort

Heap Sort

This algorithm is mainly based on Heap Tree construction

Sort the following array :

15, 25, 13, 12, 26, 9, 16, 30

Step 1 : Construct the Heap Tree

Step 2 : Delete root node and replace it with the last leaf node of the tree

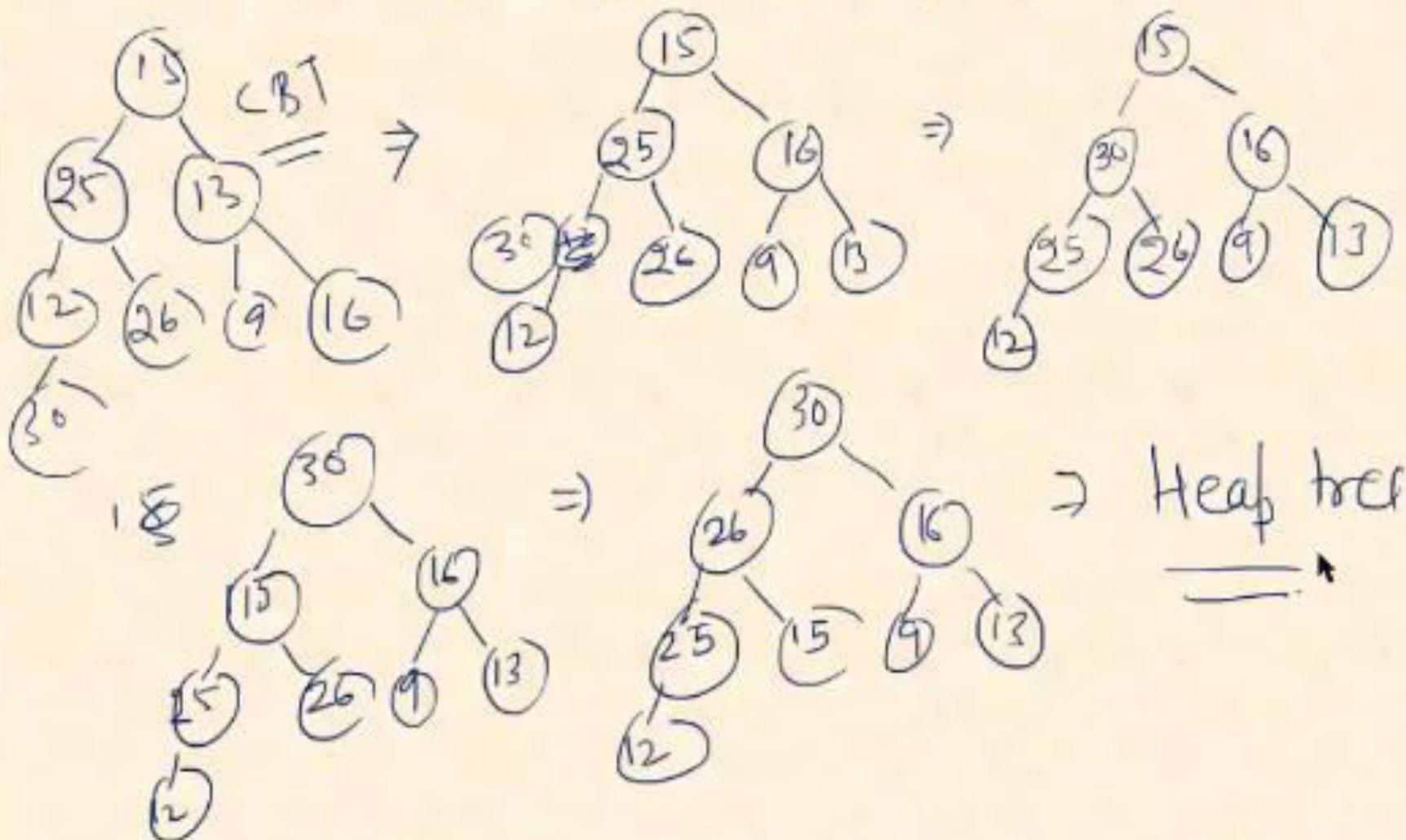
Step 3 : Adjust CBT so that it satisfies Heap tree property.



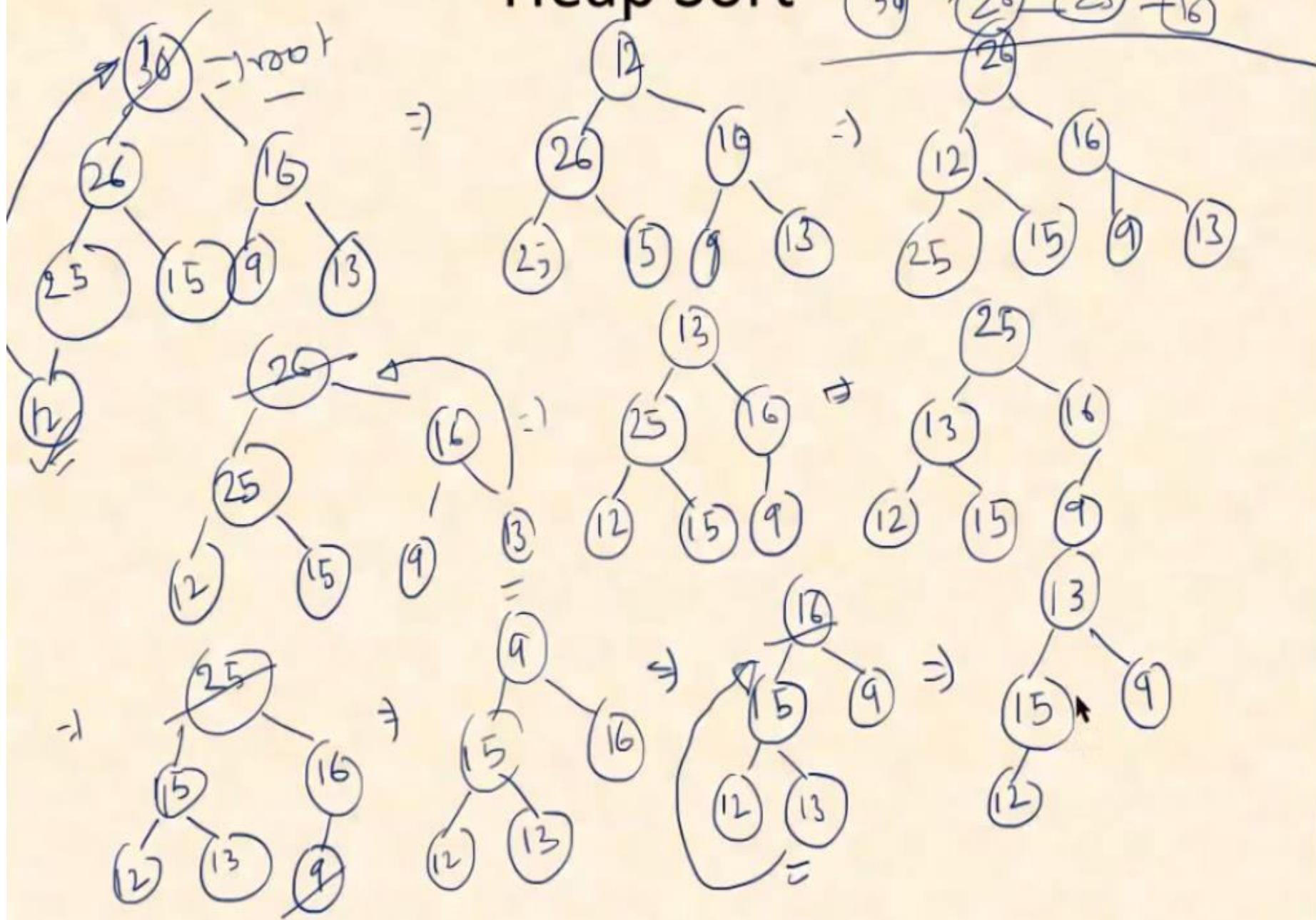
Step 4 : Continue step 2 & 3 until all elements are sorted.

Heap Sort

15, 25, 13, 12, 26, 9, 16, 30

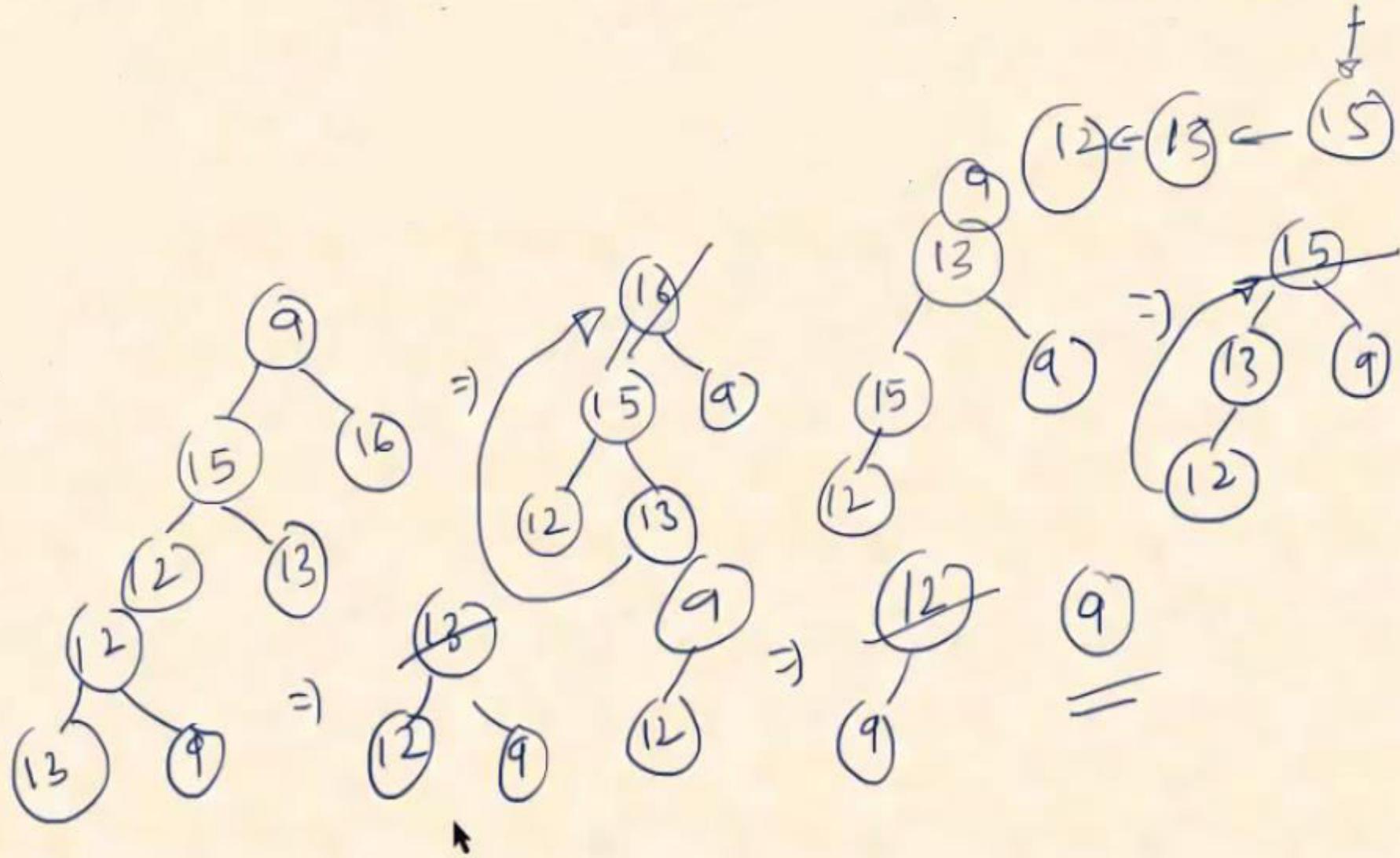


Heap Sort



Heap Sort

30 → 26 - 25 - 16



30 → 26 - 25 - 16 - 15 - 13 - 12 - 9

Heap Sort

Step 1: $O(n \log n)$

Step 2: $O(\log n)$

$$\hookrightarrow \text{total time} = O(\log n + \log n + \log n)$$

$$(\text{adjust}) = O(3 \log n)$$

$$= 3 O(\log n) \approx O(\log n)$$

steps: n times
 $\longrightarrow O(n \log n)$

total time Heap sort: $\text{time} (\text{step 1} + \text{step 2})$
= $O(n \log n)$

Quick sort

Prepare By

Dr. Mohammed Salem Atoum

Quicksort Algorithm

Given an array of n elements (e.g., integers):

- If array only contains one element, return
- Else
 - pick one element to use as *pivot*.
 - Partition elements into two sub-arrays:
 - Elements less than or equal to pivot
 - Elements greater than pivot
 - Quicksort two sub-arrays
 - Return results

Example

We are given array of n integers to sort:

| | | | | | | | | |
|----|----|----|----|----|----|---|----|-----|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|----|----|----|----|----|----|---|----|-----|

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

| | | | | | | | | |
|----|----|----|----|----|----|---|----|-----|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|----|----|----|----|----|----|---|----|-----|

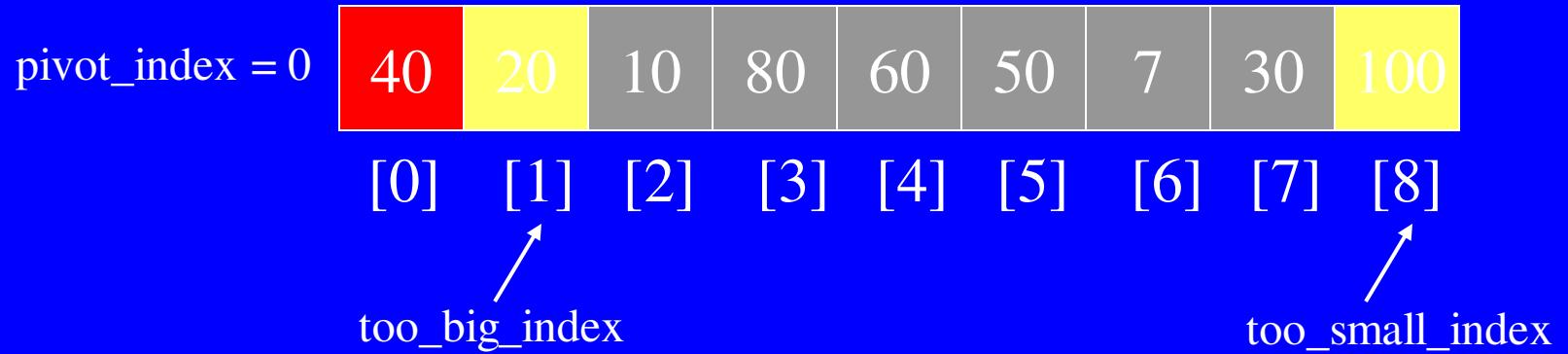
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

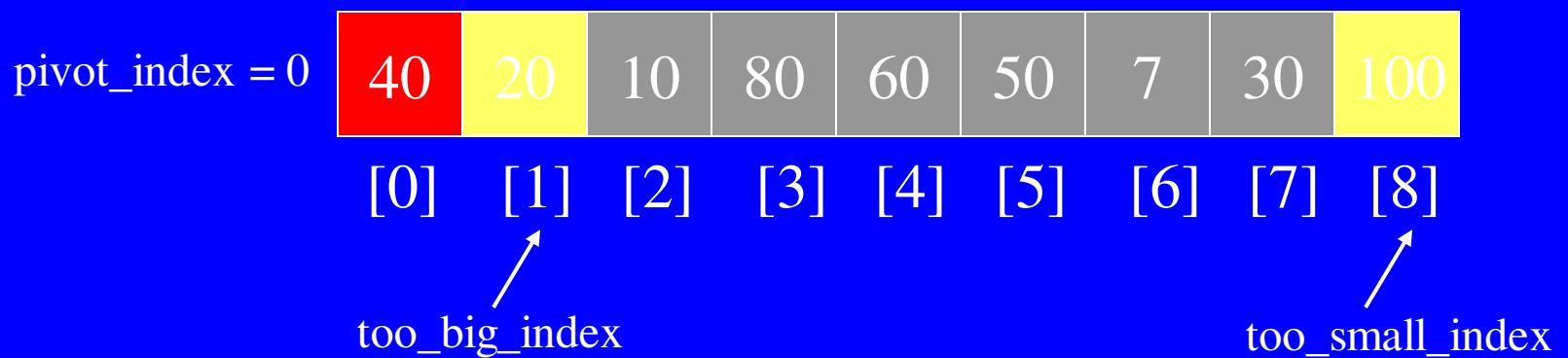
1. One sub-array that contains elements \geq pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

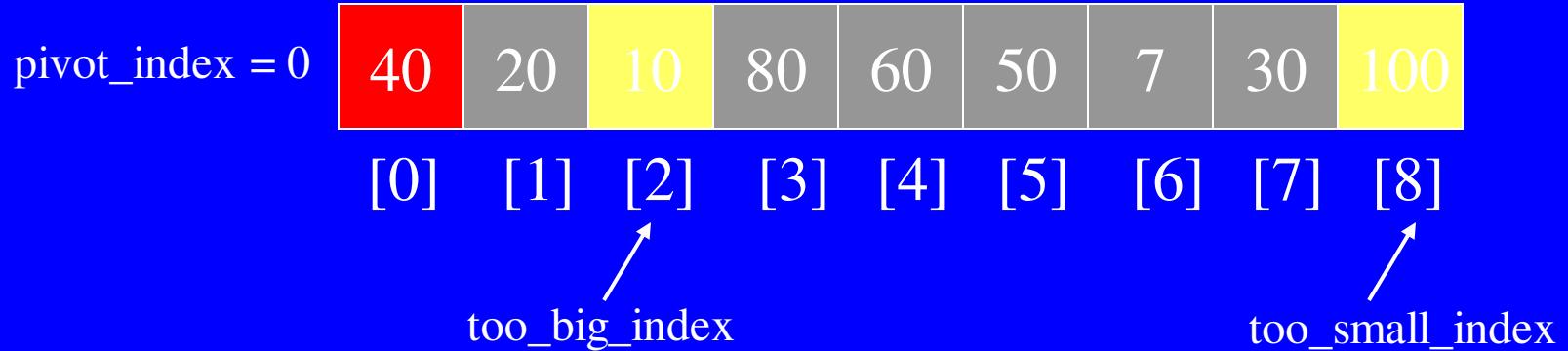
Partitioning loops through, swapping elements below/above pivot.



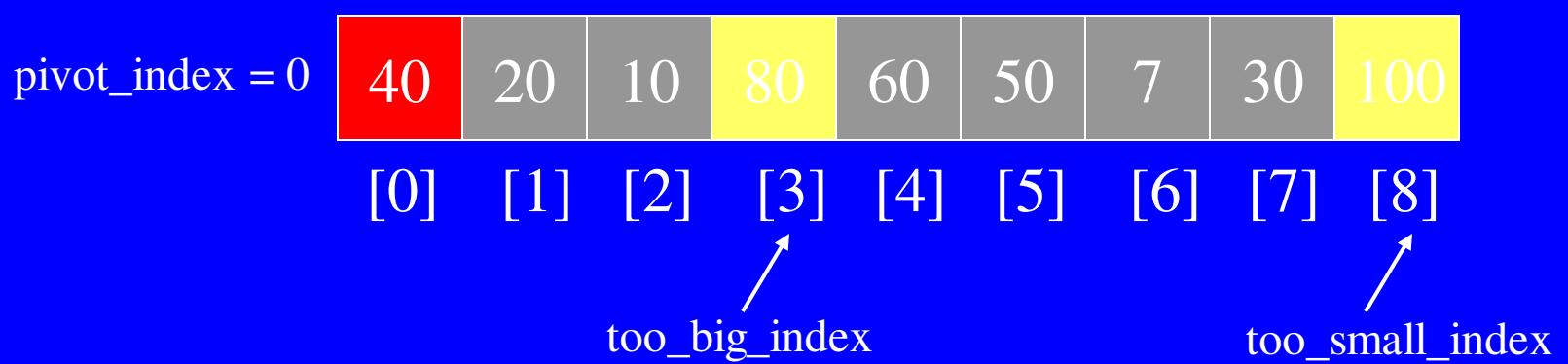
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$



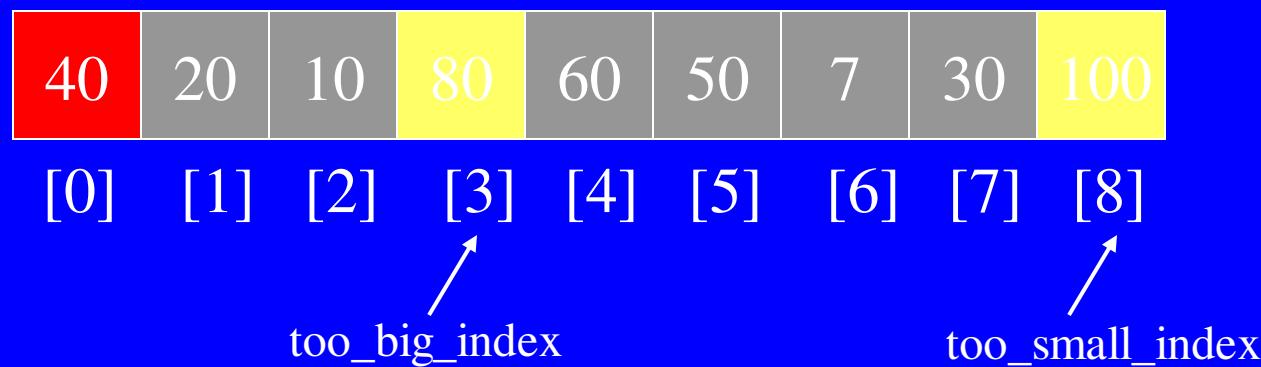
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index



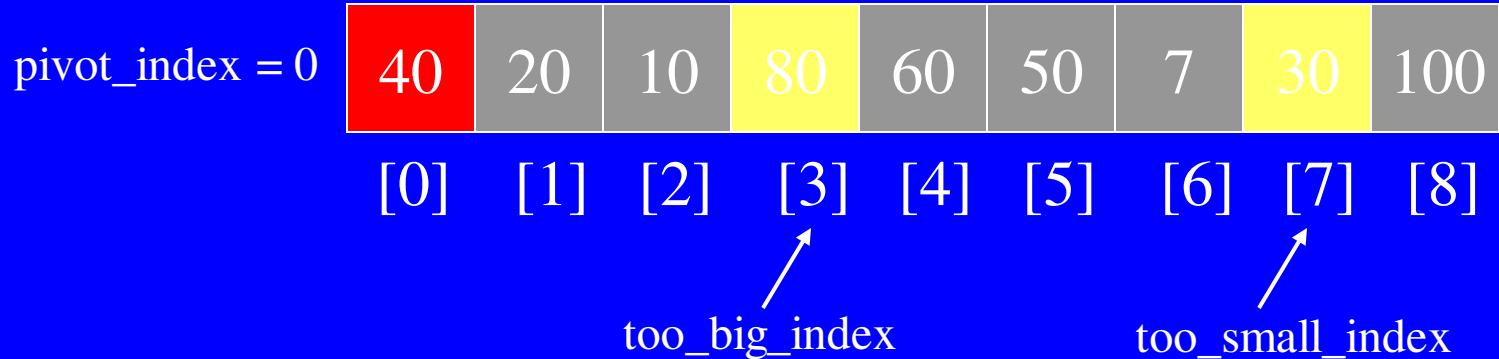
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index



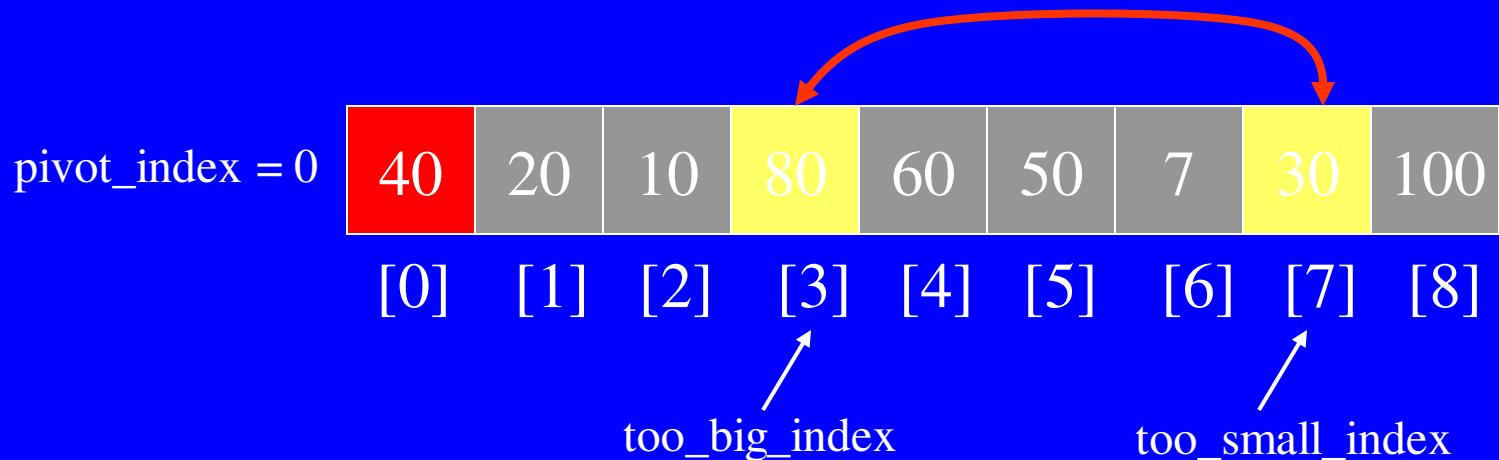
`pivot_index = 0`



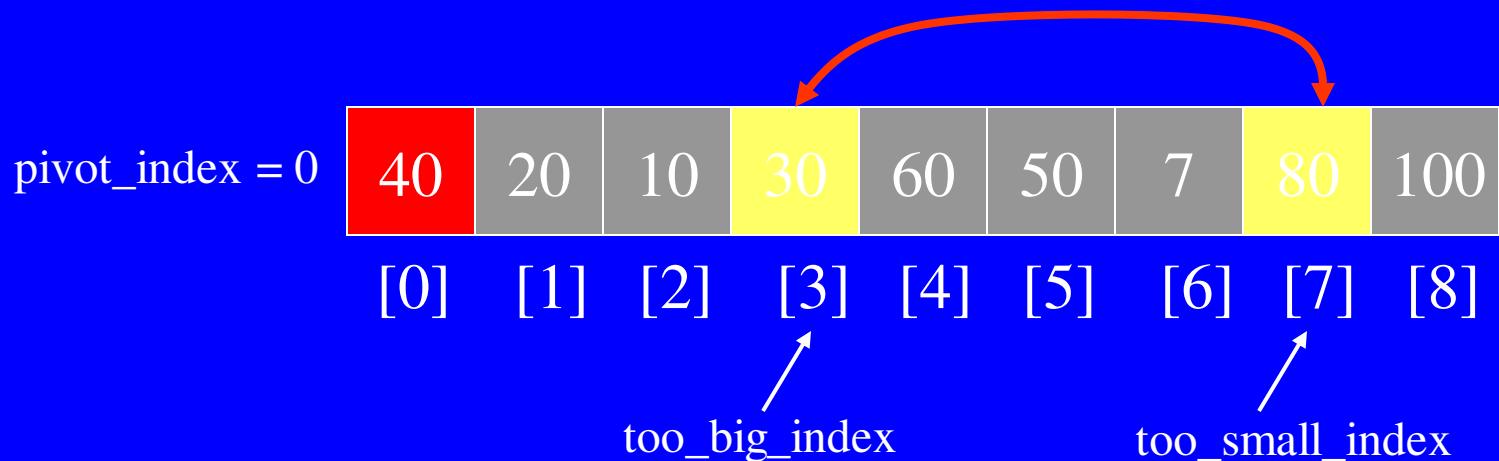
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index



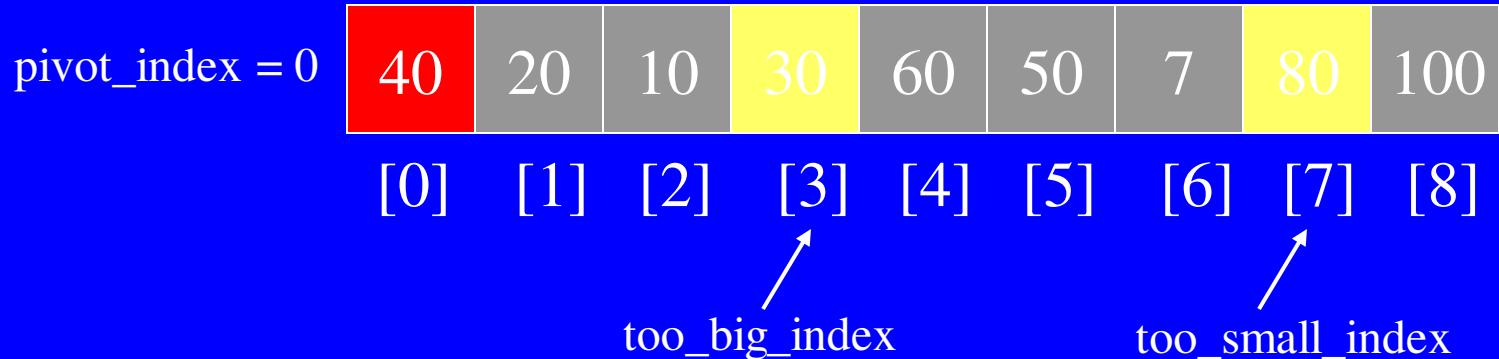
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$



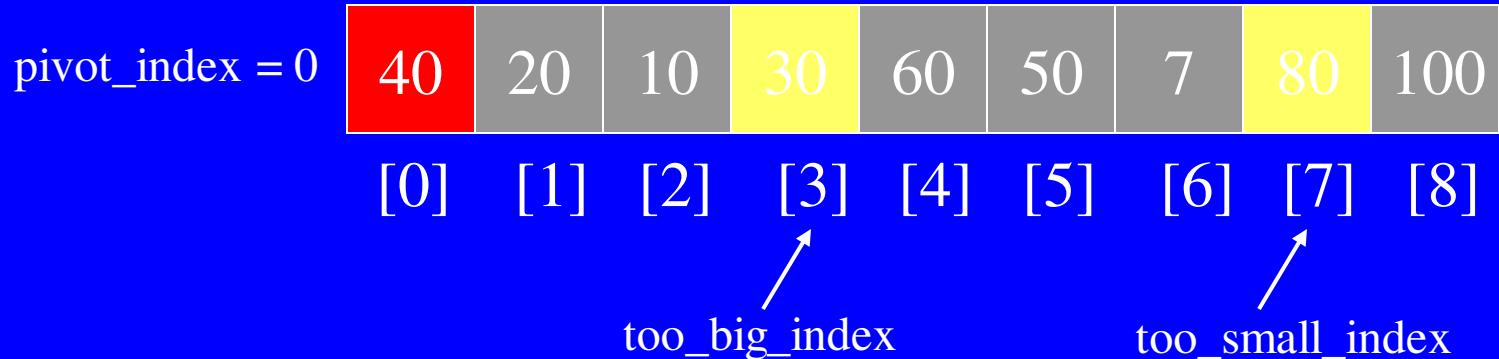
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$



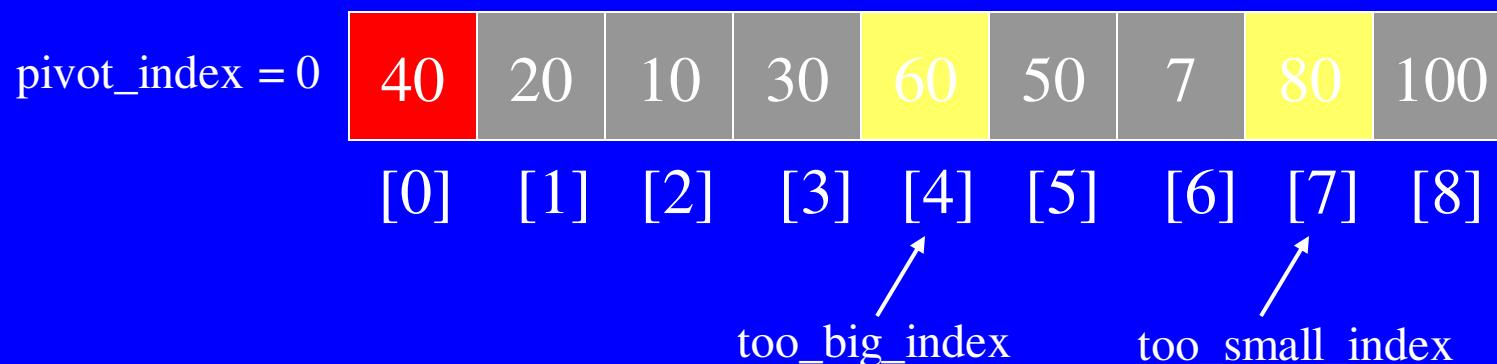
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



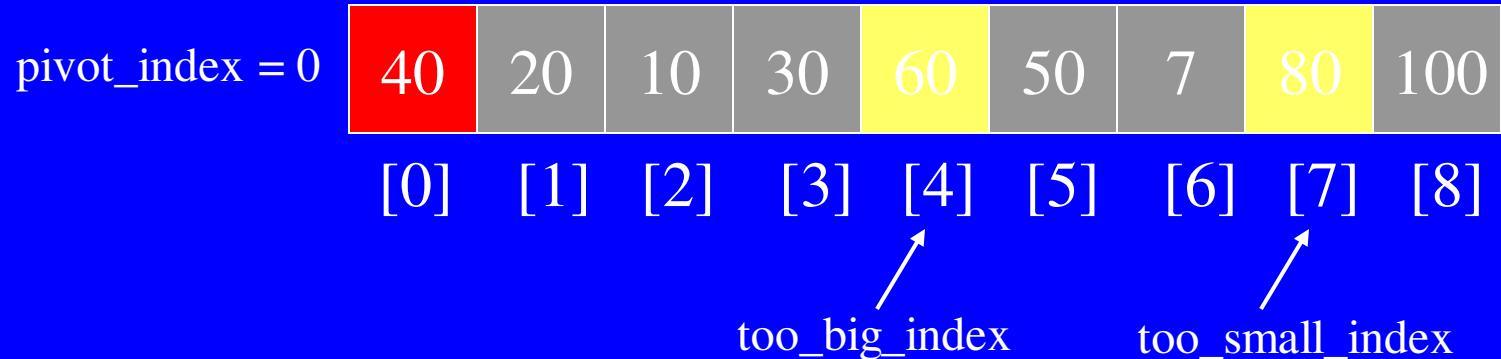
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



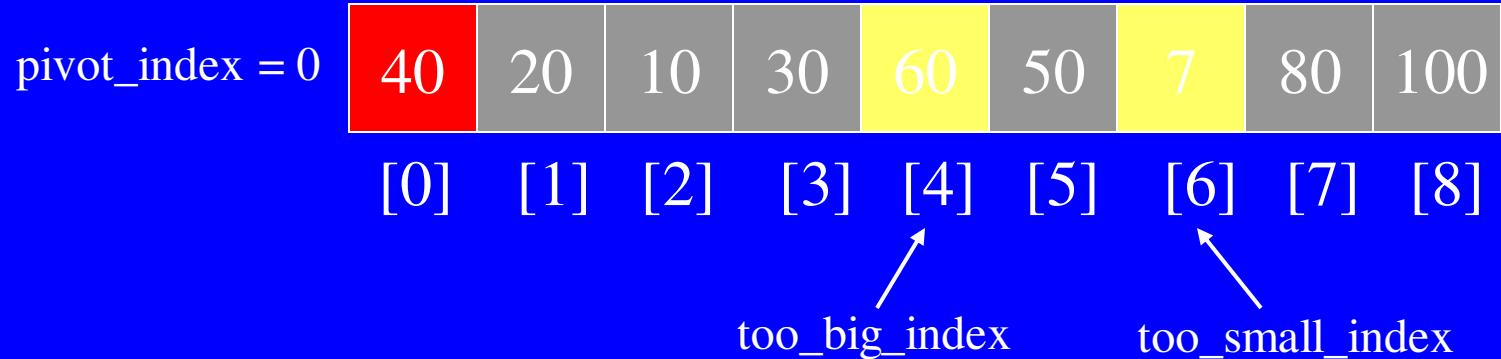
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



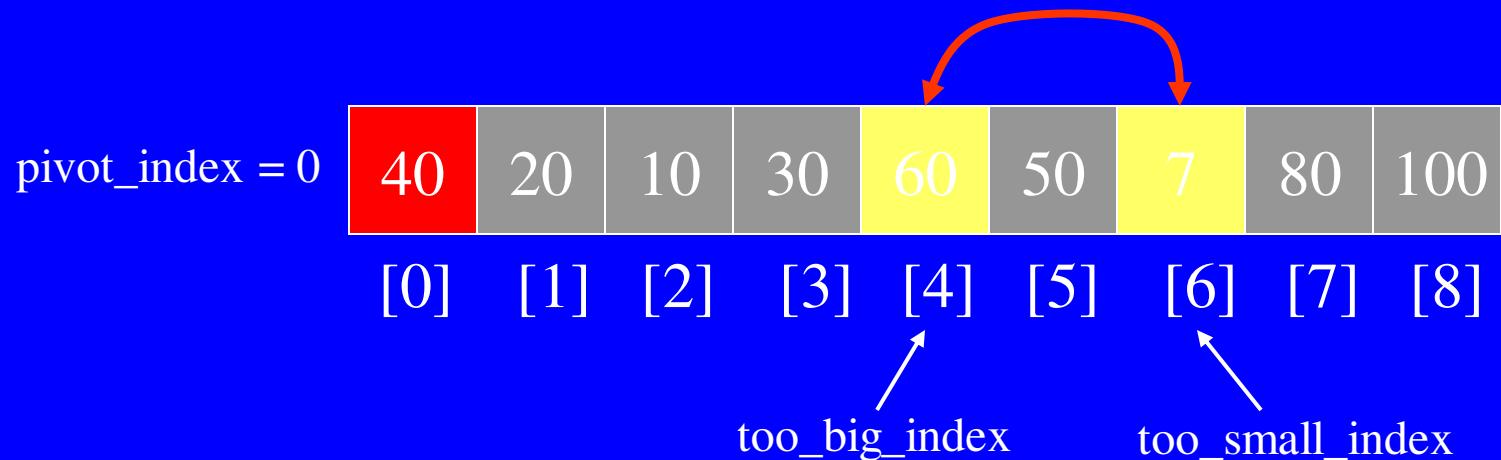
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



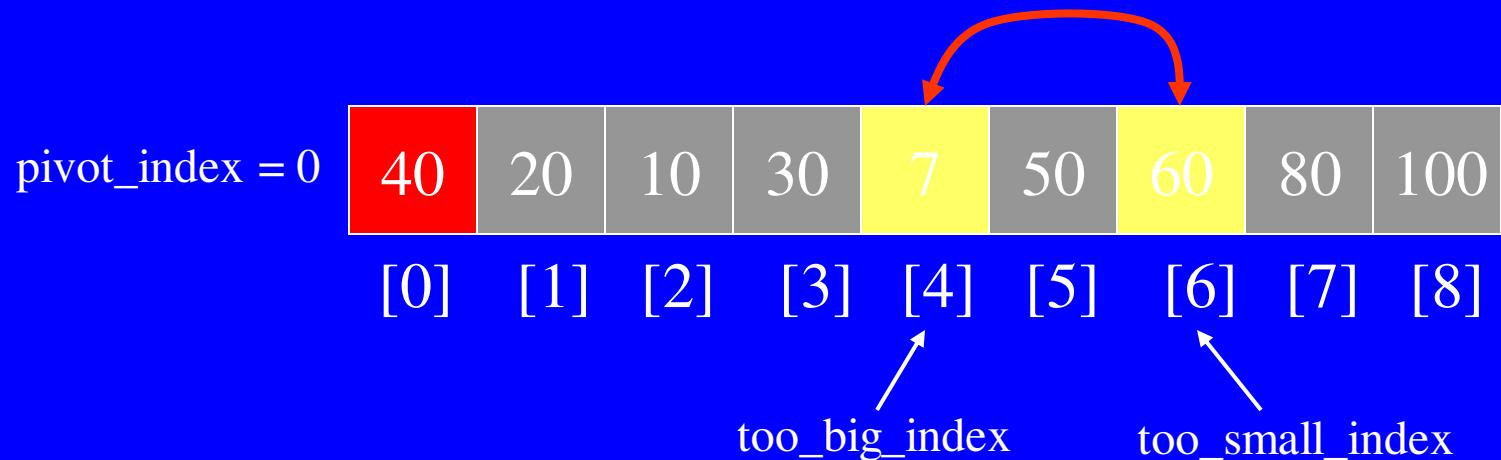
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



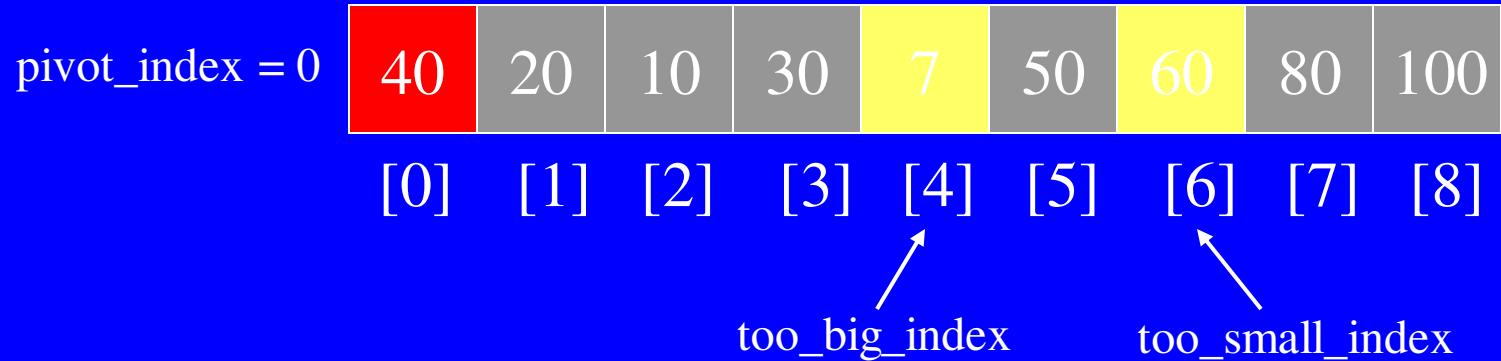
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



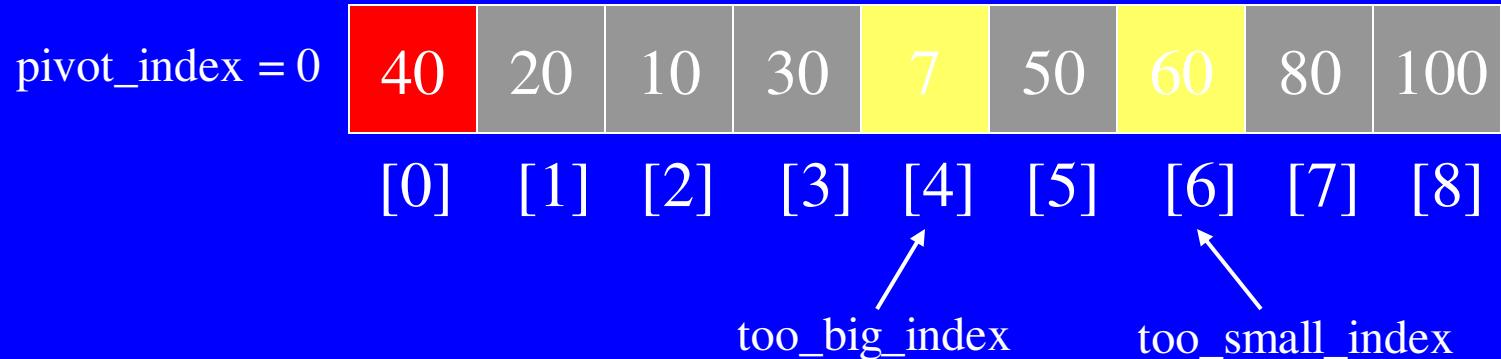
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



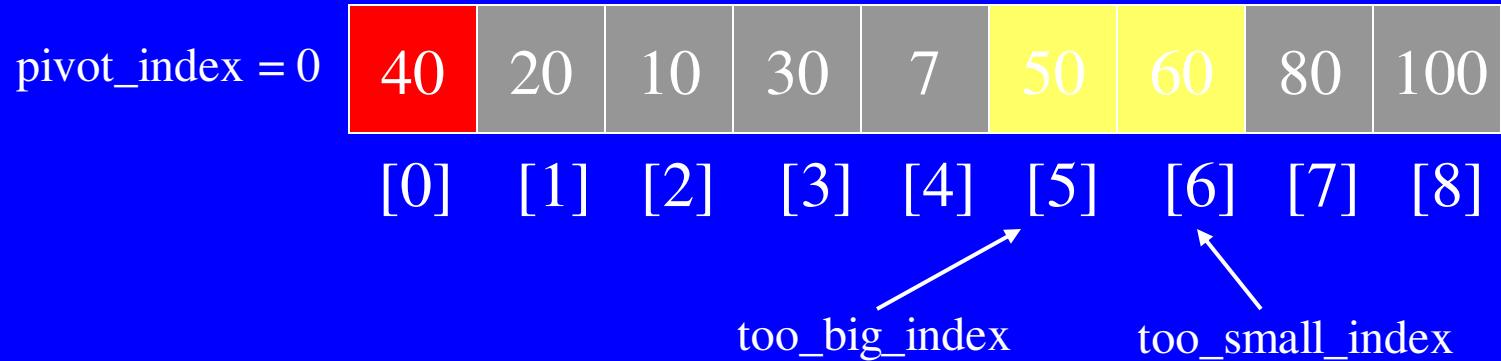
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



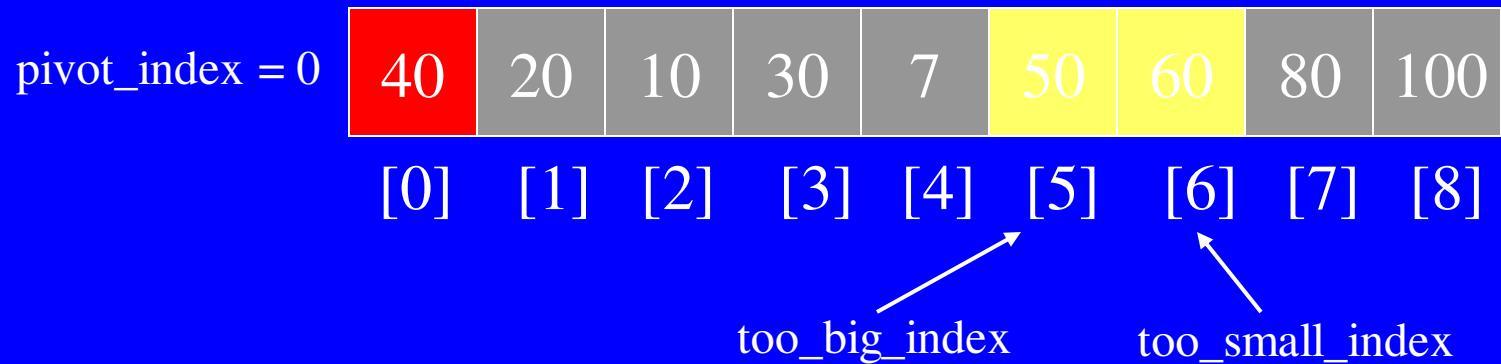
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad\quad\quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad\quad\quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad\quad\quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



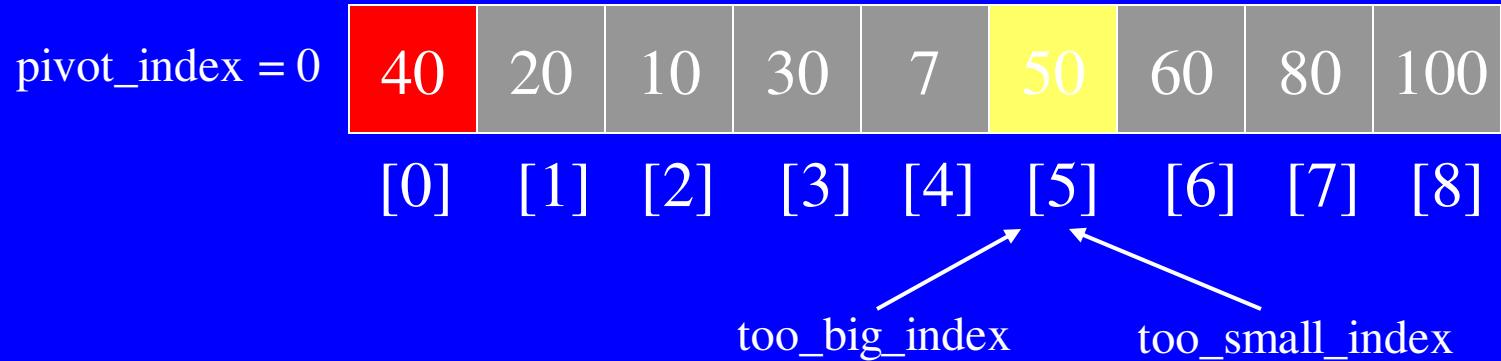
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



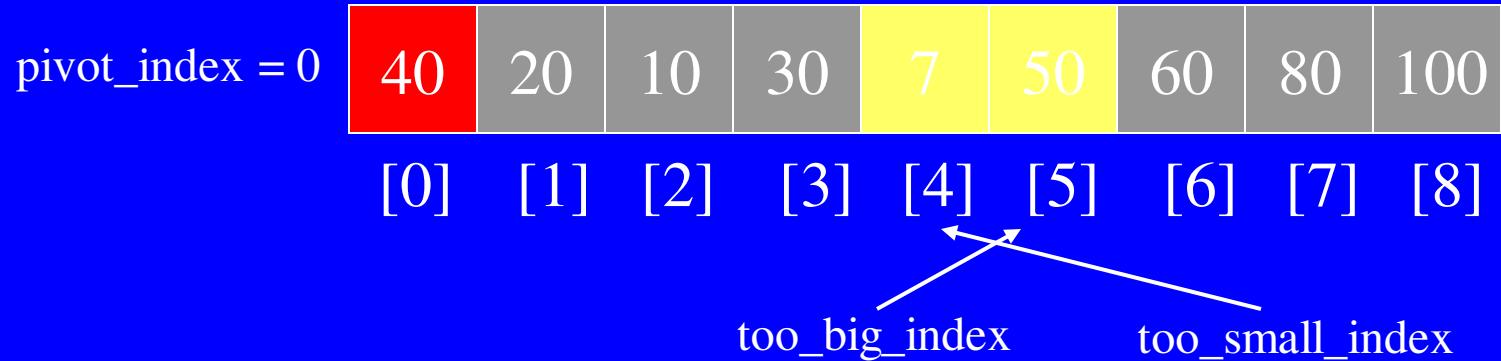
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



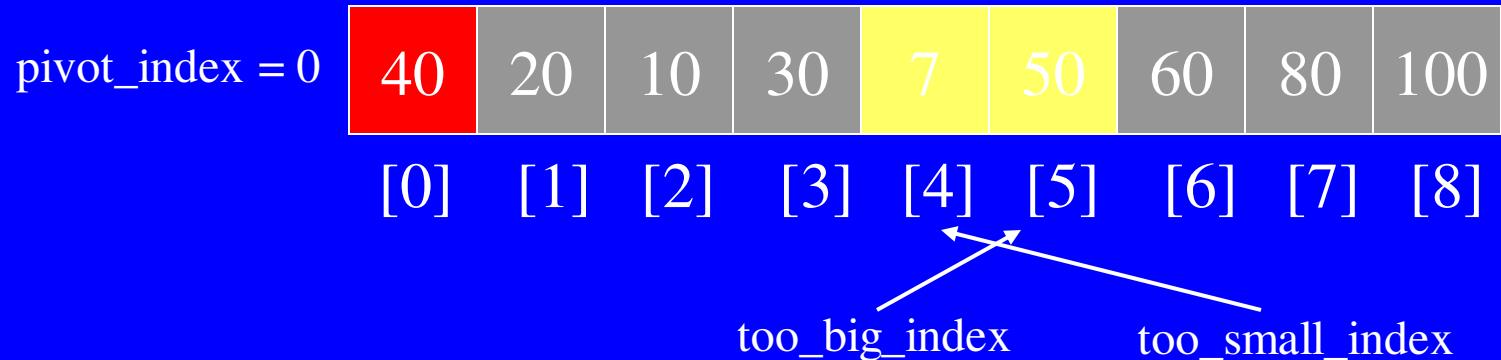
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



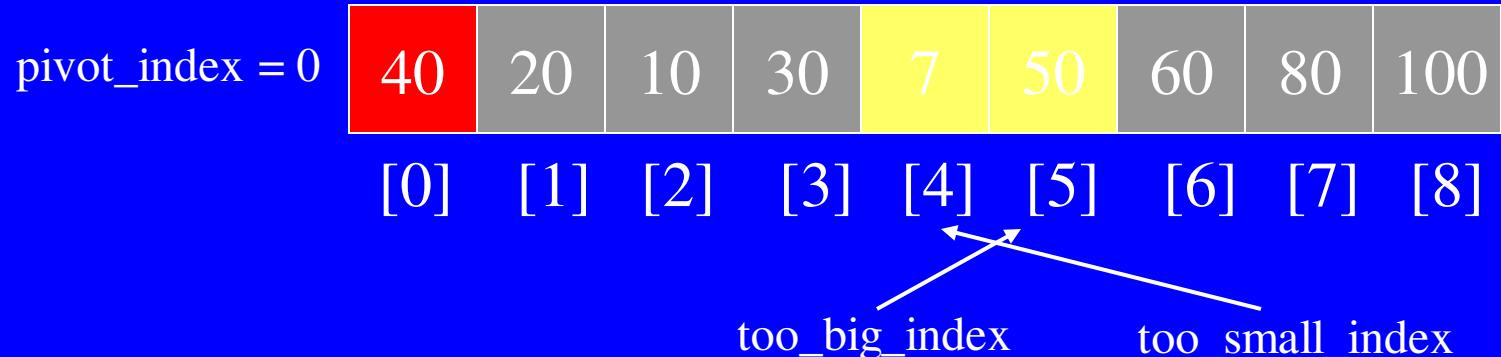
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



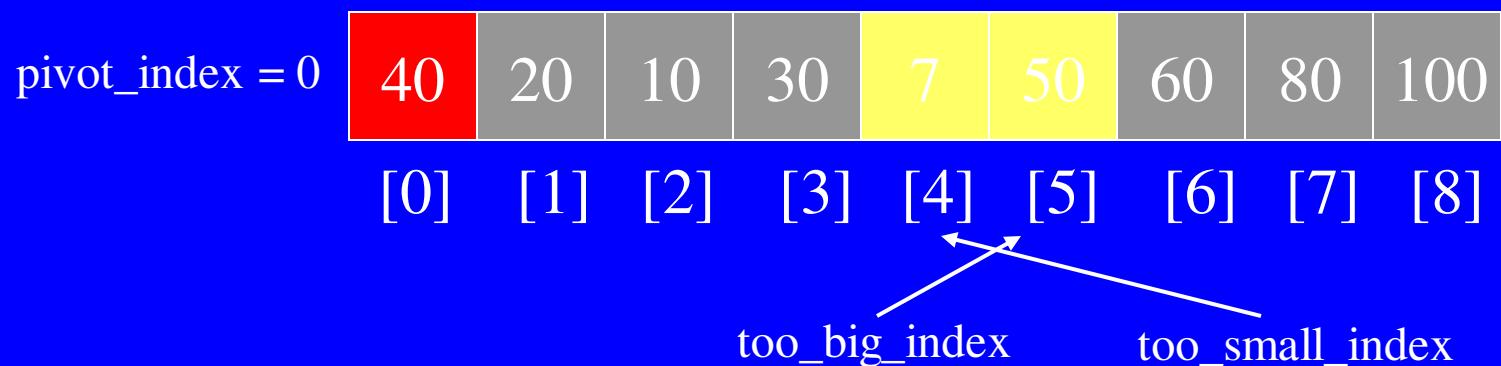
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



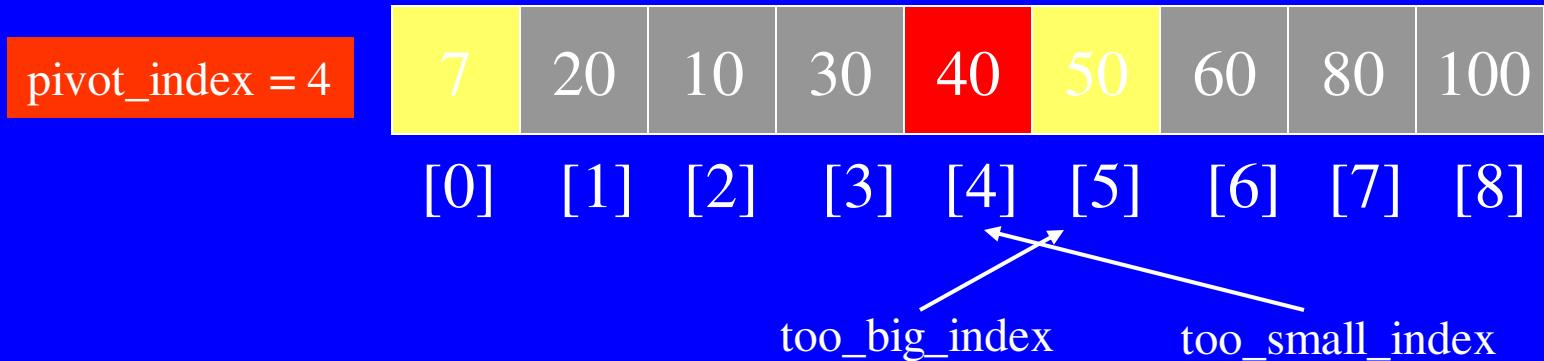
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



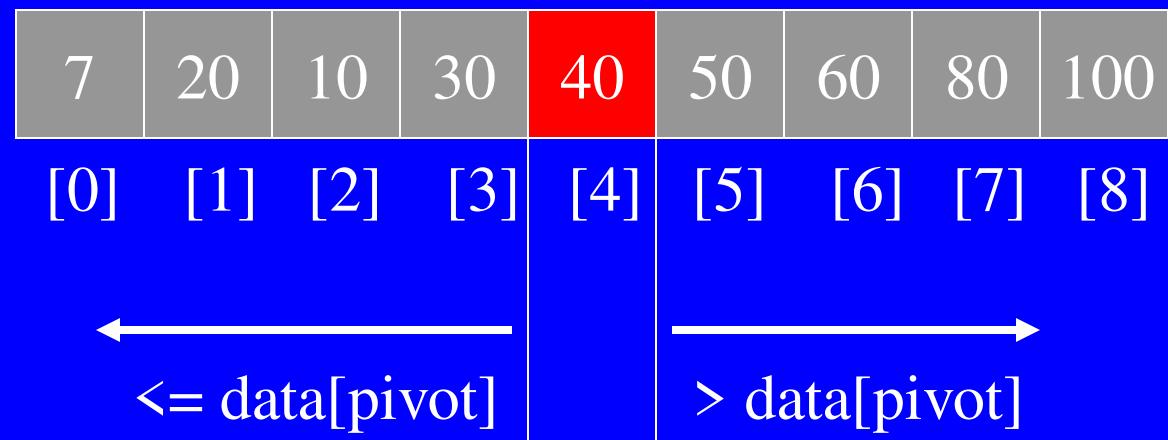
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
- 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



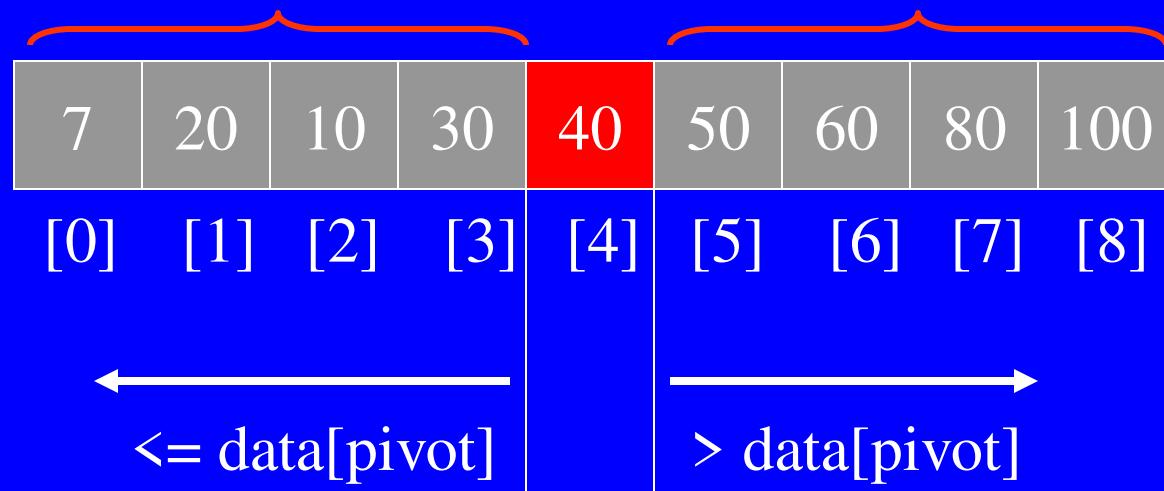
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
- 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



Partition Result



Recursion: Quicksort Sub-arrays



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$
 - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$
 - Number of accesses in partition? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

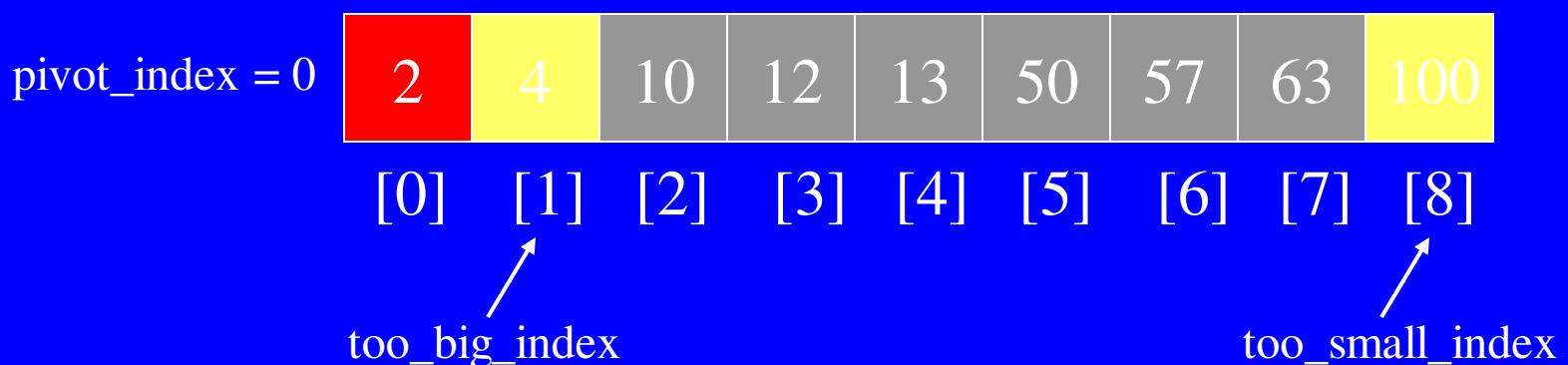
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Quicksort: Worst Case

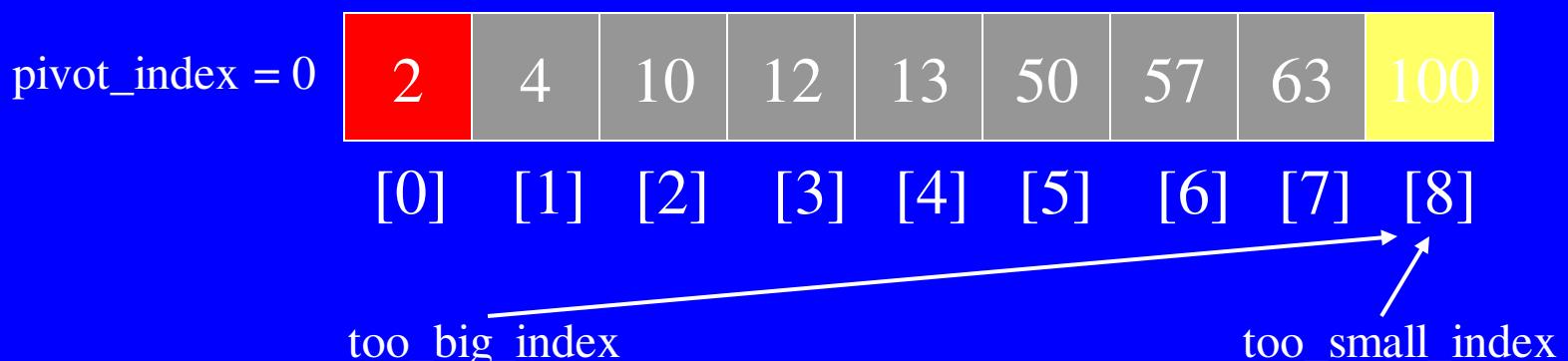
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



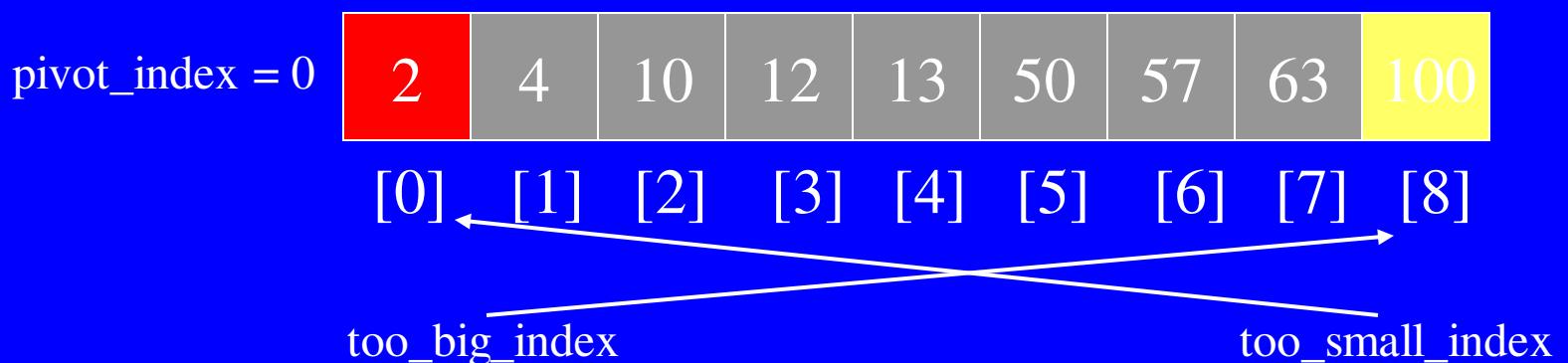
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



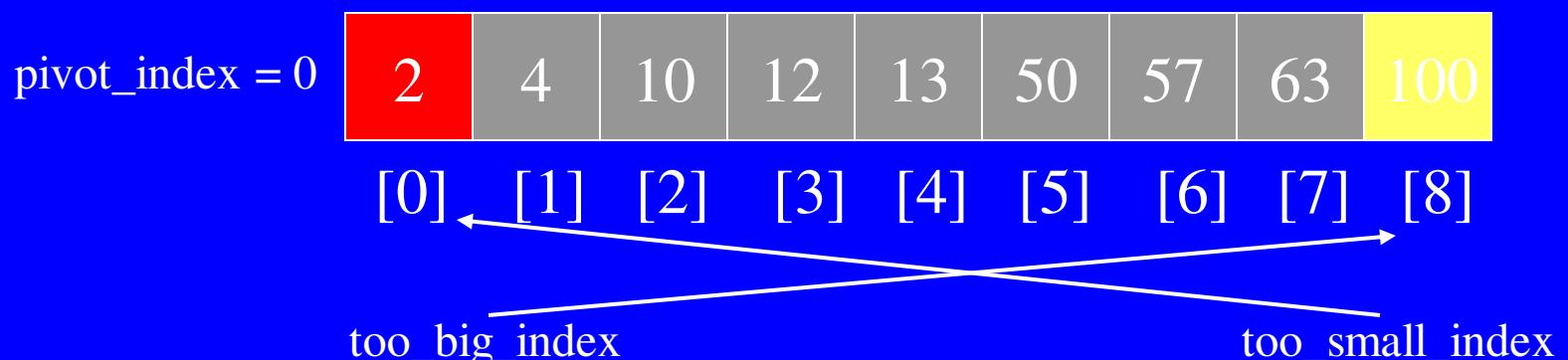
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad\quad\quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad\quad\quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



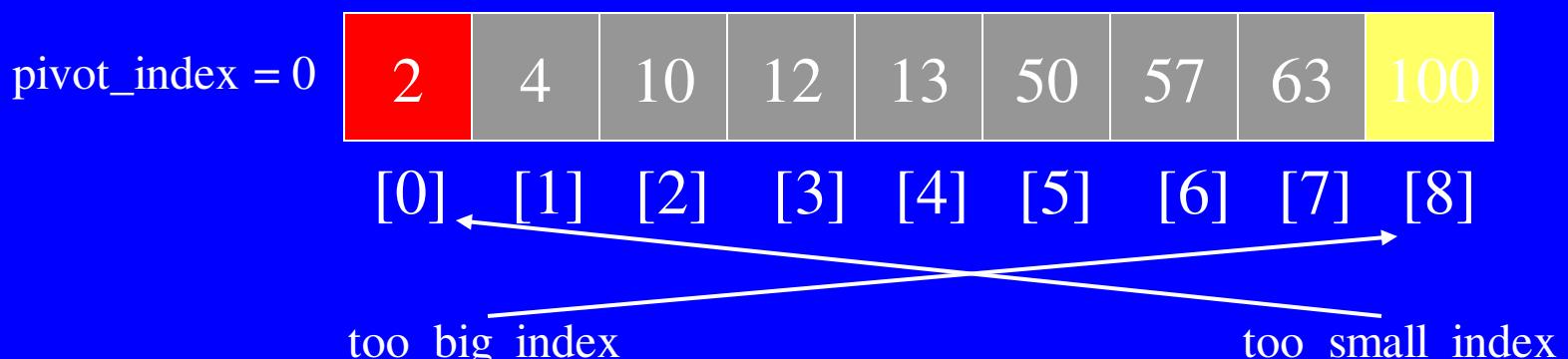
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



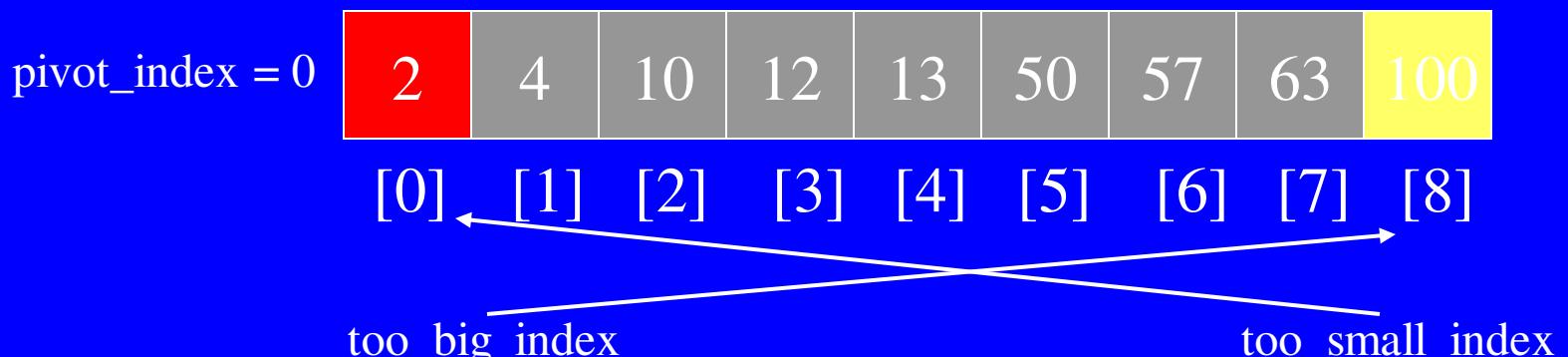
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad\quad\quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad\quad\quad \text{--too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad\quad\quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad\quad\quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$

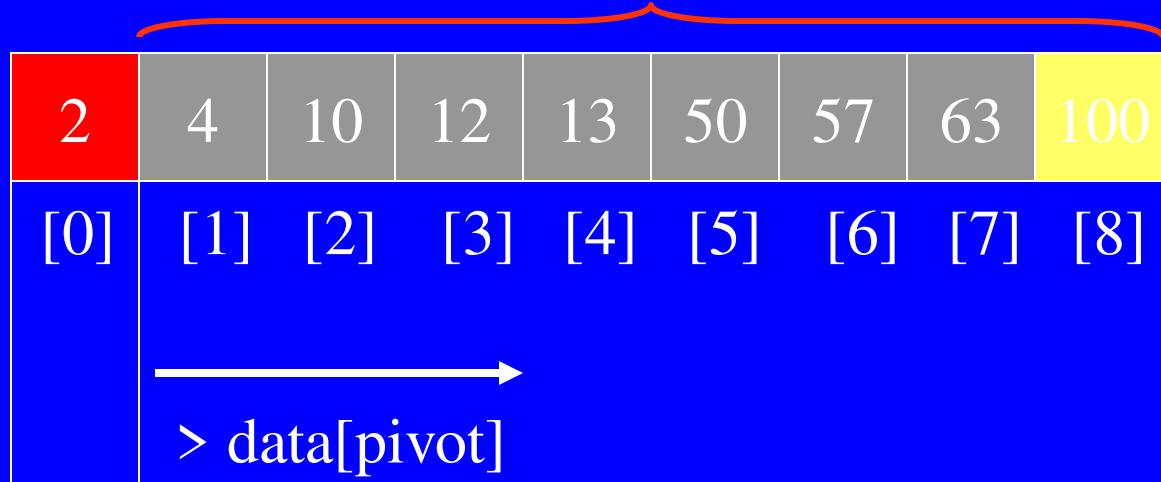


1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
- 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
- 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$

pivot_index = 0



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)!!!$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)!!!$
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:
 $\text{data}[0]$, $\text{data}[n/2]$, and $\text{data}[n-1]$.

Use this median value as pivot.

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
 - Sub-array of size 1: trivial
 - Sub-array of size 2:
 - if($\text{data[first]} > \text{data[second]}$) swap them
 - Sub-array of size 3: left as an exercise.

Algorithm Design & Analysis

Chapter 4: Sort Algorithms

Heap, Binomial Heap, Quick, Counting

Dr. Mohammed Salem Atoum

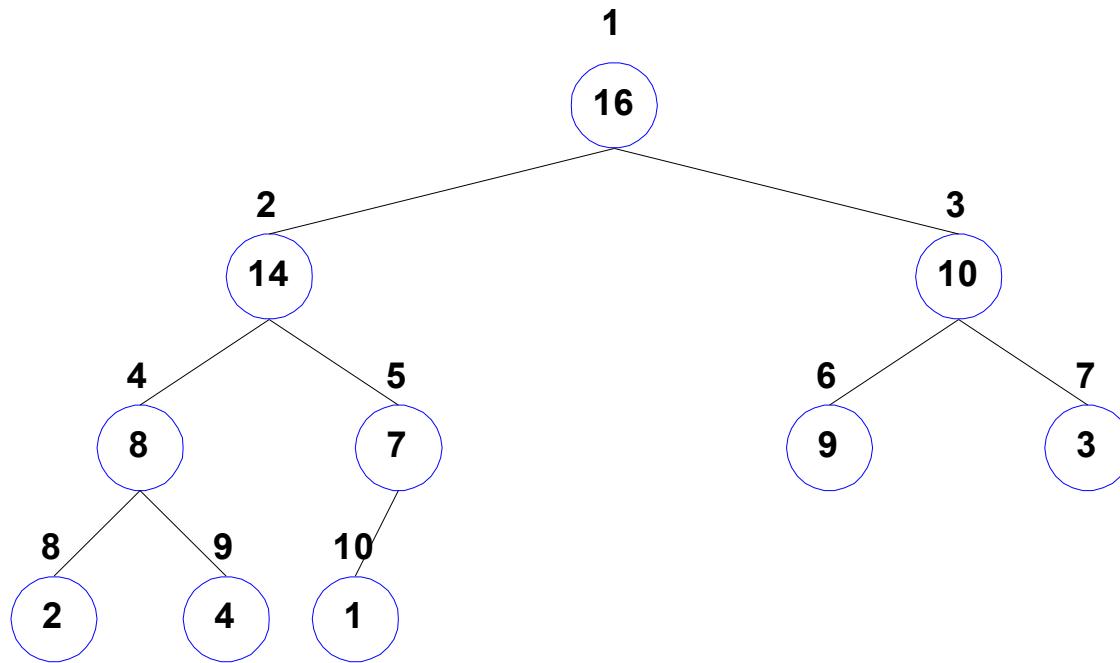
M.Atoum@inu.edu.jo

Heapsort

We introduce another sorting algorithm. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are sorted outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms.

Heapsort also introduces another algorithm design technique: the use of a data structure, in this case one we call a “**heap**” to manage information during the execution of the algorithm. Not only is the heap data structure useful for heapsort, it also makes an efficient priority queue.

Heaps



a-Heap view as a
binary tree

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|----|----|---|---|---|---|---|---|----|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

b-Heap view as an
array

Binary Heap Data Structure

Parent(i) return $\lfloor i/2 \rfloor$

Left(i) return $2i$

Right(i) return $2i + 1$

On most computer, the **Left** procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left one bit position.

Similarly, the **Right** procedure can quickly compute $2i+1$ by shifting the binary representation of i left one bit position and shifting in a 1 as the low-order bit. The **Parent** procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position.

Heaps also satisfy the **heap property**: for every node i other than the root, $A[\text{Parent}(i)] \geq A[i]$, the value of a node is at most the value of its parent. Thus, the **largest** element in a heap is stored at the **root**, and the subtrees rooted at a node contain smaller value than does the node itself. We define the height of a node in a tree to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the tree to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$.

Maintaining the Heap Property

Heapify is an important subroutine for manipulating heaps. Its input are an array A and an index i into the array. When Heapify is called, it is assumed that the binary trees rooted at $\text{Left}(i)$ and $\text{Right}(i)$ are heaps, but that $A[i]$ may be smaller than its children. The function of Heapify is to let the value at $A[i]$ “float down” in the heap so that the subtree rooted at index I becomes a heap.

Heapify(A,i)

$L \leftarrow \text{Left}(I)$

$R \leftarrow \text{Right}(I)$

If $L \leq \text{heap-size}[A]$ and $A[L] > A[I]$

then largest $\leftarrow L$

else largest $\leftarrow I$

If $R \leq \text{heap-size}[A]$ and $A[R] > A[\text{largest}]$

then largest $\leftarrow R$

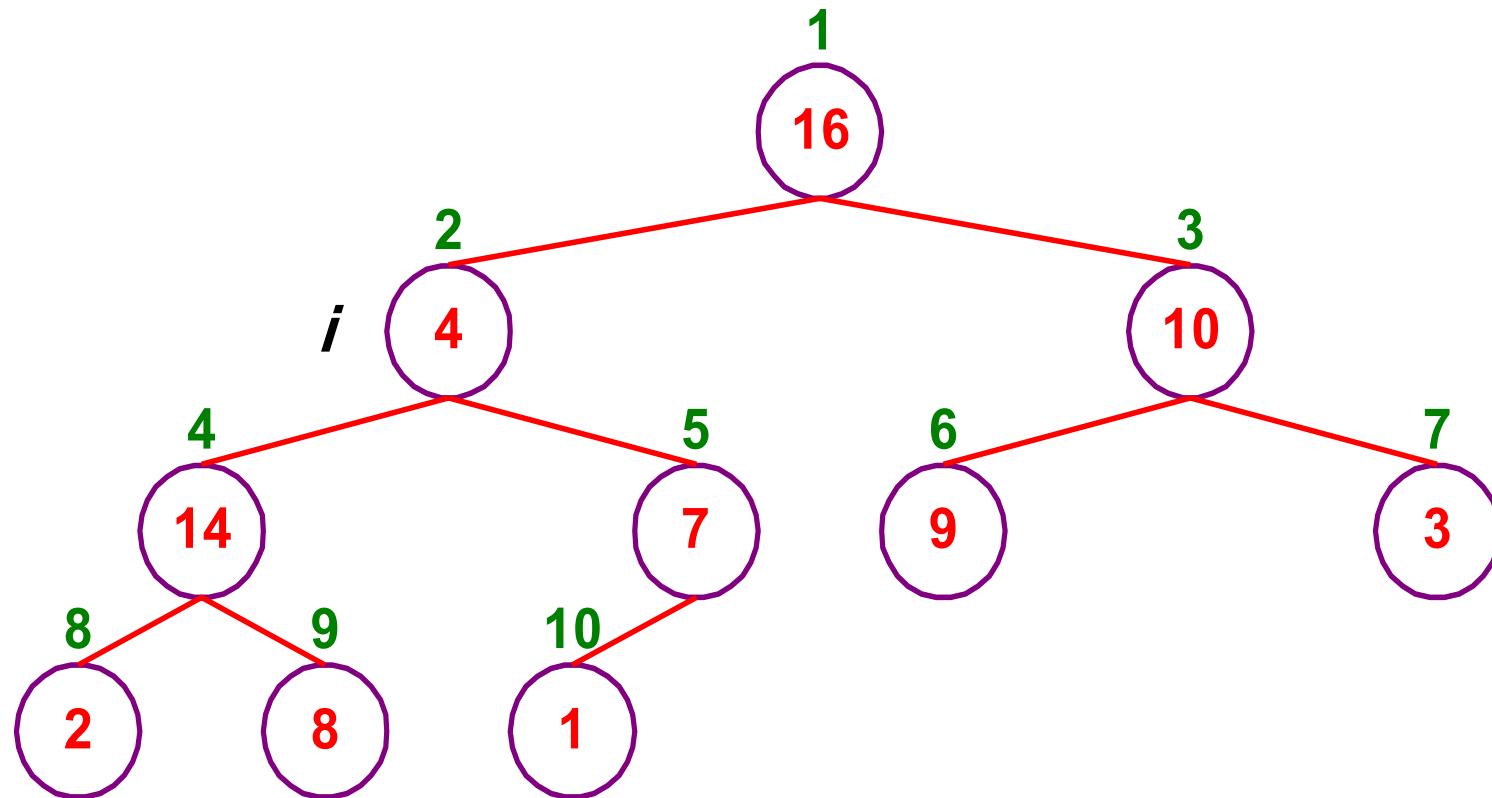
If largest $\neq I$

then exchange $A[I] \longleftrightarrow A[\text{largest}]$

Heapify($A,\text{Largest}$)

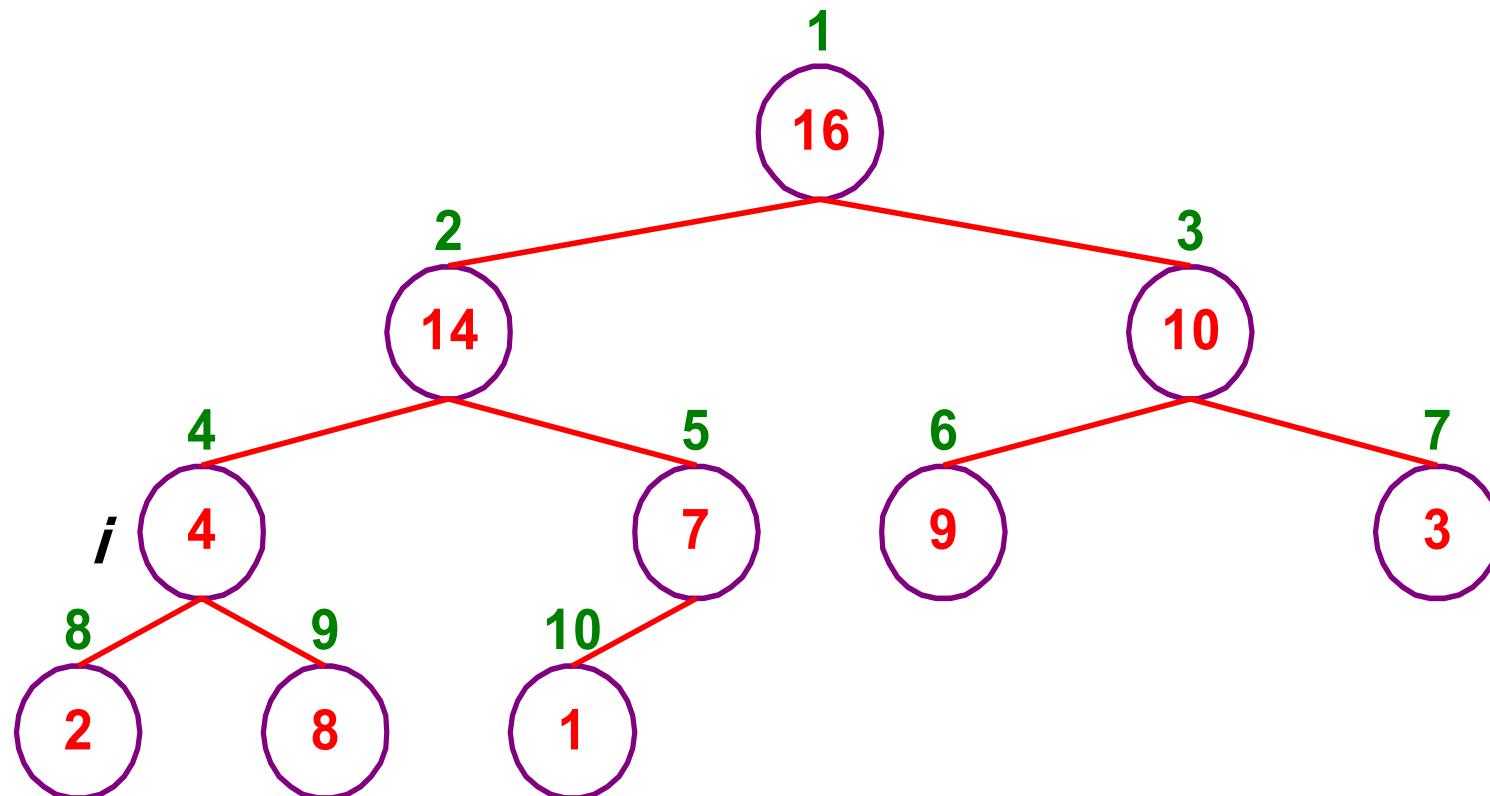
Action of Heapify

1-The action of **Heapify(A,2)**, where **heap-size[A]=10**, the initial configuration of the heap, with **A[2]** at node **I=2** violating the heap property since it is not larger than both children. The heap property is restored for node **2** .



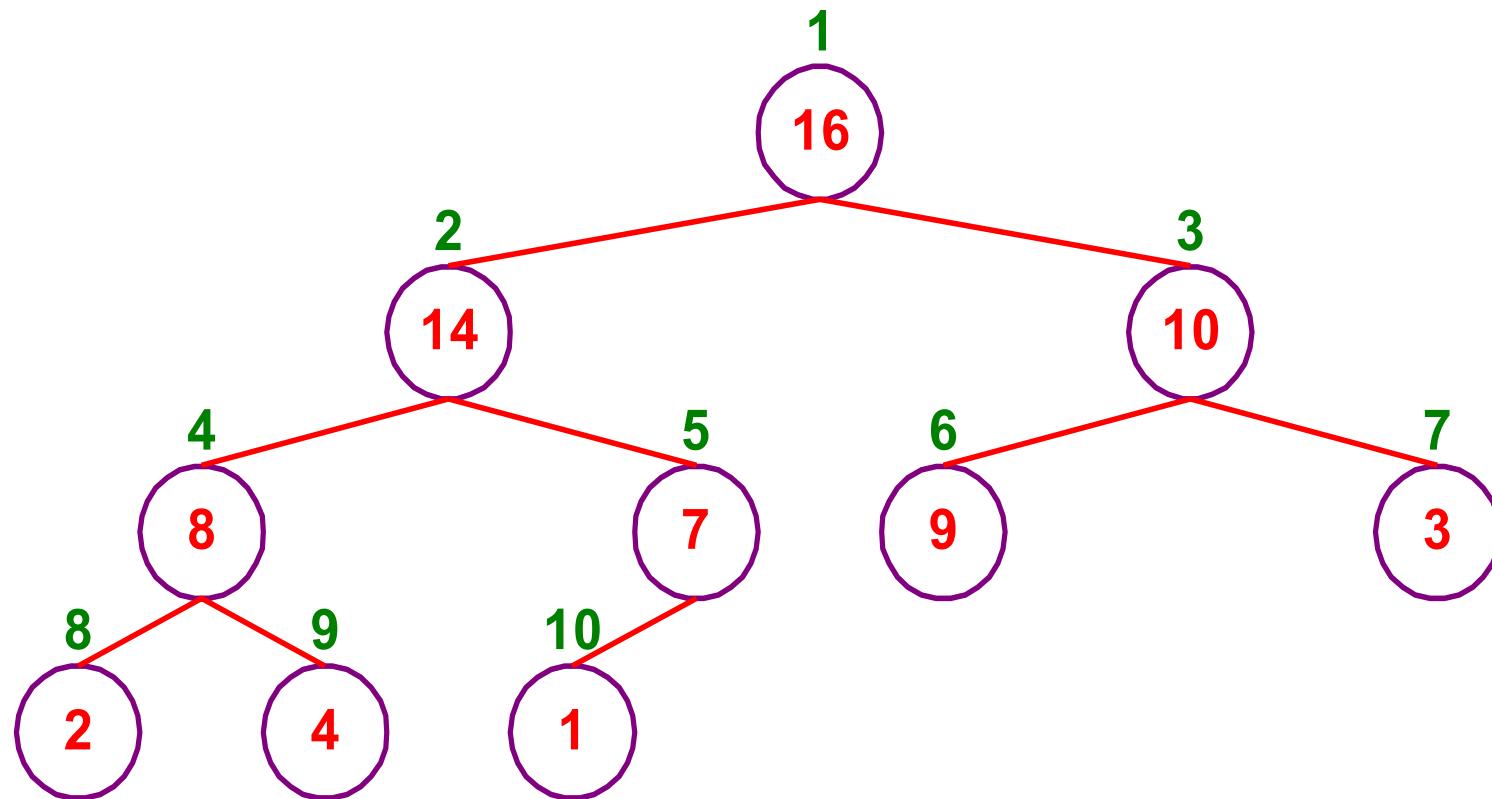
Action of Heapify

2- by exchanging A[2] with A[4], which destroys the heap property for node 4. The recursive call **Heapify(A,4)** now sets I=4.



Action of Heapify

3-After swapping $A[4]$ with $A[9]$, node 4 is fixed up, and the recursive call $\text{Heapify}(A,9)$ yields no further change to the data structure.



The Running Time of Heapify

The running time of Heapify on a subtree of **size n** rooted at given **node I** is the $\Theta(1)$ time to fix up the relationships among the elements $A[I], A[Left(I)],$ and $A[Right(I)]$, plus the time to run *Heapify on a subtree rooted at one of the children of node I .*

The children's subtrees each have size at most **$2n/3$** , the worst case occurs when the last row of the tree is exactly half full, and the running time of Heapify can therefore be described by the recurrence :

$$T(n) \leq T(2n/3) + \Theta(1)$$

Building a Heap

We can use the procedure **Heapify** in a bottom-up manner to convert an array $A[1..n]$, where $n = \text{length}[A]$, into a heap. Since the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)..n]$ are all leaves of the tree, each is a **1-element** heap to begin with. The procedure **Build-Heap** goes through the remaining nodes of the tree and runs **Heapify** on each one. The order in which the nodes are processed guarantees that the subtrees rooted at children of a node I are heaps before **Heapify** is run at that node.

Build-Heap(A)

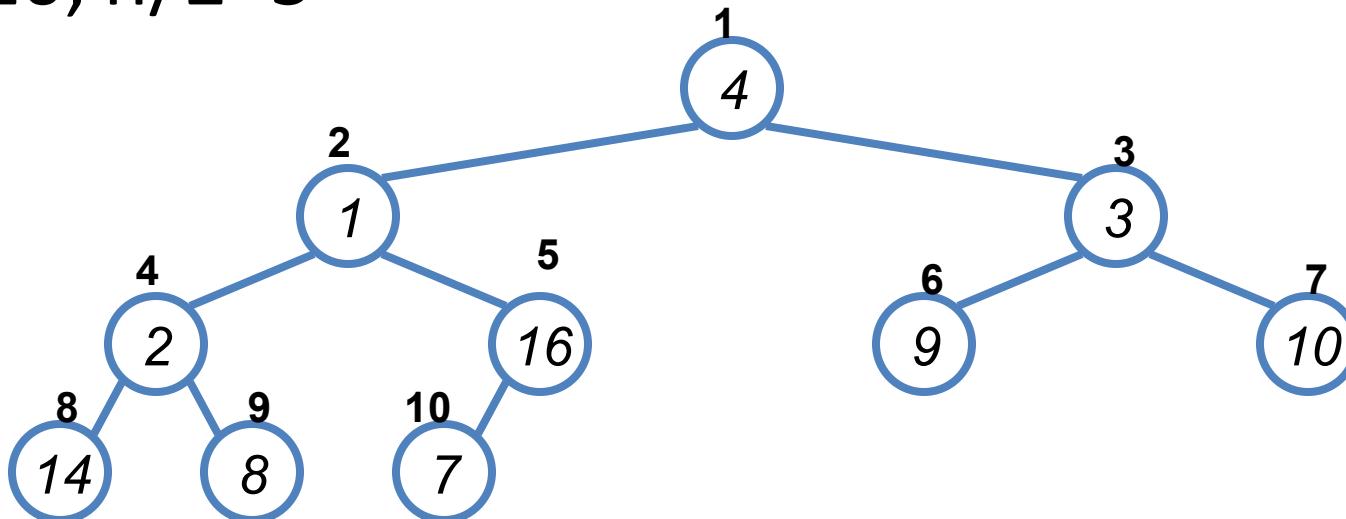
- 1 $\text{heap-size}[A] \leftarrow \text{length}[A]$
- 2 *for* $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ *downto* 1
- 3 do **Heapify**(A, I)

BuildHeap(Example)

- Work through example

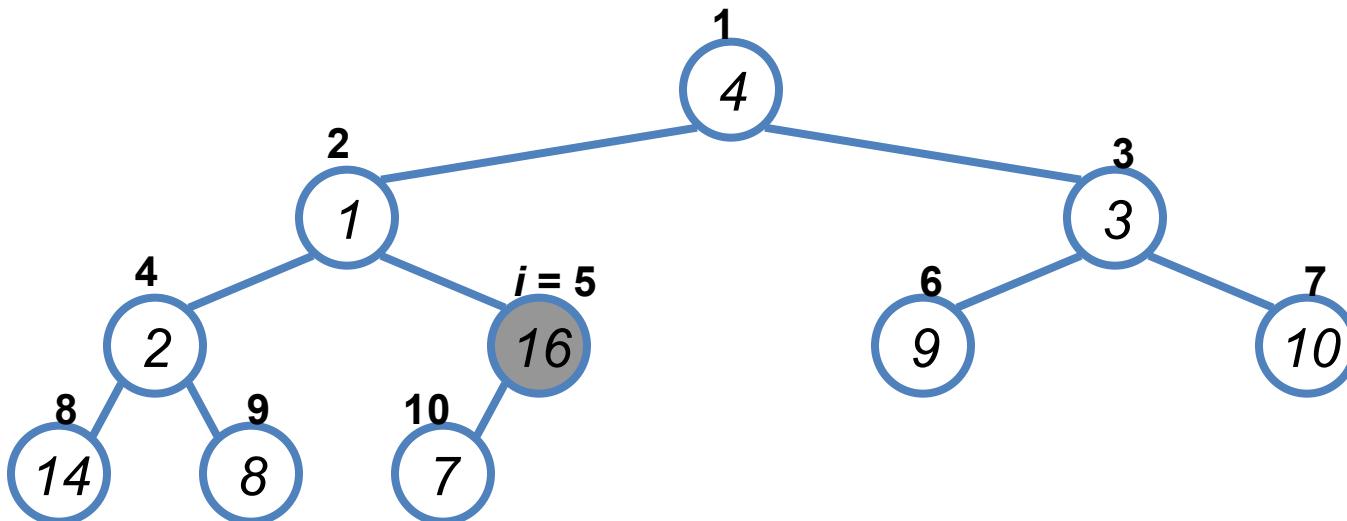
$$A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$$

$$n=10, n/2=5$$
 •



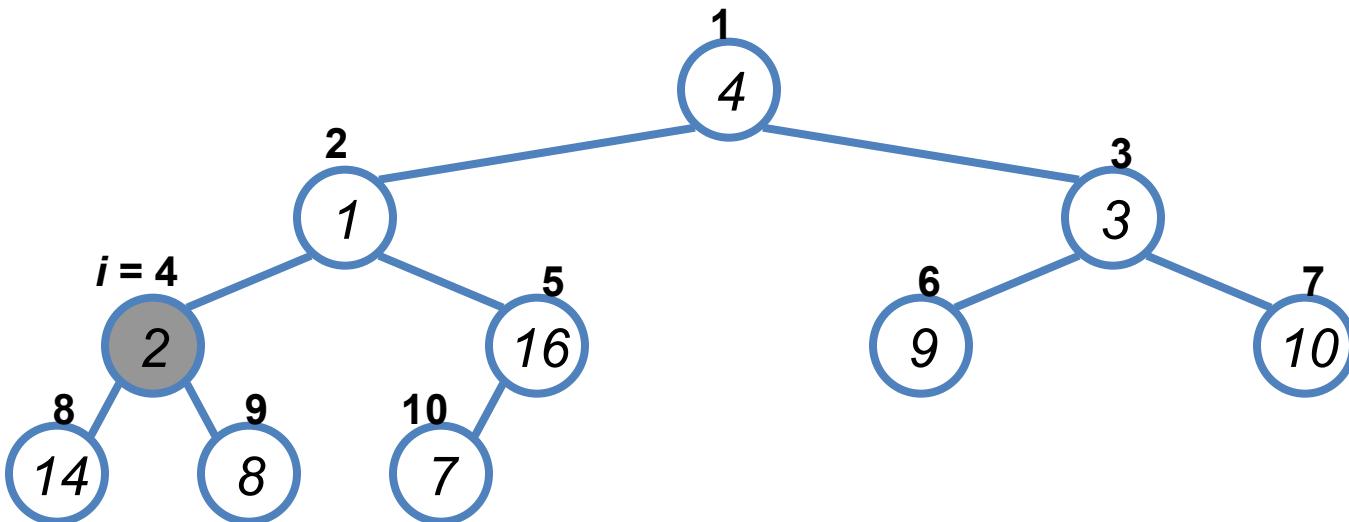
BuildHeap() Example

- $A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$



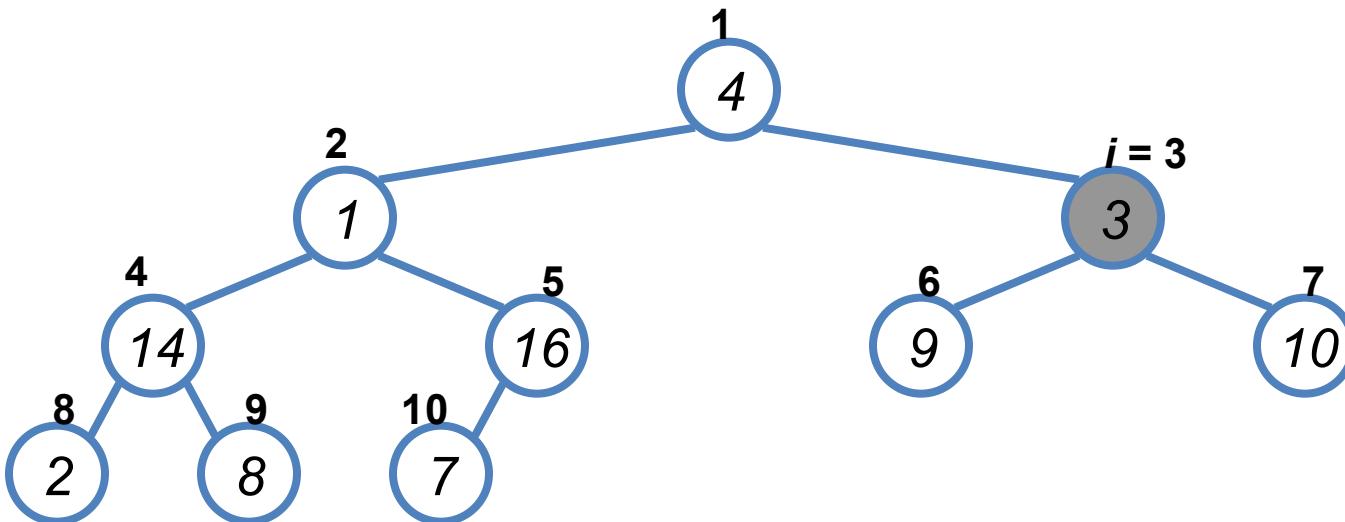
BuildHeap() Example

$A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$ •



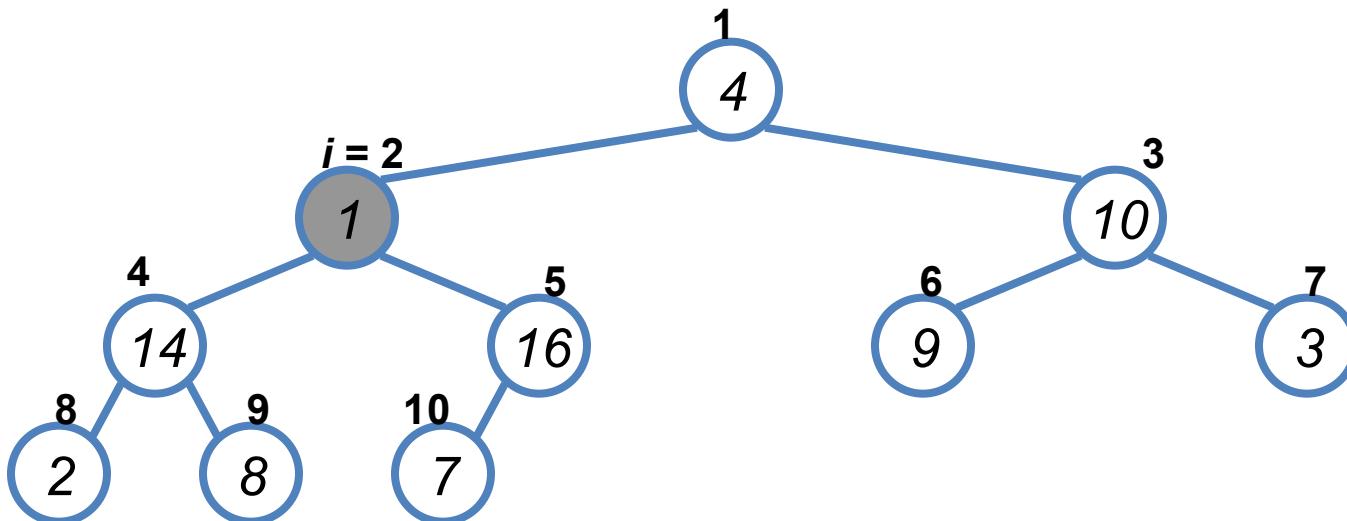
BuildHeap() Example

$A = \{4, 1, 3, \mathbf{14}, 16, 9, 10, \mathbf{2}, 8, 7\}$ •



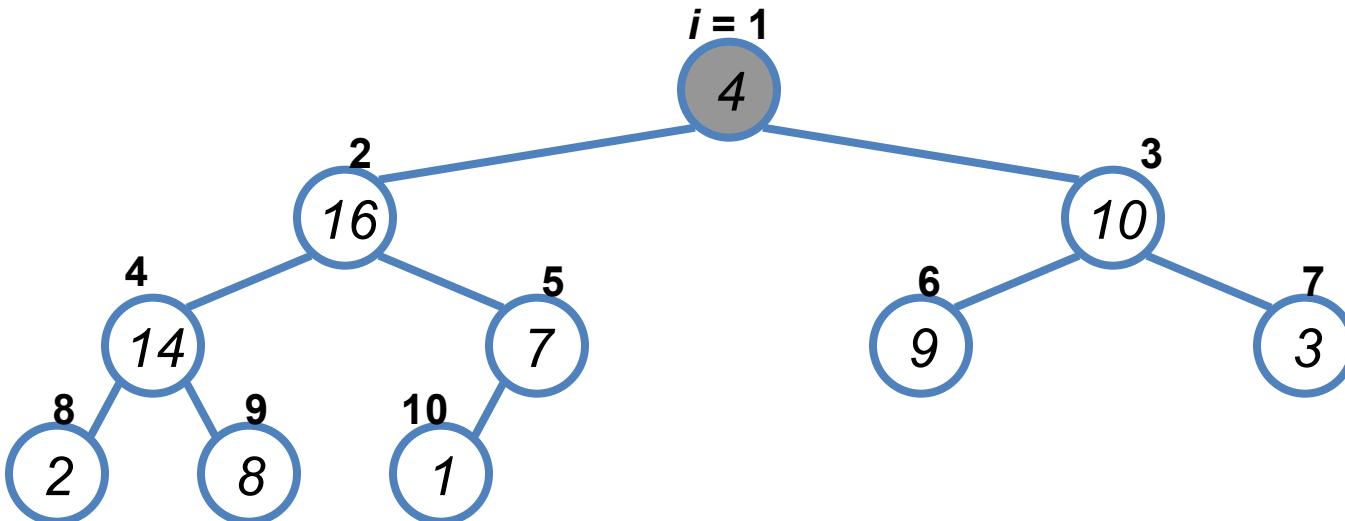
BuildHeap() Example

$A = \{4, 1, \mathbf{10}, 14, 16, 9, \mathbf{3}, 2, 8, 7\}$ •



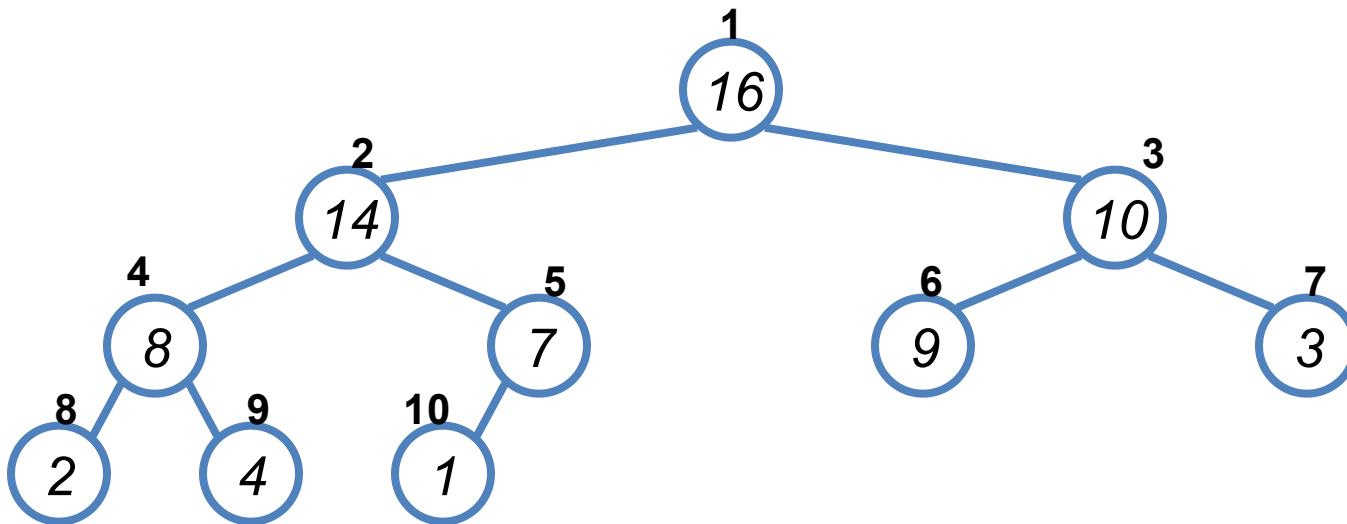
BuildHeap() Example

$A = \{4, 16, 10, 14, 7, 9, 3, 2, 8, 1\}$ •



BuildHeap() Example

$A = \{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$ •



The Heapsort Algorithm

The heapsort algorithm starts by using Build-Heap to build a heap on the input array $A[1..n]$, where $n = \text{length}[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$.

Heapsort(A)

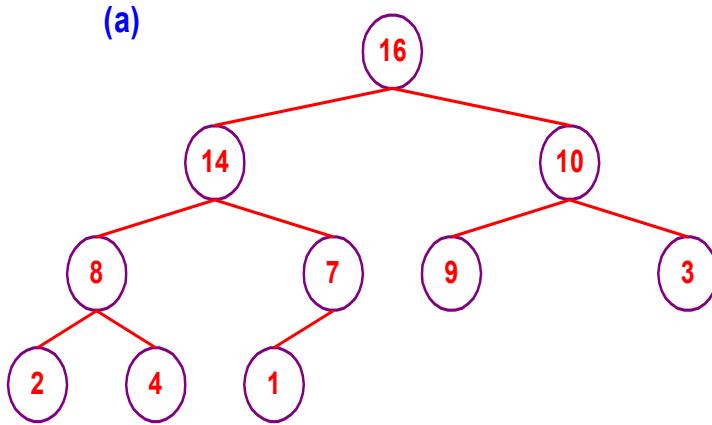
- 1 Build-Heap(A)
- 2 *for* $i \leftarrow \text{length}[A]$ *downto* 2
- 3 do exchange $A[1] \leftarrow A[I]$
- 4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
- 5 Heapify($A, 1$)

Example: The Operation of Heapsort

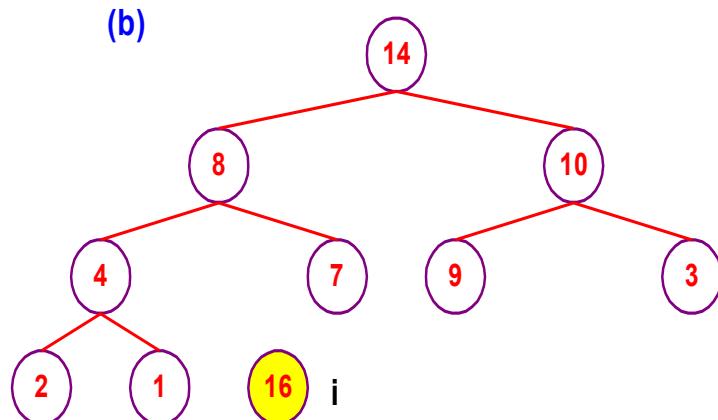
This example shows the *operation of heapsort* after the heap is initially built. Each heap is shown at the beginning of an iteration of the for loop in line 2.

The Heapsort procedure takes time $O(n \lg n)$, since the call to Build-Heap takes time $O(n)$ and each of the $n-1$ calls to Heapify takes time $O(\lg n)$.

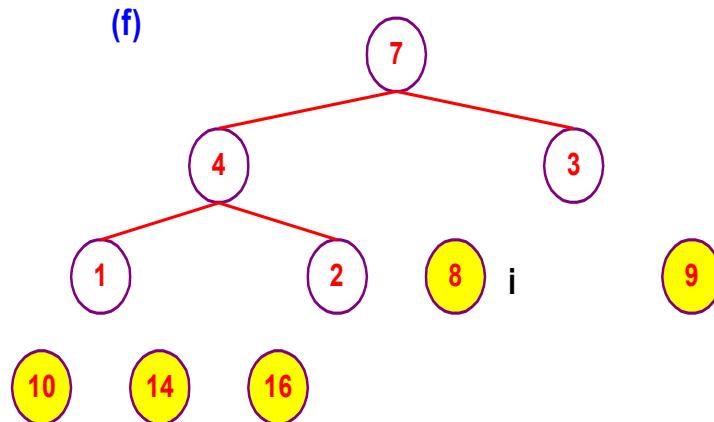
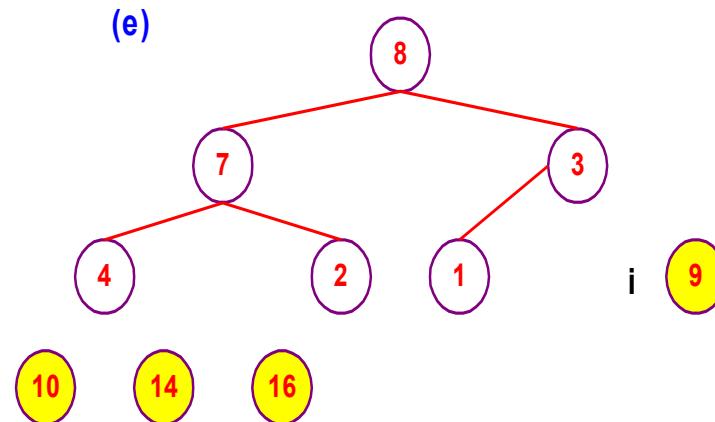
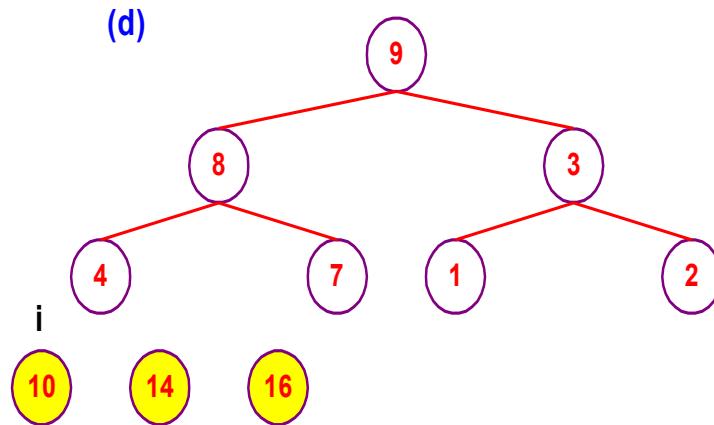
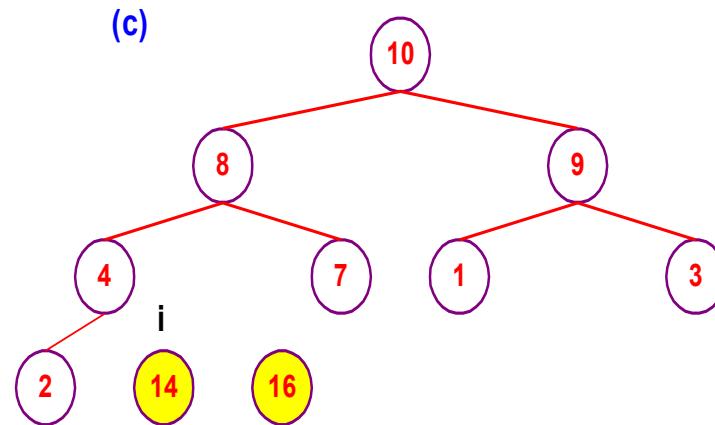
a-the heap data structure just after it has been built by Build-Heap.



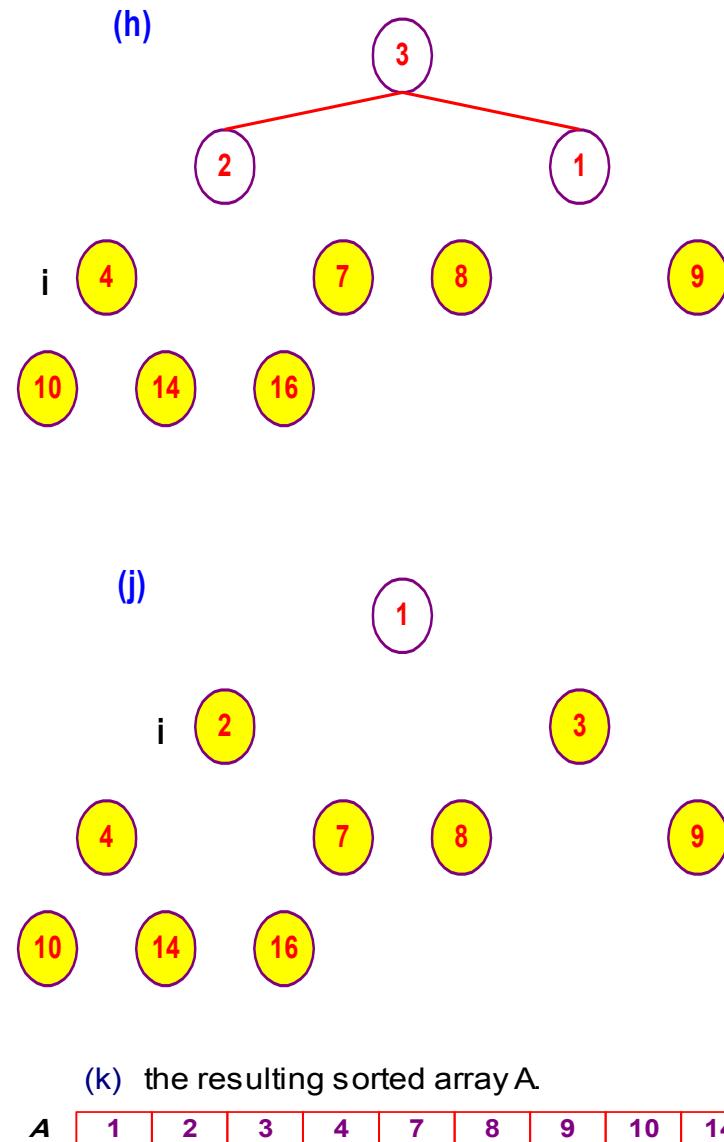
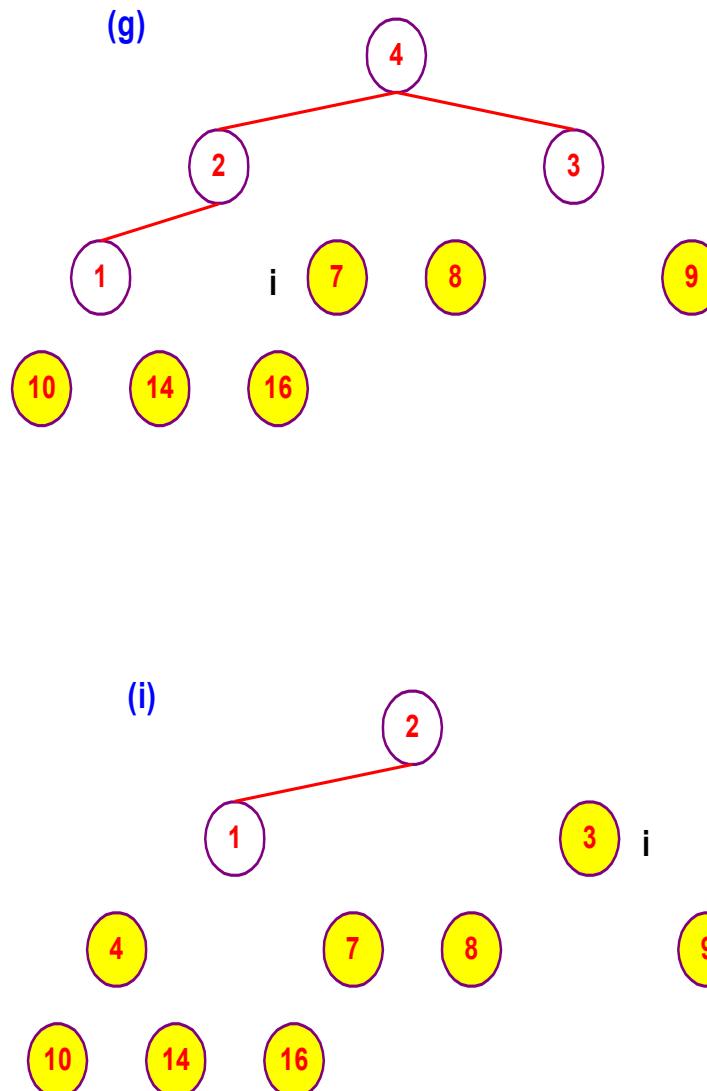
b-j the heap just after call of Heapify in line 5. The value of I at that time is shown. Only lightly shaded nodes remain in the heap.



b-j the heap just after call of Heapify in line 5. The value of I at that time is shown. Only lightly shaded nodes remain in the heap.



b-j the heap just after call of Heapify in line 5. The value of I at that time is shown. Only lightly shaded nodes remain in the heap.



Priority Queue Implementation

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A priority queue supports the following operations.

$\text{INSERT}(S, x)$ inserts the element x into the set S .

$\text{MAXIMUM}(S)$ returns an element of S with the largest key.

$\text{EXTRACT-MAX}(S)$ removes and returns an element of S with the largest key.

Clearly, one can implement the three operations using binary heaps with time complexities $O(\lg n)$, $\Theta(1)$, $O(\lg n)$, respectively.

Note that one can switch between minimization and maximization by negating the keys.

Heapsort Procedure

Heap-Extract-Max(A)

- 1 If heap-size[A] < 1
- 2 then error “heap underflow”
- 3 Max \leftarrow A[1]
- 4 A[1] \leftarrow A[heap-size[A]]
- 5 Heap-size[A] \leftarrow heap-size[A] – 1
- 6 Heapify(A,1)
- 7 Return max

The running time is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for Heapify.

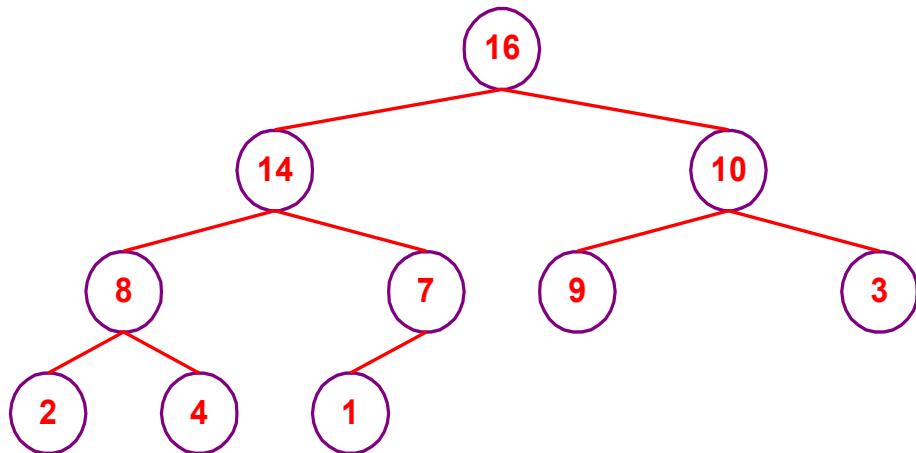
Heap-Insert(A,key)

```
1 heap-size[A] ← heap-size[A] + 1  
2 I ← heap-size[A]  
3 While I > 1 and A[Parent(I)] < key  
4 do A[I] ← A[Parent(I)]  
5 I ← Parent(I)  
6 A[I] ← key
```

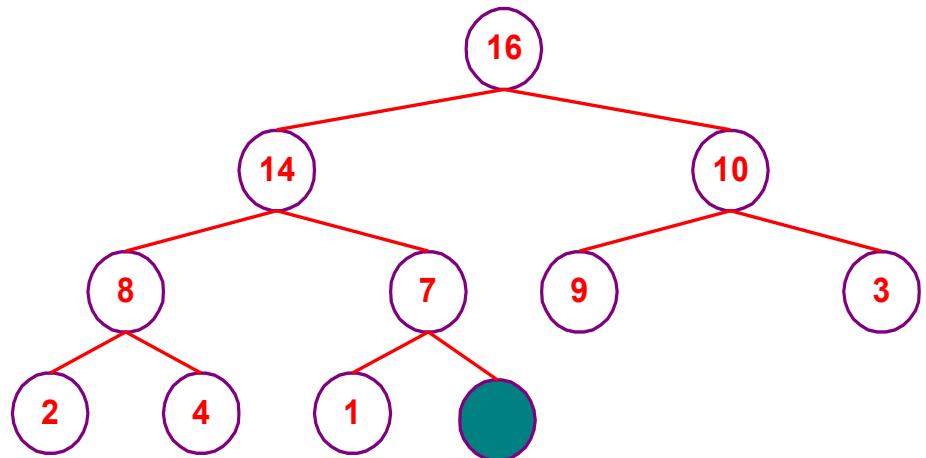
The Heap-Insert procedure inserts a node into a heap A. to do so, it first expands the heap by adding a new leaf to the tree. Then, in a manner reminiscent of the insertion loop (line 5-7) of Insertion-Sort, it traverses a path from this leaf toward the root to find a proper place for the new element.

Example: Heap-Insert Operation

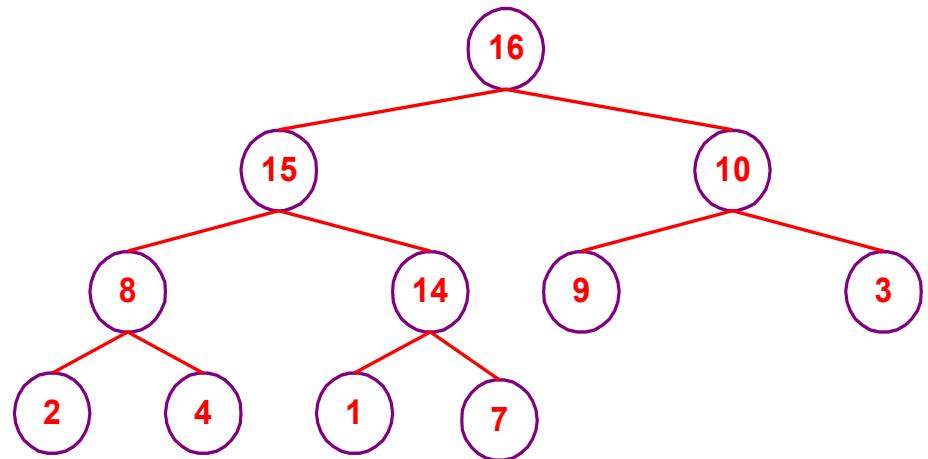
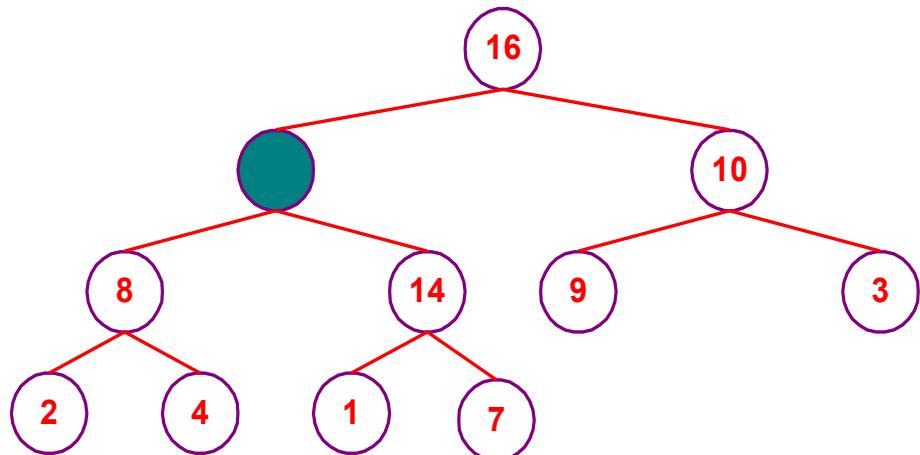
The running time of Heap-Insert on an n -element heap is $O(\lg n)$, since the path traced from the new leaf to the root has length $O(\lg n)$



(a)-the heap of last example before we insert a node with key 15.



(b)-A new leaf is added to the tree.



(c)-Values on the path from the new leaf to the root are copied down until a place for the key 15 is found.

(d)-the key 15 is inserted.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$.

Quicksort

Quicksort, like merge sort, is based on the **divide-and-conquer** paradigm. here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$.

Divide: the array $A[p..r]$ is partitioned (rearranged) into two nonempty subarray $A[p..q]$ and $A[q+1..r]$ such that each element of $A[p..q]$ is less than or equal to each element of $A[q+1..r]$. The index q is computed as part of this partitioning procedure.

Conquer: the two subarrays $A[p..q]$ and $A[q+1..r]$ are sorted by recursive calls to quicksort.

Combine: since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Implements Quicksort

Quicksort(A,p,r)

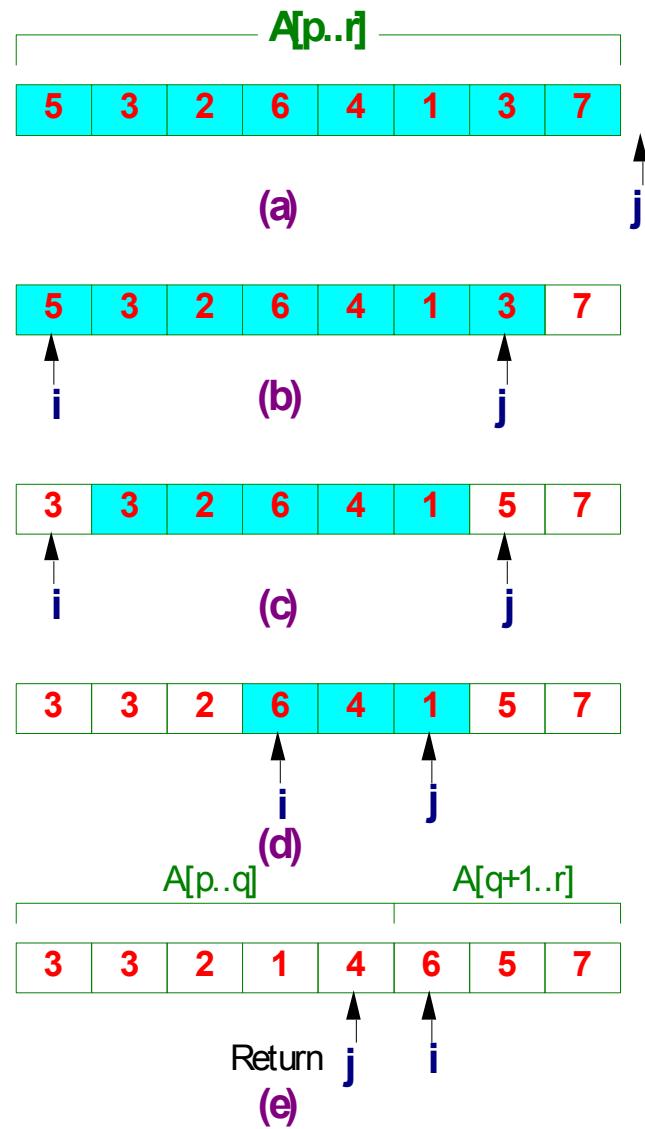
- 1 If $p < r$
- 2 then $q \leftarrow \text{Partition}(A, p, r)$
- 3 *Quicksort(A, p, q)*
- 4 *Quicksort(A, q+1, r)*

To sort an entire array A, the initial call is Quicksort(A,1,length[A])

Partition(A,p,r)

- 1 $x \leftarrow A[p]$
- 2 $\leftarrow p - 1$
- 3 $\leftarrow r + 1$
- 4 While true
- 5 do repeat $\leftarrow j - 1$
- 6 until $A[j] \leq x$
- 7 repeat $\leftarrow I + 1$
- 8 until $A[I] > x$
- 9 if $I < j$
- 10 then exchange $A[I] \longleftrightarrow A[j]$
- 11 else return j

The Operation of Partition



The operation of Partition on a sample array.

Lightly shaded array elements have been placed into the correct partitions, and heavily shaded elements are not yet in their partitions.

- (a) the input array, with the initial value of I and j just off the left and right ends of the array. We partition around $x=A[p]=5$.
- (b) the positions of I and j at line 9 of the first iteration of the while loop.
- (c) the result of exchanging the elements pointed to by I and j in line 10.
- (d) the positions of I and j at line 9 of the second iteration of the while loop.
- (e) the positions of I and j at line 9 of third and last iteration of the while loop. The procedure terminates because $I \geq j$, and the value $q=j$ is returned. Array element up to and including $A[j]$ are less than or equal to $x=5$, and array elements after $A[j]$ are greater than or equal to $x=5$.

Performance of Quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slow as insertion sort.

We shall informally investigate how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

Worst-Case Partitioning

The **worst-case** behavior for quicksort occurs when the partitioning routine produces one region with **$n-1$** elements and one with only **1** element. Let us assume that this unbalanced partitioning arises at every step of the algorithm. Since partitioning costs $\Theta(n)$ time and $T(1) = \Theta(1)$, the recurrence for the running time is :

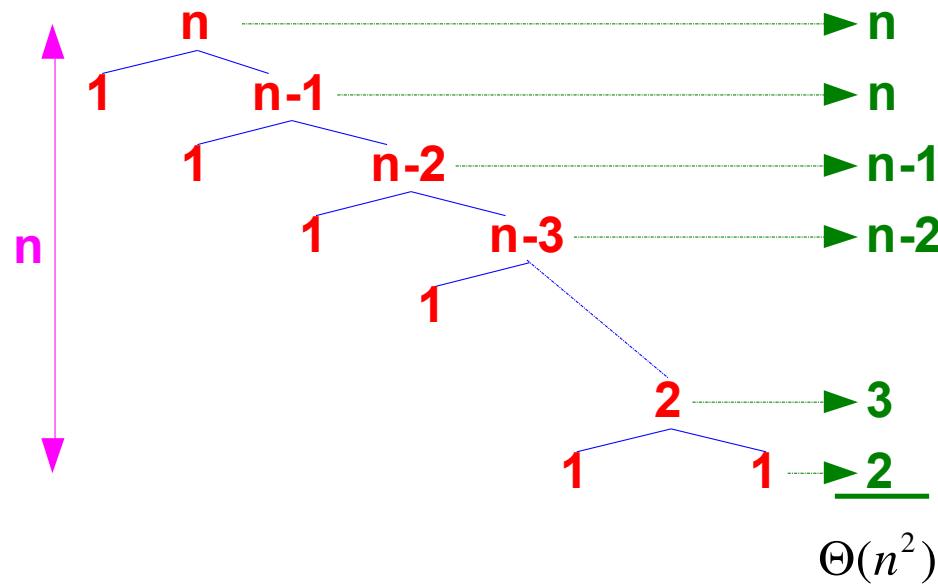
$$T(n) = T(n - 1) + \Theta(n)$$

To evaluate this recurrence, we observe that $T(1) = \Theta(1)$ and then iterate:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= \sum_{k=1}^n \Theta(k) \\ &= \Theta\left(\sum_{k=1}^n k\right) \\ &= \Theta(n^2) \end{aligned}$$

Example: Worst-Case Partitioning

We obtain the last line by observing that $\sum_{k=1}^n k$ is the arithmetic series. The figure shows a recursion tree for this worst-case execution of quicksort. Thus, if the partitioning is maximally unbalanced at every recursive step of the algorithm, the running time is $\Theta(n^2)$. Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted-a common situation in which insertion sort runs in $\Theta(n^2)$ time.

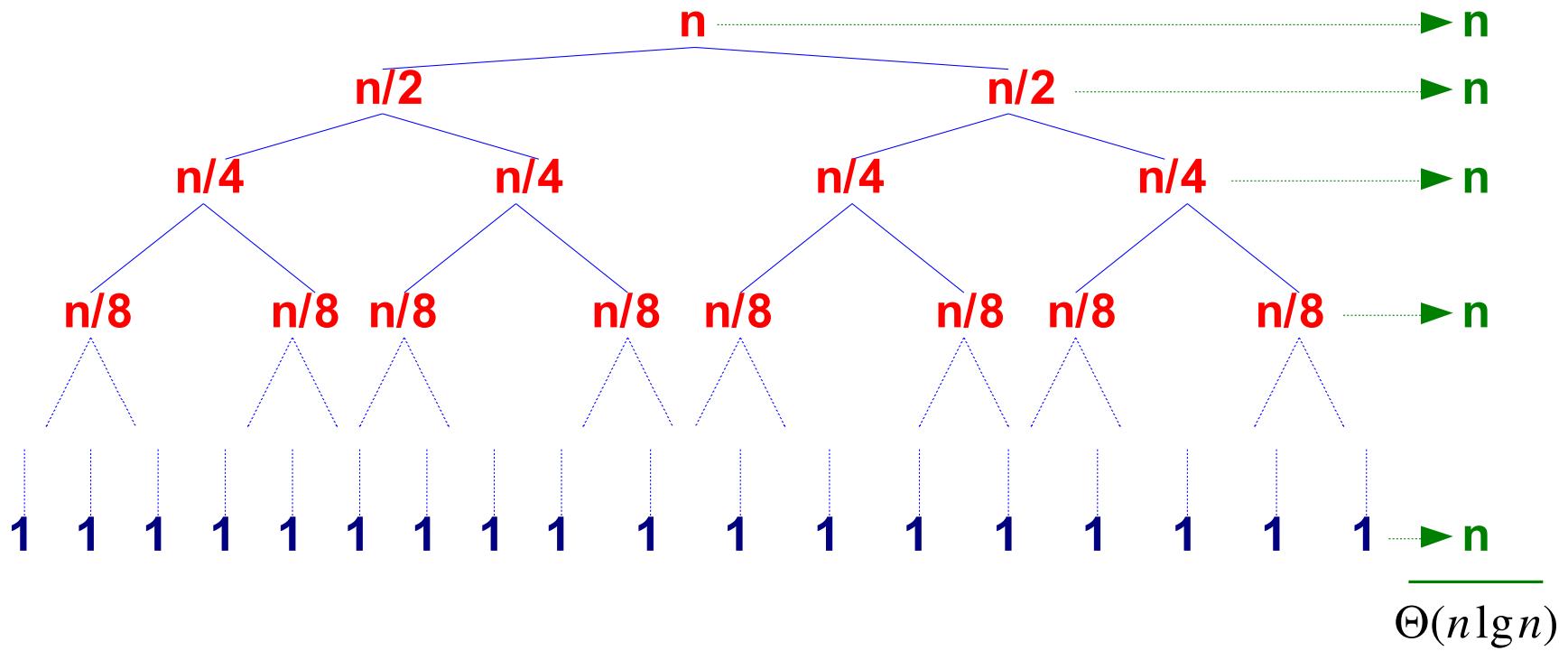


Best-case Partitioning

If the partitioning procedure produces two regions of size $n/2$, quicksort *runs much faster*. The recurrence is then:

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

The solution is $T(n) = \Theta(n \lg n)$, thus, this best-case partitioning produces a much faster algorithm.



Counting Sort

Counting sort assume that each of the n input elements is an integer in the range 1 to k , for some integer k . **when $k=O(n)$** , the sort runs in $O(n)$ time.

The basic idea of counting sort is to determine, for each input element x , the number of elements **less than x** . this information can be used to place element x directly into its position in the output array.

Example, if there are 17 elements less than x , then x belongs in output position 18 . This scheme must be modified slightly to handle the situation in which several elements have the same value, since we don't want to put them all in the same position.

Counting-Sort Algorithm

Counting-Sort(A,B,k)

```
1  For I=1 to k  
2      do c[I]=0  
3  For j=1 to length[A]  
4      do C[A[j]]=C[A[j]]+1  
5  C[I] now contains the number of  
     elements equal to I  
6  For I=2 to k  
7      do C[I]=C[I]+C[I-1]  
8  C[I] now contains the number of  
     elements less than or equal to I  
9  For j=length[A] downto 1  
10     do B[C[A[j]]]=A[j]  
11     C[A[j]]=C[A[j]]-1
```

Note:

We assume that the input is an array **A[1..n]**, and thus **length[A]=n**.

We require two other arrays, the array **B[1..n]** holds the sorted output, and the array **C[1..k]** provides **temporary working storage**.

The Operation of Counting-Sort

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| A | 3 | 6 | 4 | 1 | 3 | 4 | 1 | 4 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 0 | 2 | 3 | 0 | 1 |

(a)

the array A and the auxiliary array C after line 4

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 2 | 4 | 7 | 7 | 8 |

(b)

The array C after line 7

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| B | | | | | | | 4 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 2 | 4 | 6 | 7 | 8 |

(c)

(c)-(e) the output array B and the auxiliary array C after one, two, and three iterations of the loop in line 9-11, respectively. Only the lightly shaded elements of array B have been filled in.

The Operation of Counting-Sort(continue)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| B | 1 | | | | | 4 | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| B | 1 | | | | | 4 | 4 | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| B | 1 | 1 | 3 | 3 | 4 | 4 | 4 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 1 | 2 | 4 | 6 | 7 | 8 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 1 | 2 | 4 | 5 | 7 | 8 |

(d)

(e)

(f)

(c)-(e) the output array B and the auxiliary array C after one, two, and three iterations of the loop in line 9-11, respectively. Only the lightly shaded elements of array B have been filled in.

The final sorted output array B

How much time does counting sort require? The for loop of lines 1-2 takes time $O(k)$, the for loop of lines 3-4 takes time $O(n)$, the for loop of lines 6-7 takes time $O(k)$, and the for loop of lines 9-11 takes time $O(n)$. Thus, the overall time is $O(k+n)$. In practice, we usually use counting sort when we have $k=O(n)$, in which case the running time is $O(n)$.

Binary Search Tree

Prepare By
Dr. Mohammed Salem Atoum

A **Binary Search Tree** is a binary tree with the following properties:

- All items in the left subtree are less than the root.
- All items in the right subtree are greater or equal to the root.
- Each subtree is itself a binary search tree.

Basic Property

- In a **binary search tree**,
- the **left subtree** contains key values **less** than the root
- the **right subtree** contains key values **greater** than or **equal** to the root.

7-1 Basic Concepts

Binary search trees provide an excellent structure for searching a list and at the same time for inserting and deleting data into the list.

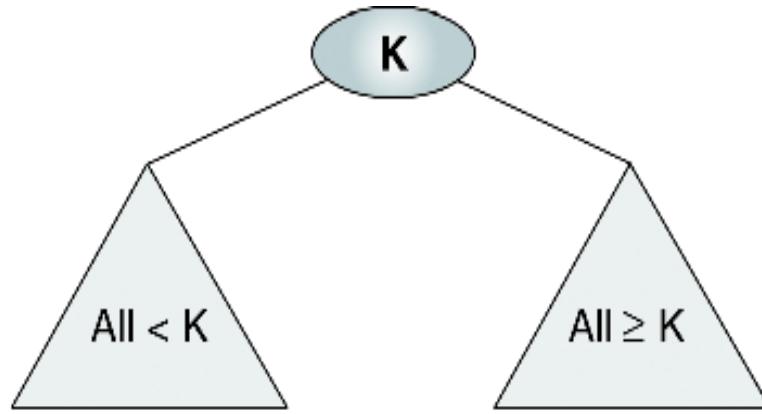
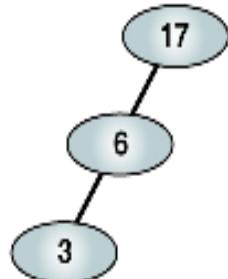
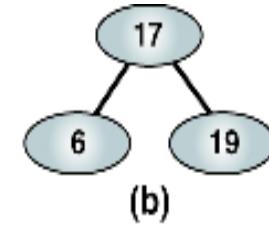


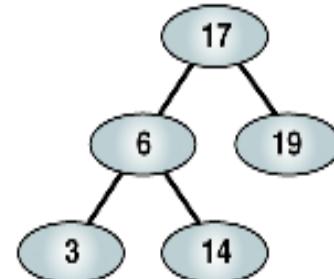
FIGURE 7-1 Binary Search Tree

17

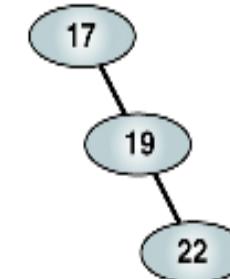
(a)



(c)



(d)



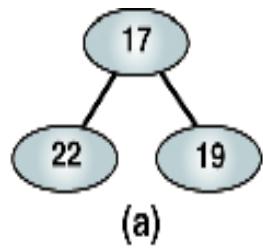
(e)

FIGURE 7-2 Valid Binary Search Trees

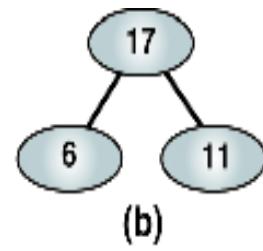
(a), (b) - complete and balanced trees;

(d) – nearly complete and balanced tree;

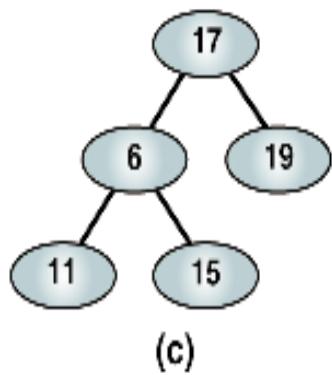
(c), (e) – neither complete nor balanced trees



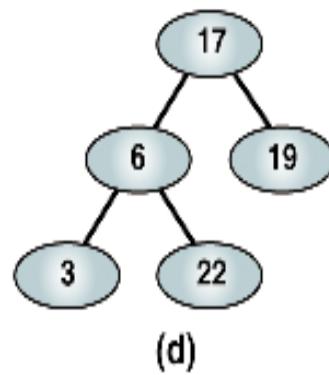
(a)



(b)



(c)



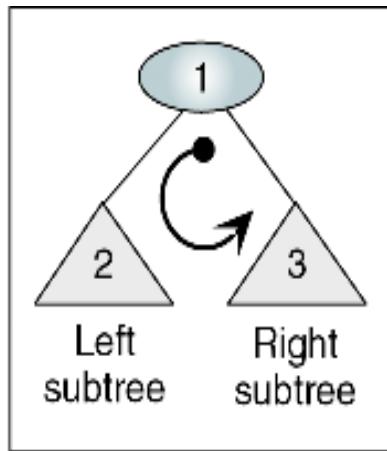
(d)

FIGURE 7-3 Invalid Binary Search Trees

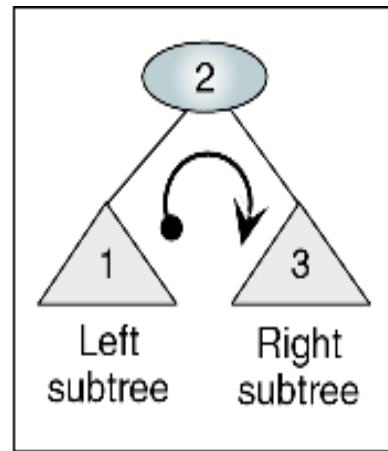
7-2 BST Operations

We discuss four basic BST operations: traversal, search, insert, and delete; and develop algorithms for searches, insertion, and deletion.

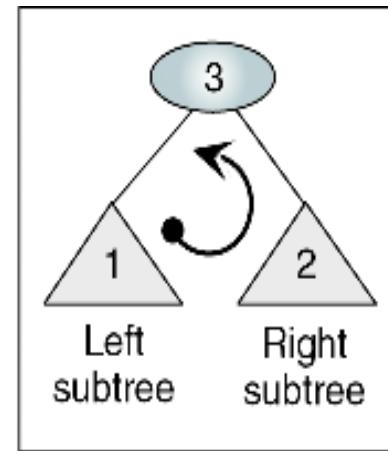
- Traversals
- Searches
- Insertion
- Deletion



(a) Preorder traversal



(b) Inorder traversal



(c) Postorder traversal

FIGURE 6-8 Binary Tree Traversals

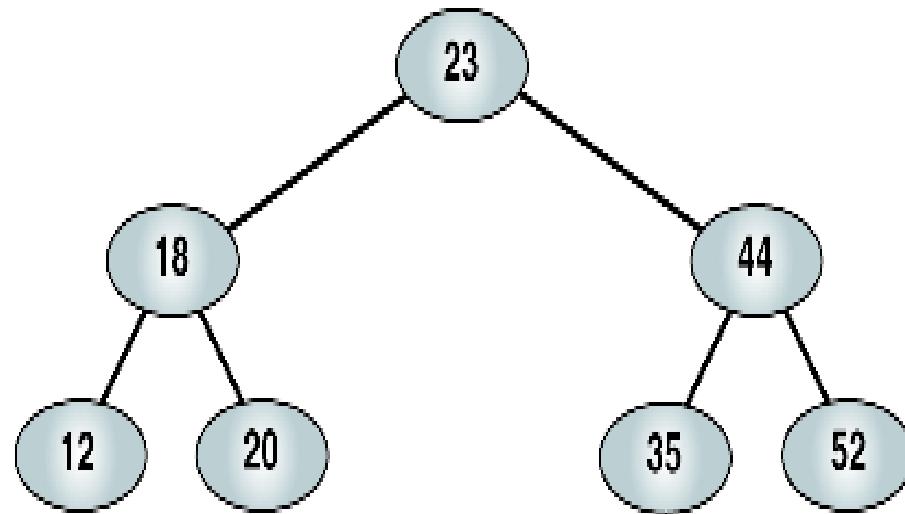


FIGURE 7-4 Example of a Binary Search Tree

Preorder Traversal

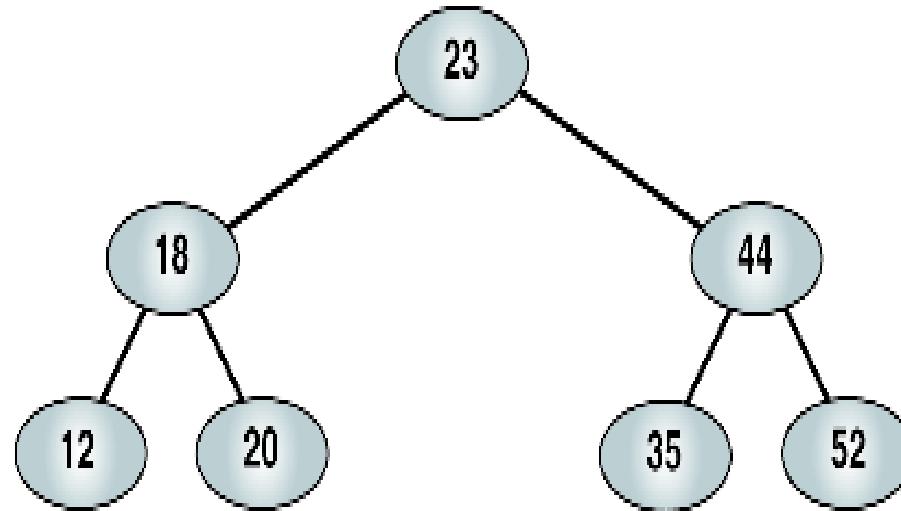


FIGURE 7-4 Example of a Binary Search Tree

23 18 12 20 44 35 52

Postorder Traversal

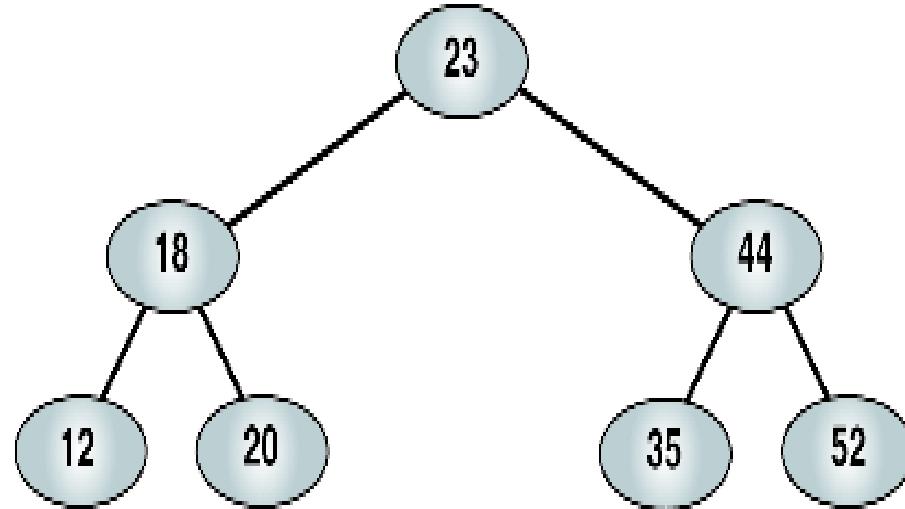


FIGURE 7-4 Example of a Binary Search Tree

12 20 18 35 52 44 23

Inorder Traversal

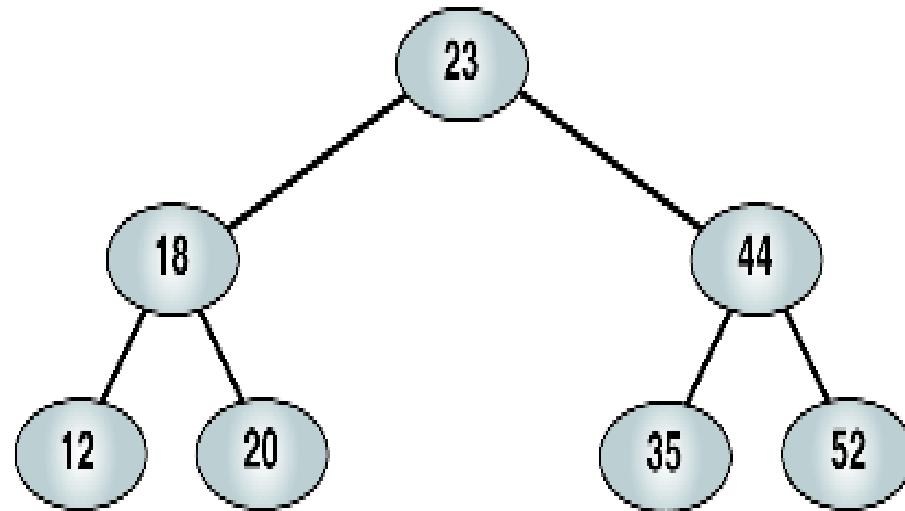


FIGURE 7-4 Example of a Binary Search Tree

12 18 20 23 35 44 52

Inorder traversal of a binary search tree produces a sequenced list

Right-Node-Left Traversal

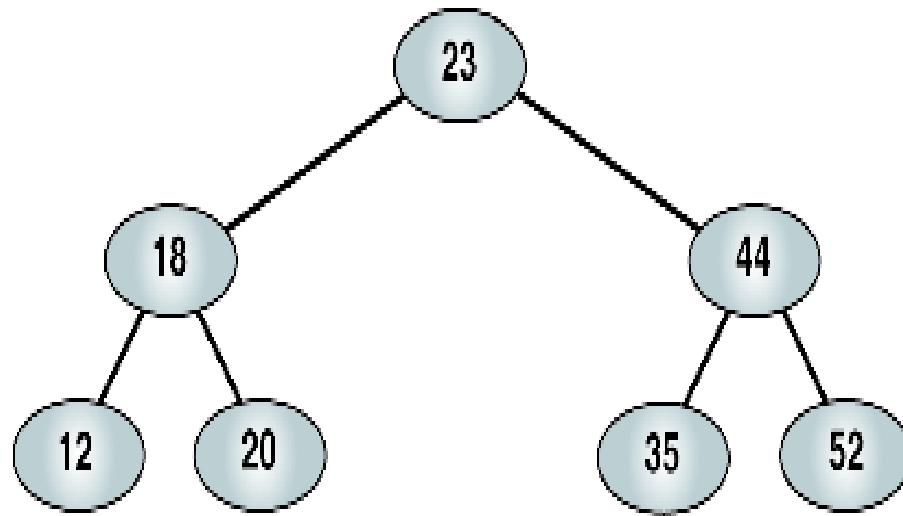


FIGURE 7-4 Example of a Binary Search Tree

52 44 35 23 20 18 12

Right-node-left traversal of a binary search tree produces a descending sequence

Three BST search algorithms:

- Find the **smallest** node
- Find the **largest** node
- Find a **requested** node

ALGORITHM 7-1 Find Smallest Node in a BST

```
Algorithm findSmallestBST (root)
```

This algorithm finds the smallest node in a BST.

Pre root is a pointer to a nonempty BST or subtree

Return address of smallest node

```
1 if (left subtree empty)
```

```
1    return (root)
```

```
2 end if
```

```
3 return findSmallestBST (left subtree)
```

```
end findSmallestBST
```

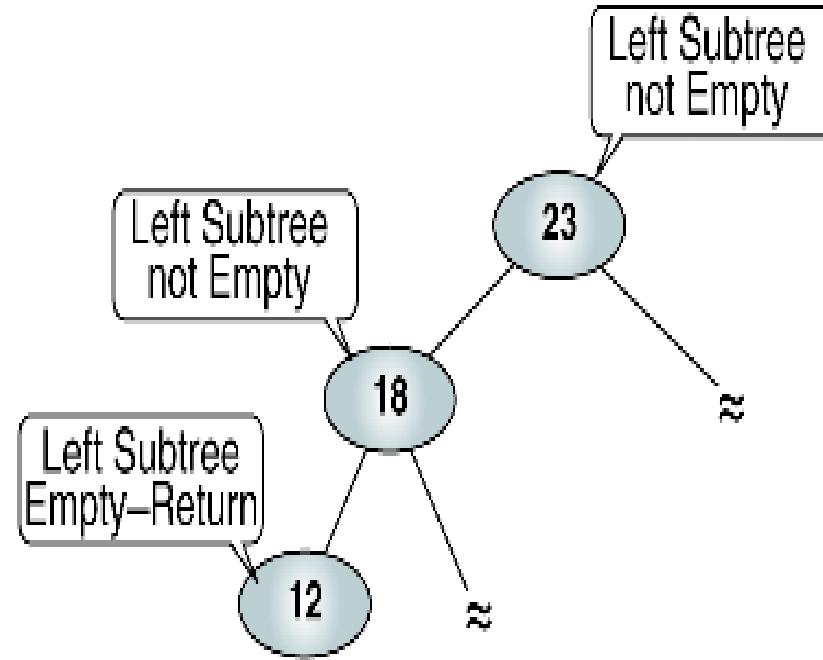


FIGURE 7-5 Find Smallest Node in a BST

ALGORITHM 7-2 Find Largest Node in a BST

```
Algorithm findLargestBST (root)
```

This algorithm finds the largest node in a BST.

Pre root is a pointer to a nonempty BST or subtree

Return address of largest node returned

```
1 if (right subtree empty)
```

```
    1 return (root)
```

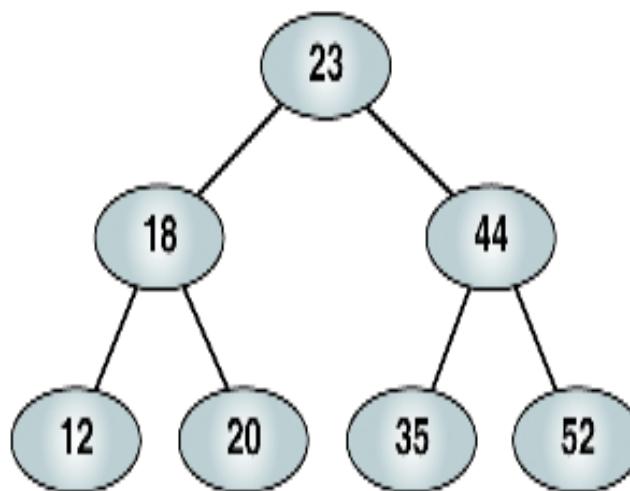
```
2 end if
```

```
3 return findLargestBST (right subtree)
```

```
end findLargestBST
```

Sequenced array

| | | | | | | |
|----|----|----|----|----|----|----|
| 12 | 18 | 20 | 23 | 35 | 44 | 52 |
|----|----|----|----|----|----|----|



Search points in binary search

FIGURE 7-6 BST and the Binary Search

ALGORITHM 7-3 Search BST

```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
    Pre    root is the root to a binary tree or subtree
           targetKey is the key value requested
    Return the node address if the value is found
           null if the node is not in the tree
1  if (empty tree)
    Not found
    1  return null
2 end if
3 if (targetKey < root)
    1  return searchBST (left subtree, targetKey)
4 else if (targetKey > root)
    1  return searchBST (right subtree, targetKey)
5 else
    Found target key
    1  return root
6 end if
end searchBST
```

Target: 20

```
1 if (empty tree)
1 return null
2 end if
3 if (targetKey < root)
1 return searchBST (left subtree, ...)
4 elseif (targetKey > root)
1 return searchBST (right subtree, ...)
5 else
1 return root
6 end if
```

```
1 if (empty tree)
1 return null
2 end if
3 if (targetKey < root)
1 return searchBST (left subtree, ...)
4 elseif (targetKey > root)
1 return searchBST (right subtree, ...)
5 else
1 return root
6 end if
```

```
1 if (empty tree)
1 return null
2 end if
3 if (targetKey < root)
1 return searchBST (left subtree, ...)
4 elseif (targetKey > root)
1 return searchBST (right subtree, ...)
5 else
1 return root
6 end if
```

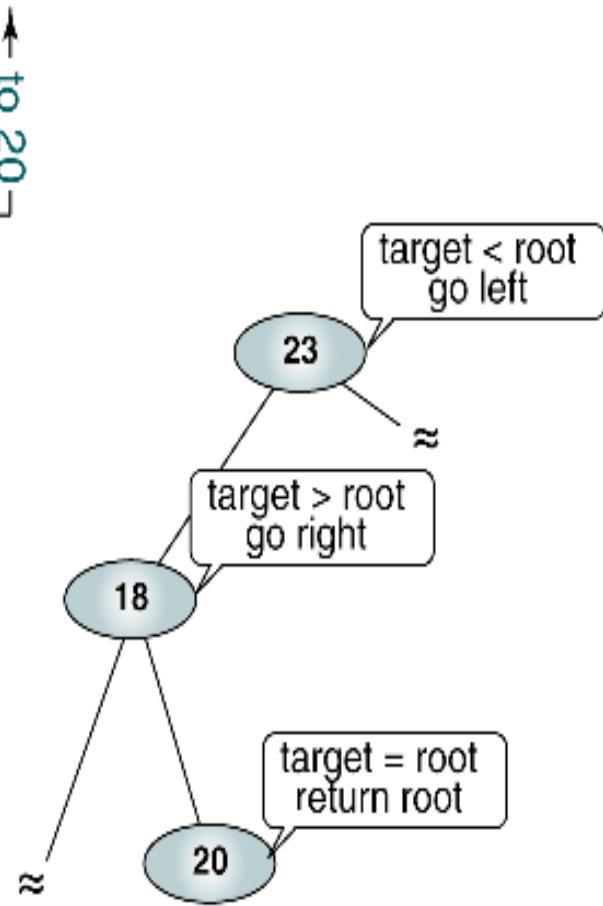
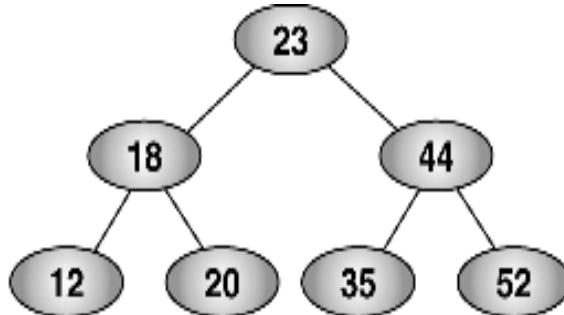


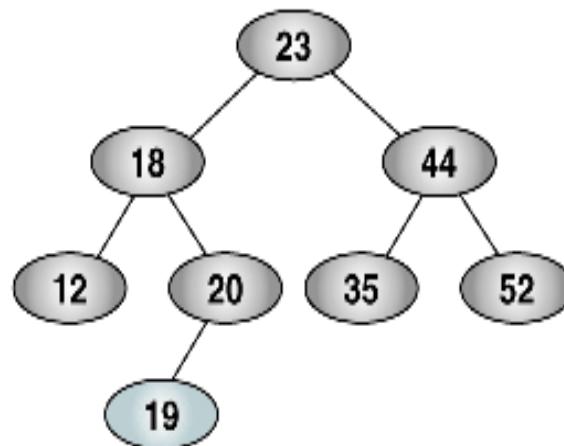
FIGURE 7-7 Searching a BST

BST Insertion

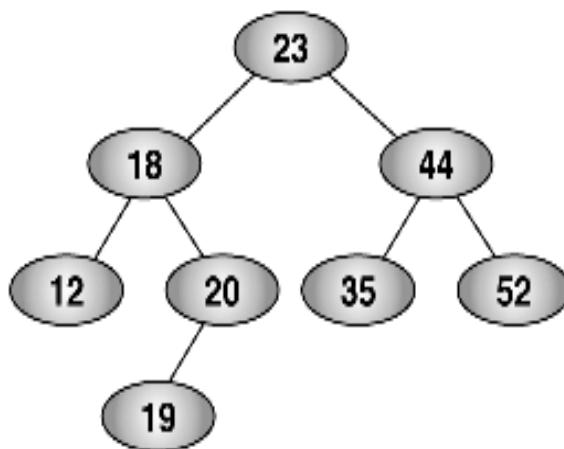
- To insert data all we need to do is follow the branches to an empty subtree and then insert the new node.
- In other words, all inserts take place at a leaf or at a leaflike node – a node that has only one null subtree.



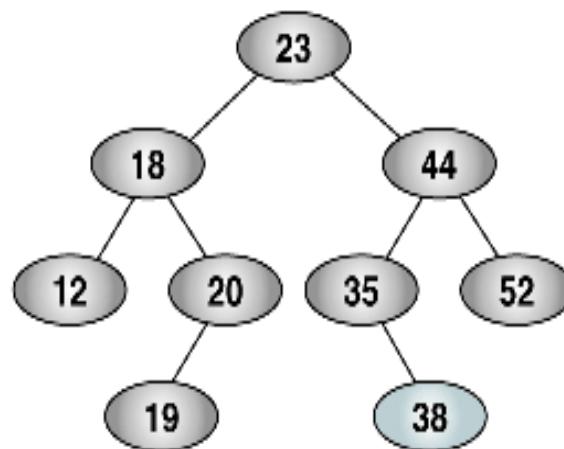
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38



(d) After inserting 38

FIGURE 7-8 BST Insertion

ALGORITHM 7-4 Add Node to BST

Algorithm addBST (root, newNode)

Insert node containing new data into BST using recursion.

Pre root is address of current node in a BST

newNode is address of node containing data

Post newNode inserted into the tree

Return address of potential new tree root

```
1 if (empty tree)
    1 set root to newNode
    2 return newNode
2 end if
```

continued

ALGORITHM 7-4 Add Node to BST (*continued*)

```
    Locate null subtree for insertion
3 if (newNode < root)
    1 return addBST (left subtree, newNode)
4 else
    1 return addBST (right subtree, newNode)
5 end if
end addBST
```

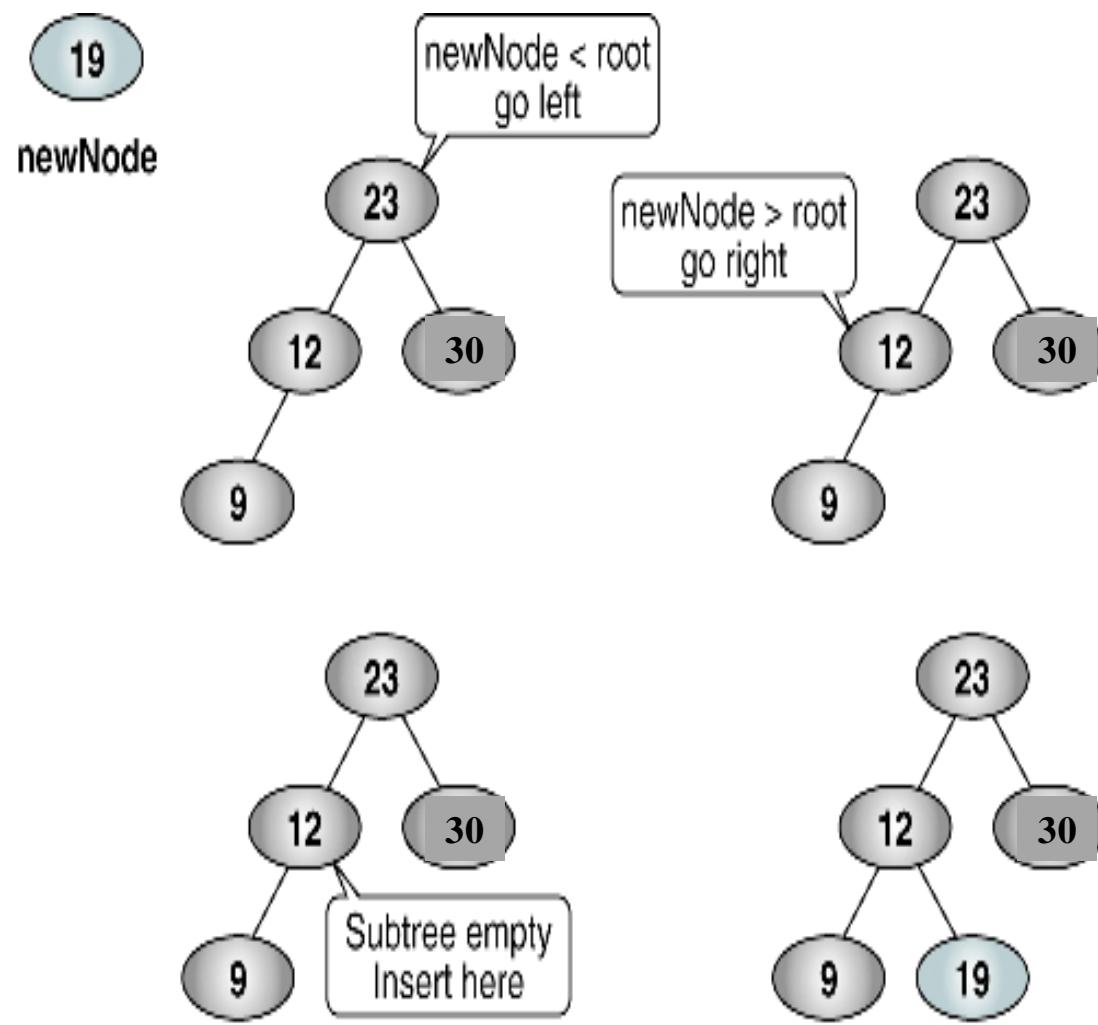


FIGURE 7-9 Trace of Recursive BST Insert

Deletion

- There are the following possible cases when we delete a node:
- The node to be deleted has no children. In this case, all we need to do is delete the node.
- The node to be deleted has only a right subtree. We delete the node and attach the right subtree to the deleted node's parent.
- The node to be deleted has only a left subtree. We delete the node and attach the left subtree to the deleted node's parent.
- The node to be deleted has two subtrees. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced trees.

Deletion from the middle of a tree

- Rather than simply delete the node, we try to maintain the existing structure as much as possible by finding data to take the place of the deleted data. This can be done in one of two ways.

Deletion from the middle of a tree

- We can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.
- We can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.
- Either of these moves preserves the integrity of the binary search tree.

ALGORITHM 7-5 Delete Node from BST

Algorithm deleteBST (root, dltKey)

This algorithm deletes a node from a BST.

Pre root is reference to node to be deleted
 dltKey is key of node to be deleted

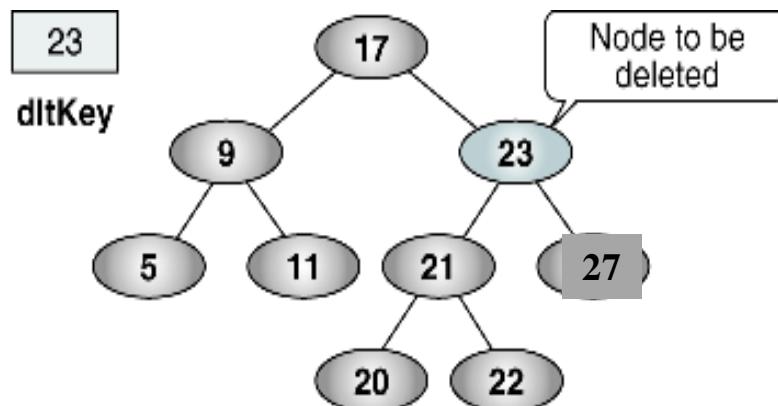
Post node deleted
 if dltKey not found, root unchanged

Return true if node deleted, false if not found

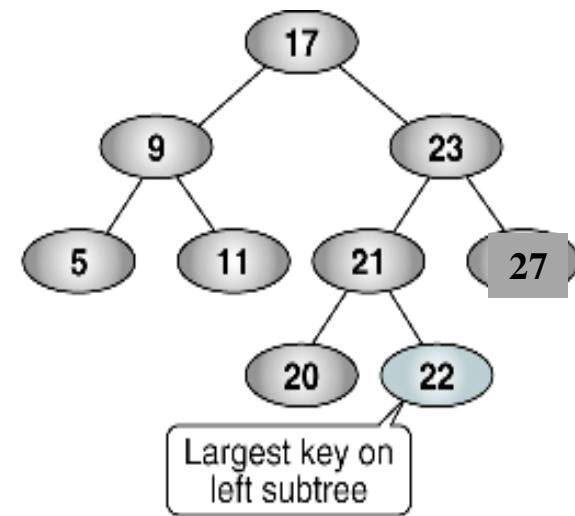
```
1 if (empty tree)
  1 return false
2 end if
3 if (dltKey < root)
  1 return deleteBST (left subtree, dltKey)
4 else if (dltKey > root)
  1 return deleteBST (right subtree, dltKey)
5 else
    Delete node found--test for leaf node
    1 If (no left subtree)
      1 make right subtree the root
      2 return true
```

ALGORITHM 7-5 Delete Node from BST (continued)

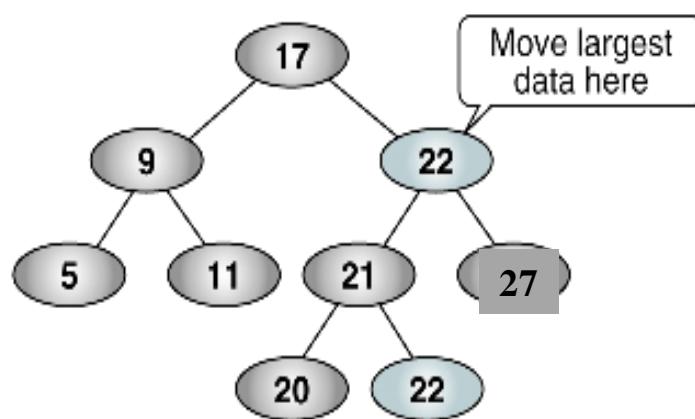
```
2 else if (no right subtree)
    1 make left subtree the root
    2 return true
3 else
    Node to be deleted not a leaf. Find largest node on
    left subtree.
    1 save root in deleteNode
    2 set largest to largestBST (left subtree)
    3 move data in largest to deleteNode
    4 return deleteBST (left subtree of deleteNode,
                        key of largest
4 end if
6 end if
end deleteBST
```



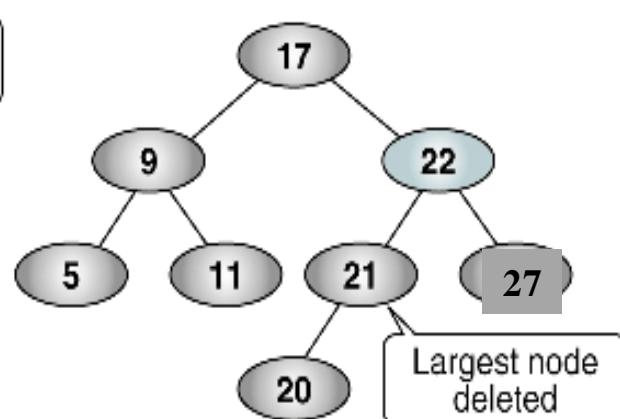
(a) Find dltKey



(b) Find largest



(c) Move largest data



(d) Delete largest node

FIGURE 7-10 Delete BST Test Cases

7-3 Binary Search Tree ADT

We begin this section with a discussion of the BST data structure and write the header file for the ADT. We then develop 14 programs that we include in the ADT.

- Data Structure
- Algorithms

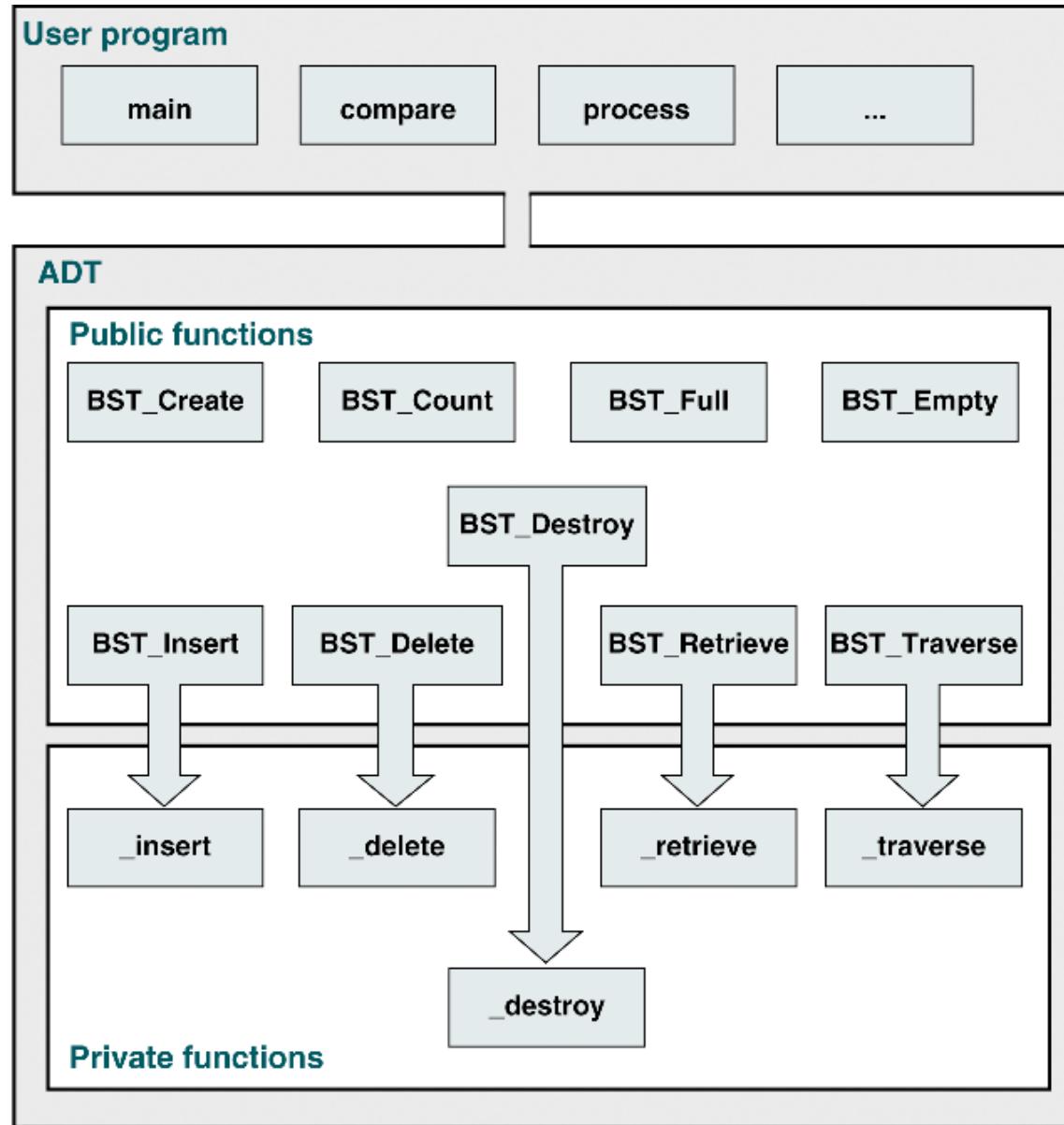
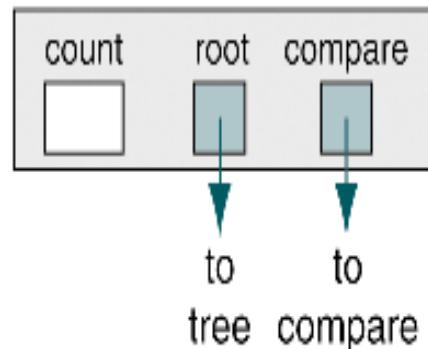
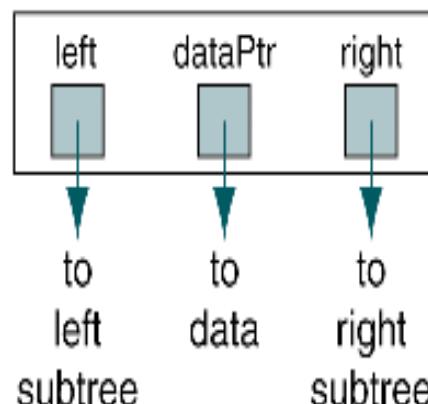


FIGURE 7-11 BST ADT Design

BST_TREE



NODE



```
typedef struct
{
    int count;
    int (*compare)
        (void* arg1,
         void* arg2);
    NODE* root;
} BST_TREE;
```

```
typedef struct node
{
    void* dataPtr;
    struct node* left;
    struct node* right;
} NODE;
```

FIGURE 7-12 BST Tree Data Structure

Algorithm Design & Analysis

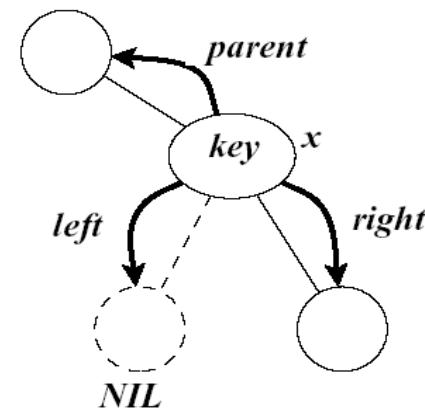
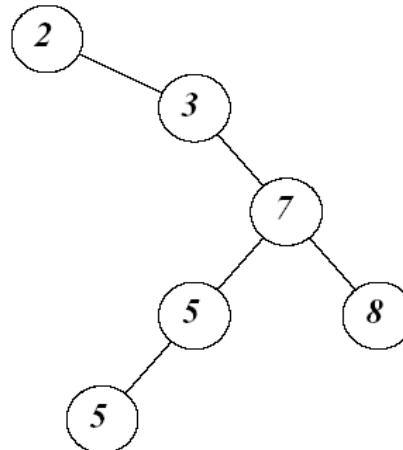
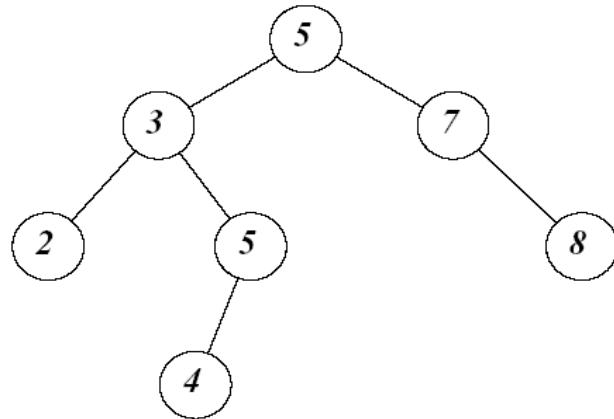
Chapter 5 : Binary Search Trees Red-Black Trees

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

Binary Search Trees

A binary search tree is a binary tree represented by a linked data structure. Each node has a **key** field and the three node-pointer fields **left**, **right** and **parent**.



In addition, the keys must satisfy the **binary-search-tree property**:

$$key[y] \leq key[x] \quad \forall \text{ node } y \text{ in the left subtree of } x$$

$$key[z] \geq key[x] \quad \forall \text{ node } z \text{ in the right subtree of } x.$$

Preorder, Inorder, Postorder Walks

Given a node x , do recursively.

- Preorder(x) : Visit x , the left subtree, the right subtree,
- Inorder(x) : Visit the left subtree, x , the right subtree,
- Postorder(x) : Visit the left subtree, the right subtree, x .

INORDER-TREE-WALK (x)

```
1  if  x <> NIL
2      then INORDER-TREE-WALK (left [x] )
3          print key [x]
4          INORDER-TREE-WALK (right [x] )
```

The complexity of INORDER-TREE-WALK is $\Theta(n)$.

Easy to prove by the substitution argument, see Theorem 13[12].1.

Querying a Binary Search Tree

Recursive Search

```
TREE-SEARCH (x, k)
1 if x = NIL or k = key[x]
2     then return x
3 if k < key[x]
4     then return TREE-SEARCH (left[x], k)
5     else return TREE-SEARCH (right[x], k)
```

Non-Recursive Search

```
ITERATIVE-TREE-SEARCH (x, k)
1 while x <> NIL and k <> key[x]
2     do if k < key[x]
3         then x <- left[x]
4         else x <- right[x]
5     return x
```

Min, Max, Successor in a Binary Search Tree

Minimum-key Node Search

```
TREE-MINIMUM (x)
1   while left [x] <> NIL
2       do x <- left [x]
3   return x
```

Maximum-key Node Search

```
TREE-MAXIMUM (x)
1   while right [x] <> NIL
2       do x <- right [x]
3   return x
```

Successor Node Search

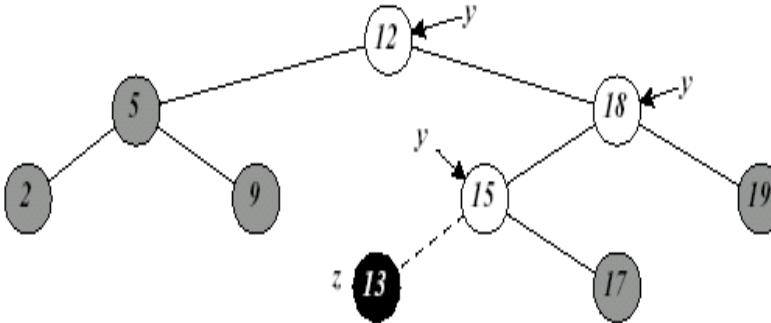
```
TREE SUCCESSOR (x)
1   if right [x] <> NIL
2       then return TREE-MINIMUM(right [x] )
3   y <- parent [x]
4   while y <> NIL and x = right [y]
5       do x <- y
6           y <- parent [y]
7   return y
```

They all run in $O(h)$ time, where h is the height of the binary tree.

Insertion in a Binary Search Tree

```
TREE-INSERT(T, z)
```

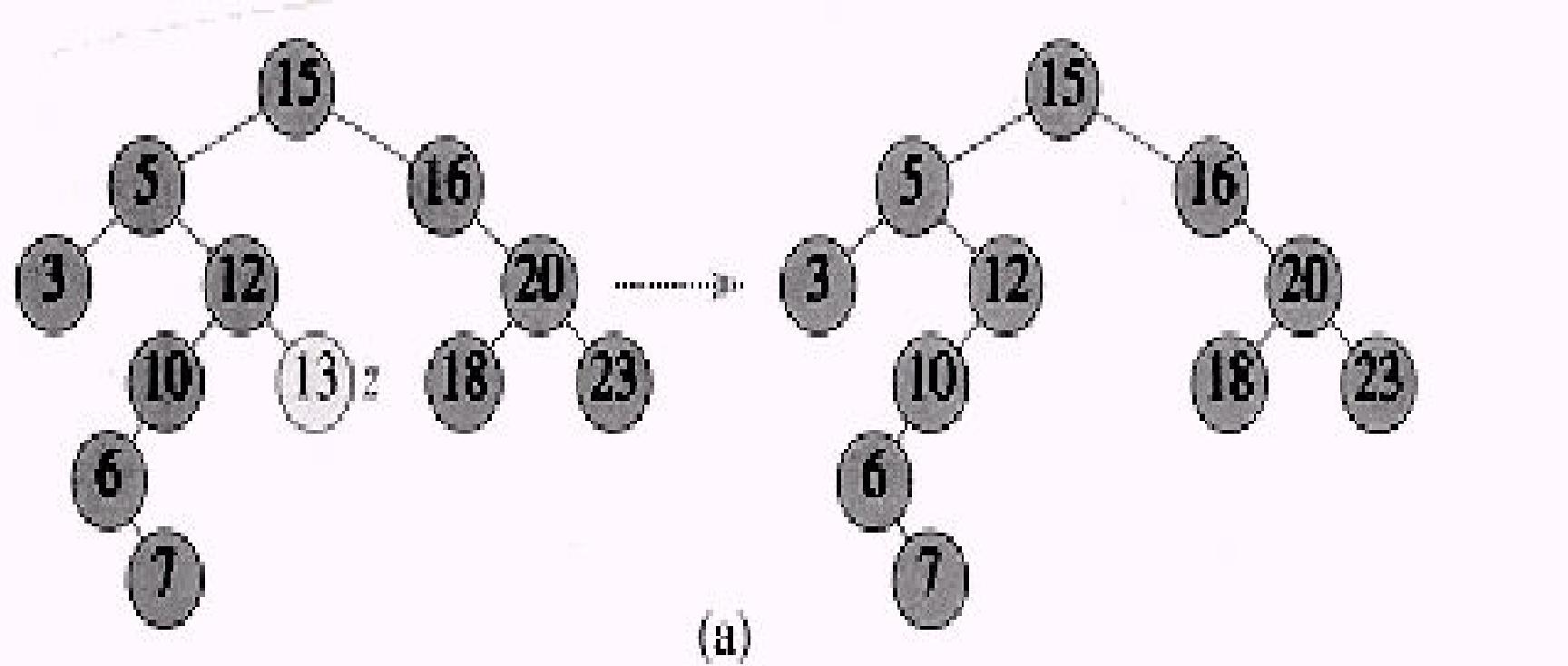
```
1  y <- NIL
2  x <- root[T]
3  while x <> NIL
4      do y <- x
5          if key[z] < key[x]
6              then x <- left[x]
7          else x <- right[x]
8  p[z] <- y
9  if y = NIL
10     then root[T] <- z           /* Tree T was empty */
11     else if key[z] < key[y]
12         then left[y] <- z
13         else right[y] <- z
```



Insert Note:

Tree-Insert begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. After initialization, the while loop in line 3-7 causes these two pointers to move down the tree, going left or right depending on the comparison of key[z] with key[x], until x is set to NIL. This NIL occupies the position where we wish to place the input item z. Lines 8-13 set the pointers that cause z to be inserted. It runs in O(h) time on a tree of height h.

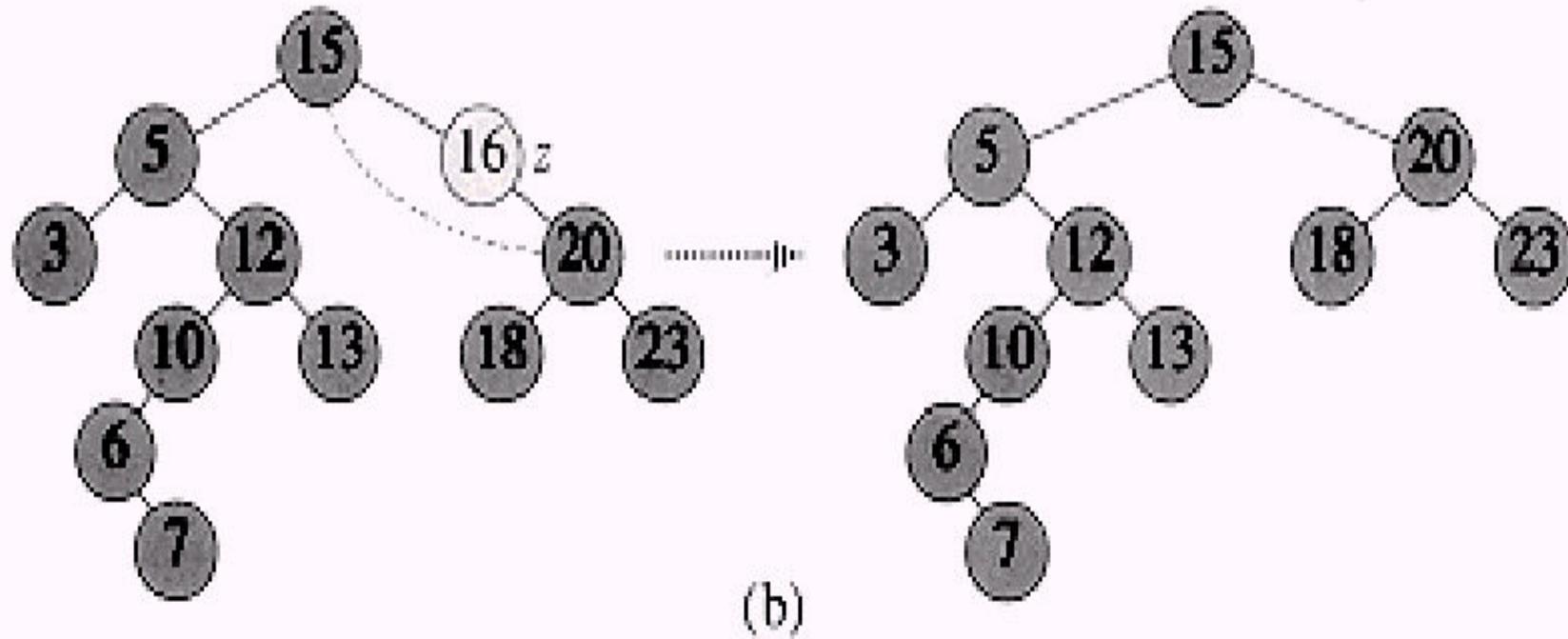
Deletion in a Binary Search Tree



Deleting a node z from a binary search tree. In each case, the node actually removed is lightly shaded.

(a) If z has no children, we just remove it.

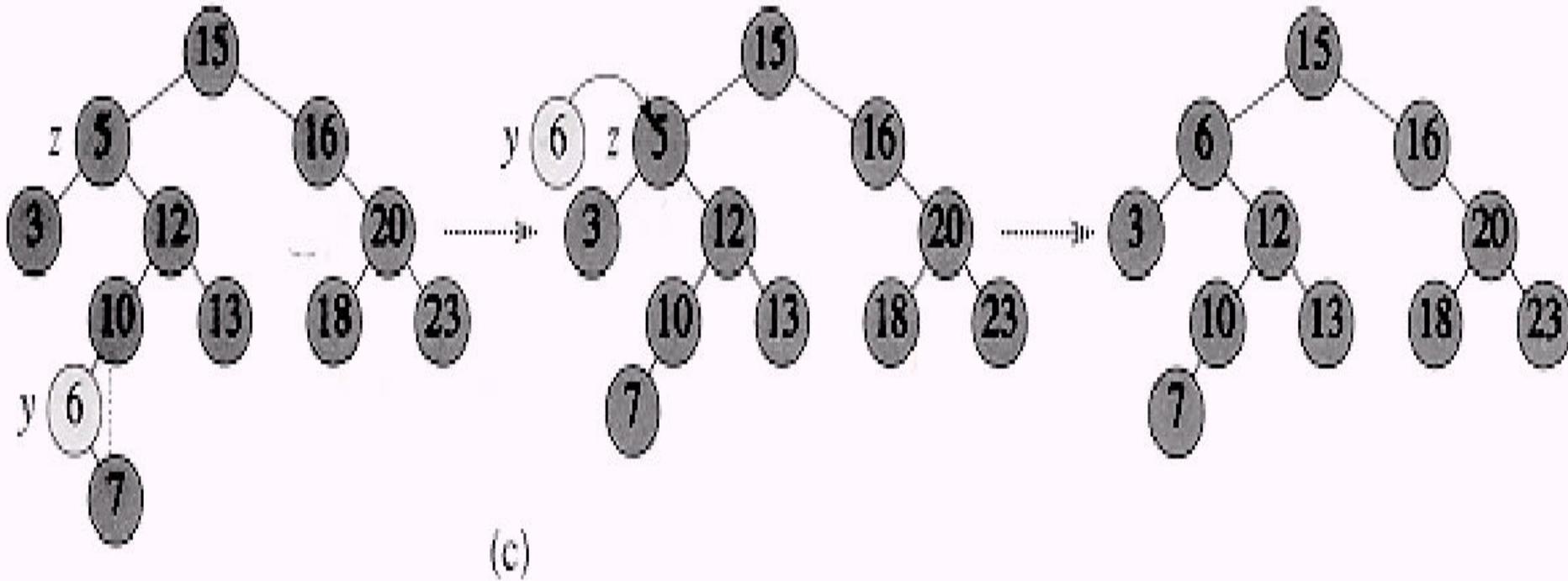
Deletion in a Binary Search Tree



Deleting a node z from a binary search tree. In each case, the node actually removed is lightly shaded.

(b) If z has only one child, we splice out z .

Deletion in a Binary Search Tree



Deleting a node **z from a binary search tree. In each case, the node actually removed is lightly shaded.**

(c) If **z has two children, we splice out its successor **y**, which has at most one child, and then replace the contents of **z** with the contents of **y**.**

Best and Worst Cases of Binary Search Trees

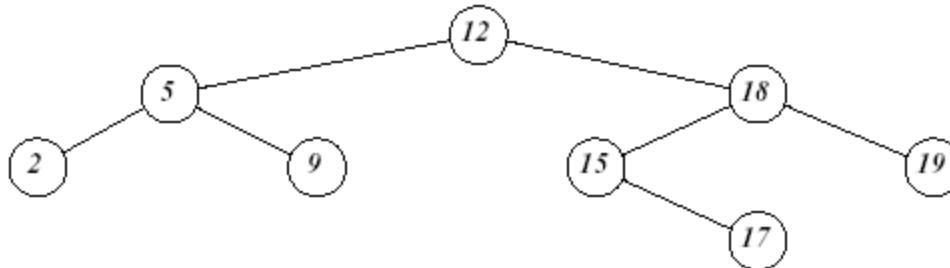
- What sequence of keys makes the insertion-built binary search tree short (i.e. $O(\lg n)$) or tall (i.e. $\Theta(n)$)?
- A random sequence of n distinct keys? The expected height of the insertion-built tree?

The expected height is $O(\lg n)$. (See Section 13[12].4.)

- When keys are character strings of a bounded length, $a = a_1 \cdots a_p$ where $p \leq k$ for some constant k , one can build a binary search tree (of order as large as n^k with n representing the number of characters), called Radix tree, with height at most k . (See Problem 13[12]-2.)

Balanced Trees: Red-Black Trees

A binary search tree is called balanced or AVL-tree if the heights of the left and the right subtrees of each node differ by at most one.



- The height h of a balanced tree of order n is $O(\lg n)$. This can be proved using the fact that $n \geq F_h$, where F_h is the h th Fibonacci number. Note that F_h grows exponentially in h .
- There are ways to implement $O(\lg n)$ -time INSERT and DELETE to build and maintain an AVL tree, although it is quite cumbersome.
- There are simpler variants of similar performance. A red-black tree is an approximately-balanced tree whose height is $O(\lg n)$. A splay tree is a “self-adjusting” tree with $O(\lg n)$ amortized complexity.

Red-Black Trees

Searching - Re-visited

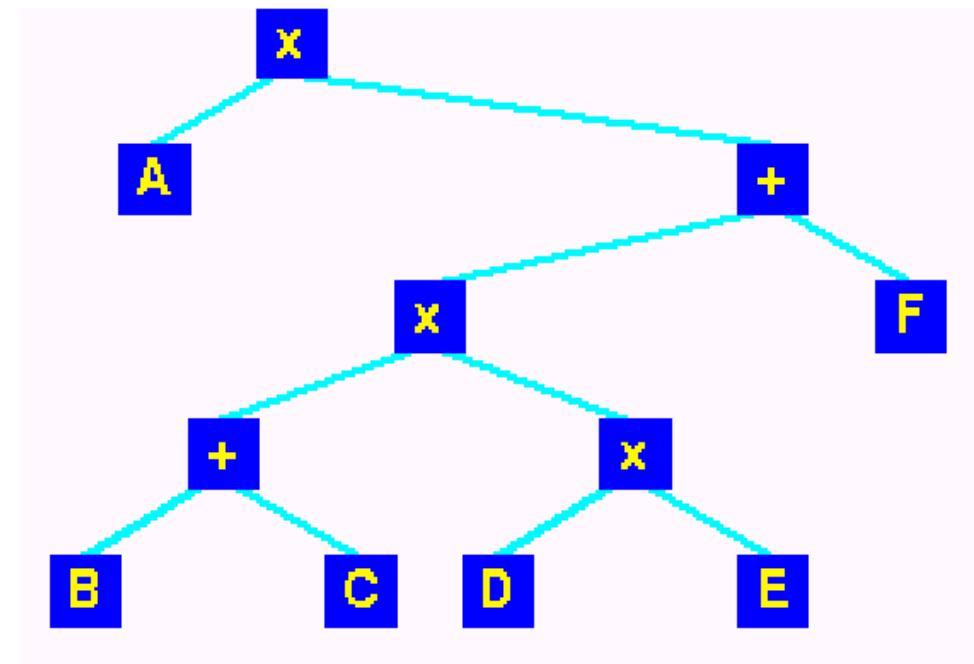
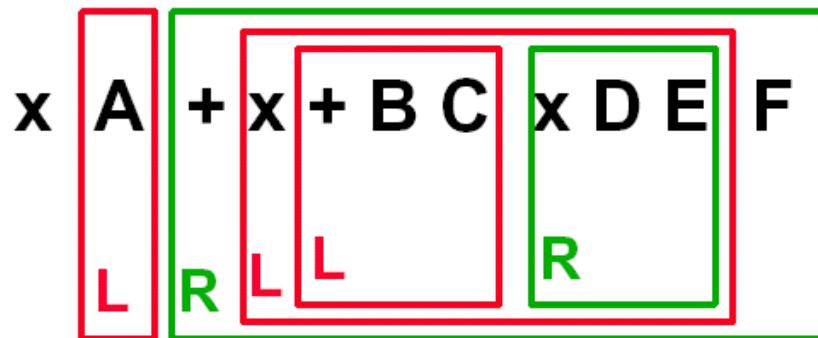
- **Binary tree $O(\log n)$ *if it stays balanced***
 - Simple binary tree good for **static collections**
 - Low (preferably zero) frequency of **insertions/deletions**
- ***but my collection keeps changing!***
 - It's **dynamic**
 - Need to keep the tree **balanced**
- **First, examine some basic tree operations**
 - Useful in several ways!

Tree Traversal

- Traversal = visiting every node of a tree
- Three basic alternatives

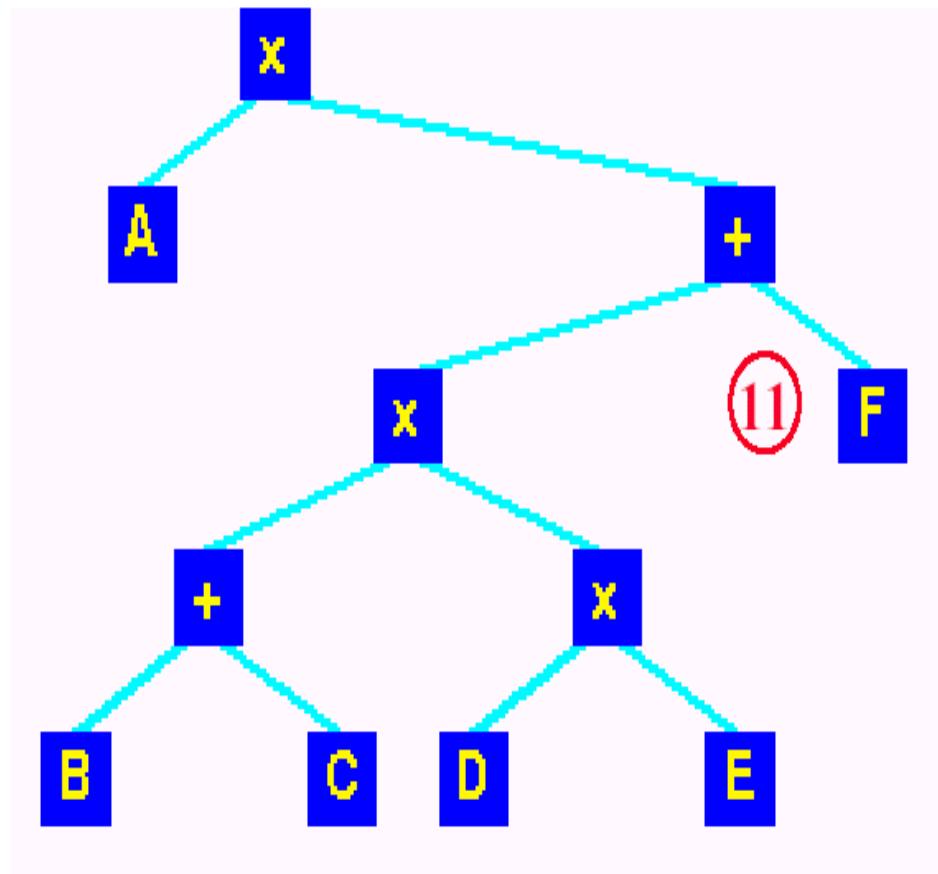
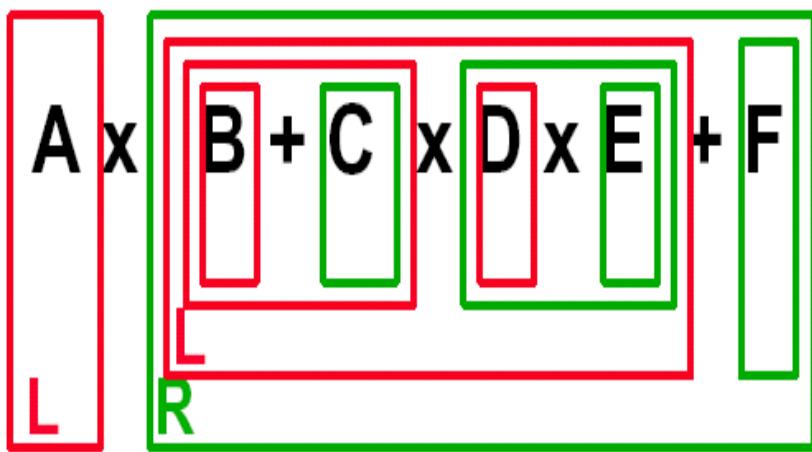
① Pre-order

- Root
- Left sub-tree
- Right sub-tree



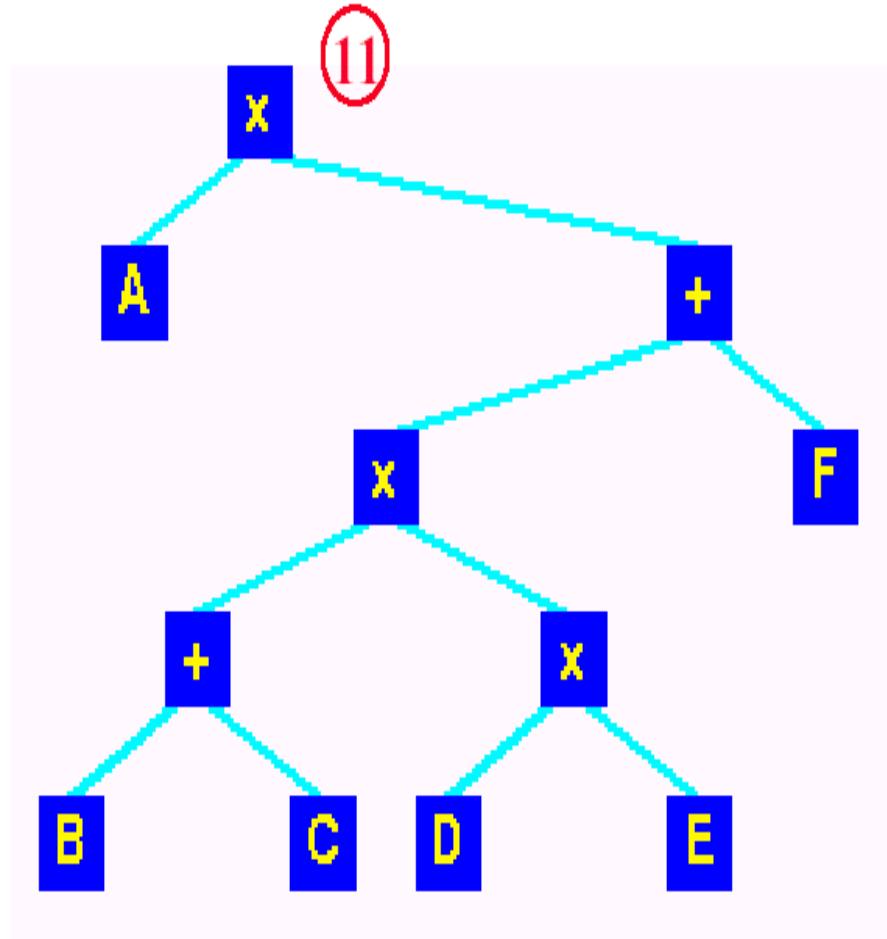
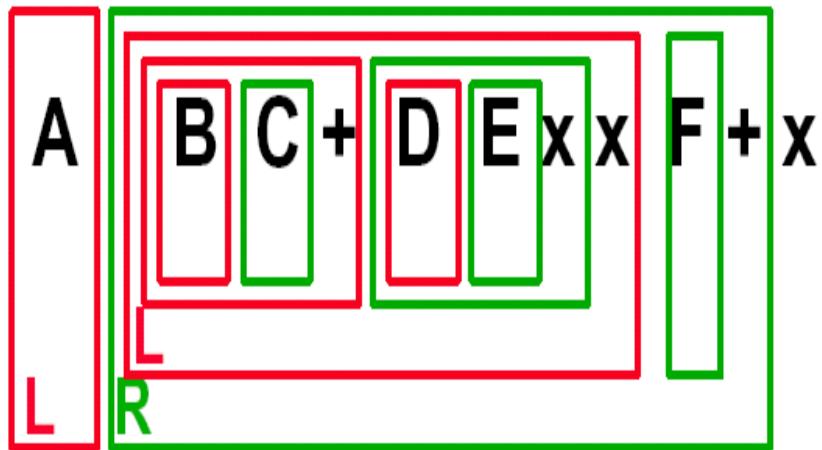
In-order

- Left sub-tree
- Root
- Right sub-tree



Post-order

- Left sub-tree
- Right sub-tree
- Root



Post-order

- Left sub-tree
- Right sub-tree
- Root

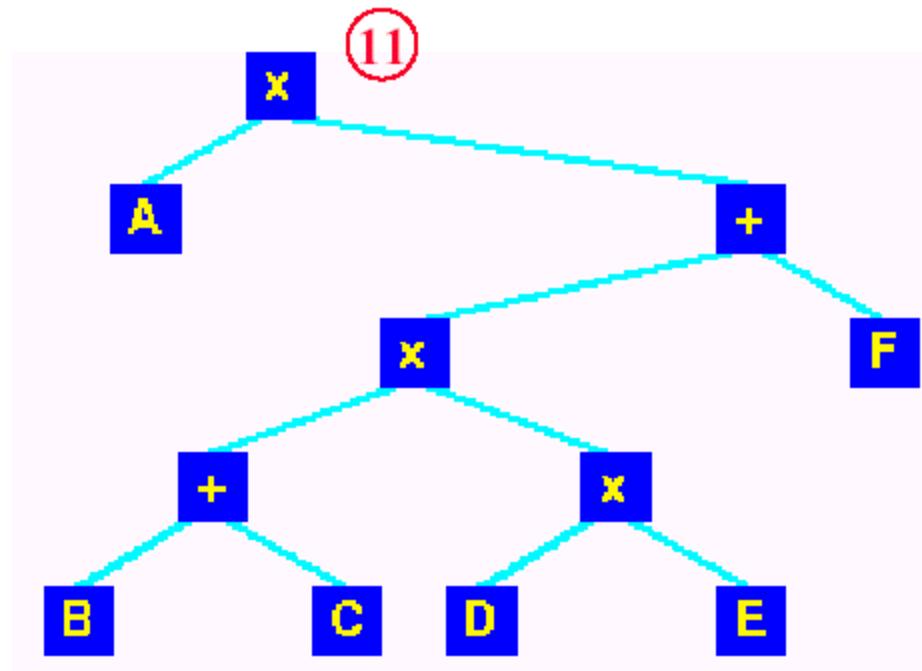
Reverse-Polish

(A (((BC+)(DEx) x) F +)x)

- Normal algebraic form

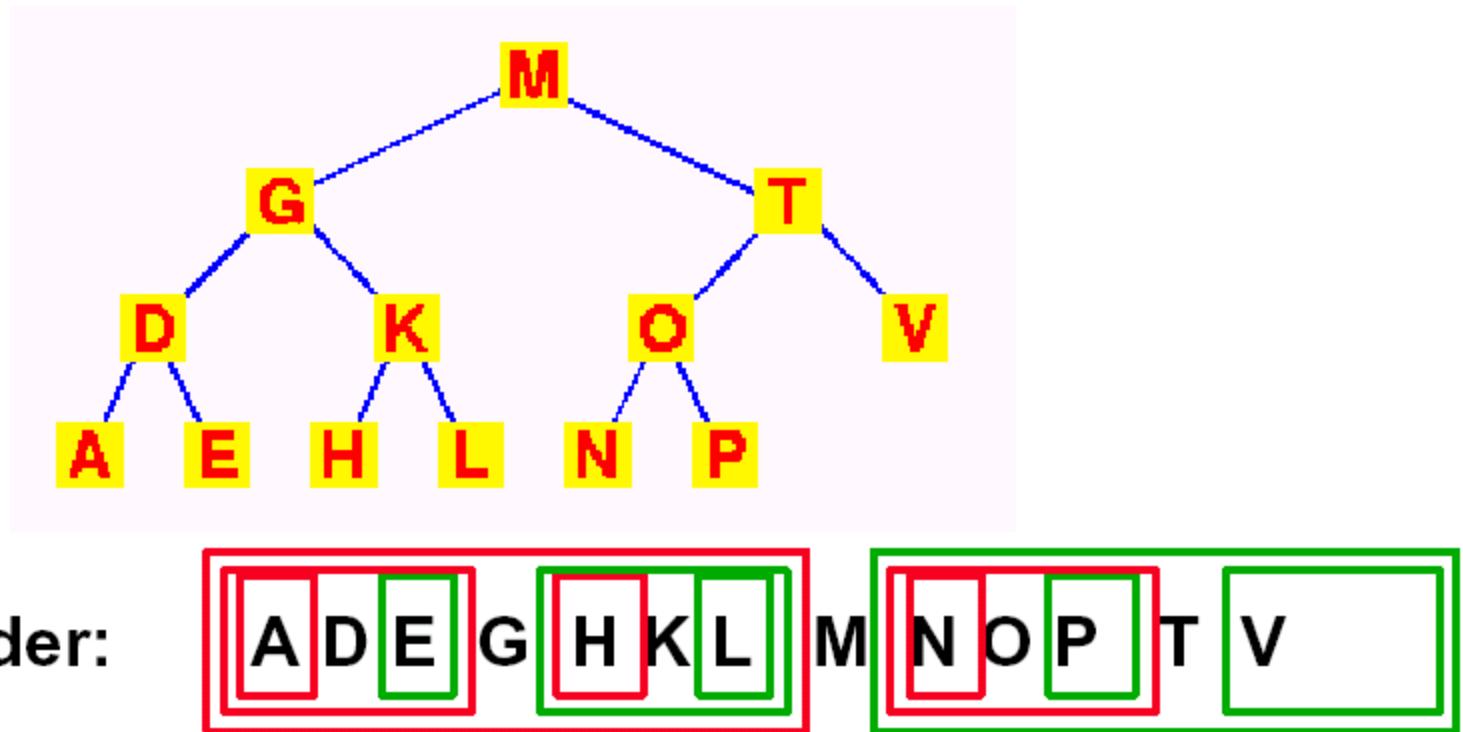
(A x((B+C)(DxE))+F))

= which traversal?



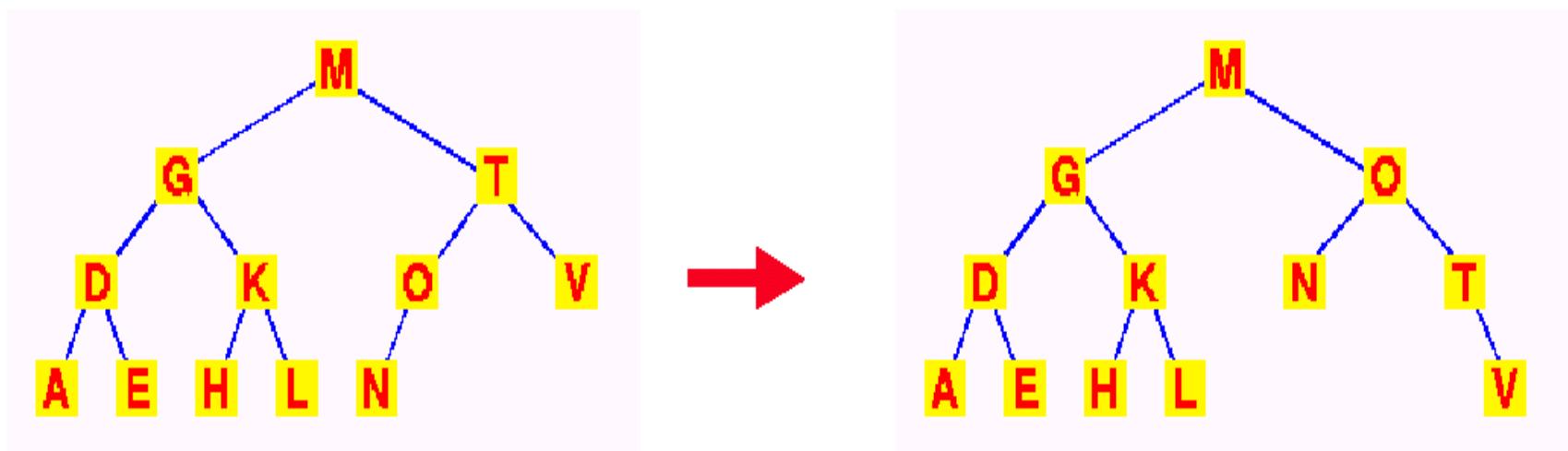
Trees - Searching

- Binary search tree
 - Produces a sorted list by **in-order traversal**



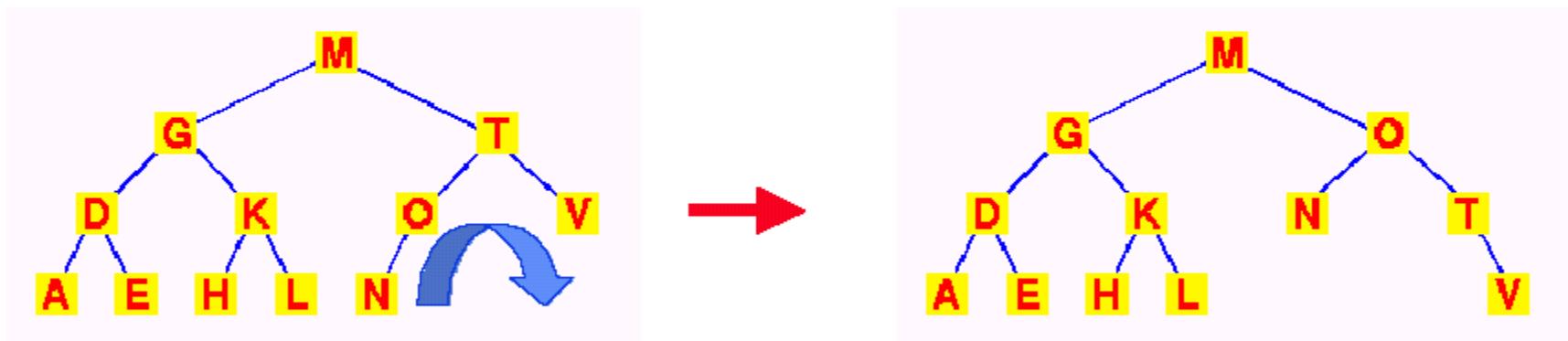
Trees - Searching

- Binary search tree
 - Preserving the order
 - Observe that this transformation preserves the search tree



Trees - Searching

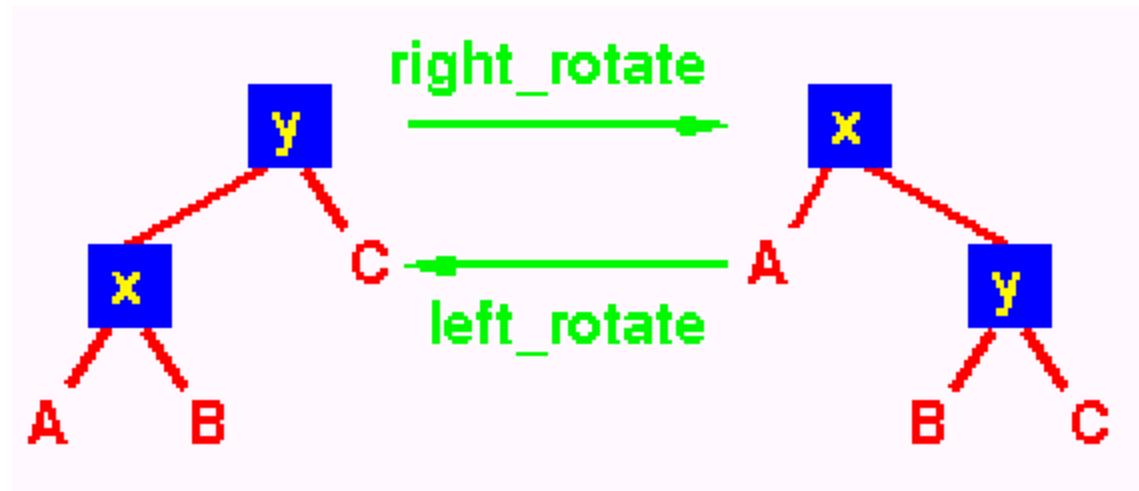
- Binary search tree
 - Preserving the order
 - Observe that this transformation preserves the search tree



- We've performed a **rotation** of the sub-tree about the T and O nodes

Trees - Searching

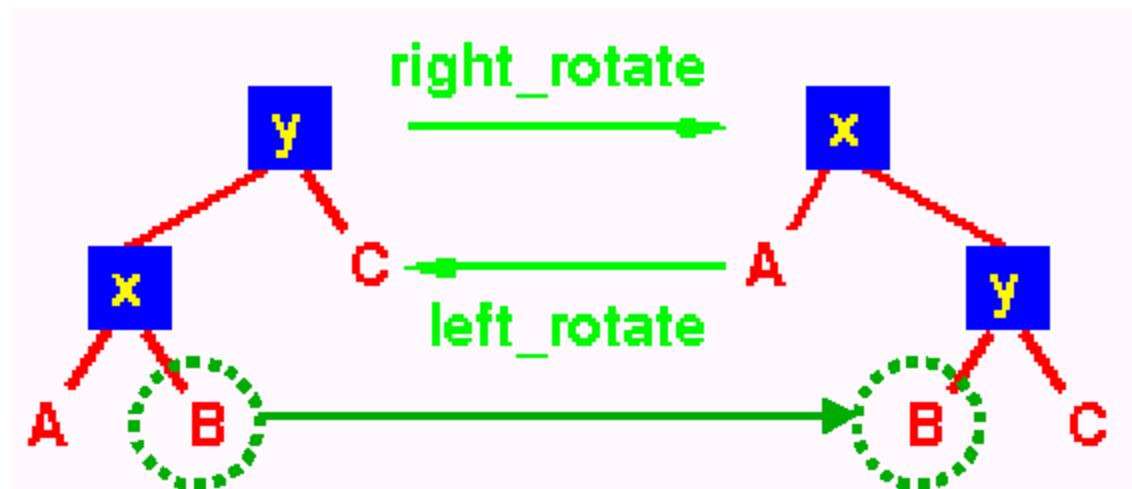
- Binary search tree
 - Rotations can be either left- or right-rotations



- For both trees: the **inorder traversal** is
A x B y C

Trees - Searching

- Binary search tree
 - Rotations can be either left- or right-rotations



- Note that in this rotation, it was necessary to move **B** from the right child of **x** to the left child of **y**

Now it's Time for...

Recurrence Relations

Recurrence Relations

A **recurrence relation** for the sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more of the previous terms of the sequence, namely, a_0, a_1, \dots, a_{n-1} , for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer.

A sequence is called a **solution** of a recurrence relation if its terms satisfy the recurrence relation.

Recurrence Relations

In other words, a recurrence relation is like a recursively defined sequence, but **without specifying any initial values (initial conditions)**.

Therefore, the same recurrence relation can have (and usually has) **multiple solutions**.

If **both** the initial conditions and the recurrence relation are specified, then the sequence is **uniquely determined**.

Recurrence Relations

Example:

Consider the recurrence relation

$$a_n = 2a_{n-1} - a_{n-2} \text{ for } n = 2, 3, 4, \dots$$

Is the sequence $\{a_n\}$ with $a_n=3n$ a solution of this recurrence relation?

For $n \geq 2$ we see that

$$2a_{n-1} - a_{n-2} = 2(3(n-1)) - 3(n-2) = 3n = a_n.$$

Therefore, $\{a_n\}$ with $a_n=3n$ is a solution of the recurrence relation.

Recurrence Relations

Is the sequence $\{a_n\}$ with $a_n=5$ a solution of the same recurrence relation?

For $n \geq 2$ we see that

$$2a_{n-1} - a_{n-2} = 2 \cdot 5 - 5 = 5 = a_n.$$

Therefore, $\{a_n\}$ with $a_n=5$ is also a solution of the recurrence relation.

Modeling with Recurrence Relations

Example:

Someone deposits \$10,000 in a savings account at a bank yielding 5% per year with interest compounded annually. How much money will be in the account after 30 years?

Solution:

Let P_n denote the amount in the account after n years.

How can we determine P_n on the basis of P_{n-1} ?

Modeling with Recurrence Relations

We can derive the following **recurrence relation**:

$$P_n = P_{n-1} + 0.05P_{n-1} = 1.05P_{n-1}$$

The initial condition is $P_0 = 10,000$.

Then we have:

$$P_1 = 1.05P_0$$

$$P_2 = 1.05P_1 = (1.05)^2P_0$$

$$P_3 = 1.05P_2 = (1.05)^3P_0$$

...

$$P_n = 1.05P_{n-1} = (1.05)^nP_0$$

We now have a **formula** to calculate P_n for any natural number n and can avoid the iteration.

Modeling with Recurrence Relations

Let us use this formula to find P_{30} under the initial condition $P_0 = 10,000$:

$$P_{30} = (1.05)^{30} \cdot 10,000 = 43,219.42$$

After 30 years, the account contains \$43,219.42.

Modeling with Recurrence Relations

Another example:

Let a_n denote the number of bit strings of length n that do not have two consecutive 0s ("valid strings"). Find a recurrence relation and give initial conditions for the sequence $\{a_n\}$.

Solution:

Idea: The number of valid strings equals the number of valid strings ending with a 0 plus the number of valid strings ending with a 1.

Modeling with Recurrence Relations

Let us assume that $n \geq 3$, so that the string contains at least 3 bits.

Let us further assume that we know the number a_{n-1} of valid strings of length $(n - 1)$.

Then how many valid strings of length n are there, if the string ends with a 1?

There are a_{n-1} such strings, namely the set of valid strings of length $(n - 1)$ with a 1 appended to them.

Note: Whenever we append a 1 to a valid string, that string remains valid.

Modeling with Recurrence Relations

Now we need to know: How many valid strings of length n are there, if the string ends with a 0?

Valid strings of length n ending with a 0 must have a 1 as their $(n - 1)$ st bit (otherwise they would end with 00 and would not be valid).

And what is the number of valid strings of length $(n - 1)$ that end with a 1?

We already know that there are a_{n-1} strings of length n that end with a 1.

Therefore, there are a_{n-2} strings of length $(n - 1)$ that end with a 1.

Modeling with Recurrence Relations

So there are a_{n-2} valid strings of length n that end with a 0 (all valid strings of length $(n - 2)$ with 10 appended to them).

As we said before, the number of valid strings is the number of valid strings ending with a 0 plus the number of valid strings ending with a 1.

That gives us the following recurrence relation:

$$a_n = a_{n-1} + a_{n-2}$$

Modeling with Recurrence Relations

What are the initial conditions?

$$a_1 = 2 \text{ (0 and 1)}$$

$$a_2 = 3 \text{ (01, 10, and 11)}$$

$$a_3 = a_2 + a_1 = 3 + 2 = 5$$

$$a_4 = a_3 + a_2 = 5 + 3 = 8$$

$$a_5 = a_4 + a_3 = 8 + 5 = 13$$

...

This sequence satisfies the same recurrence relation as the Fibonacci sequence.

Since $a_1 = f_3$ and $a_2 = f_4$, we have $a_n = f_{n+2}$.

Solving Recurrence Relations

In general, we would prefer to have an **explicit formula** to compute the value of a_n rather than conducting n iterations.

For one class of recurrence relations, we can obtain such formulas in a systematic way.

Those are the recurrence relations that express the terms of a sequence as **linear combinations** of previous terms.

Solving Recurrence Relations

Definition: A linear homogeneous recurrence relation of degree k with constant coefficients is a recurrence relation of the form:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k},$$

Where c_1, c_2, \dots, c_k are real numbers, and $c_k \neq 0$.

A sequence satisfying such a recurrence relation is uniquely determined by the recurrence relation and the k initial conditions

$$a_0 = C_0, a_1 = C_1, a_2 = C_2, \dots, a_{k-1} = C_{k-1}.$$

Solving Recurrence Relations

Examples:

The recurrence relation $P_n = (1.05)P_{n-1}$ is a linear homogeneous recurrence relation of **degree one**.

The recurrence relation $f_n = f_{n-1} + f_{n-2}$ is a linear homogeneous recurrence relation of **degree two**.

The recurrence relation $a_n = a_{n-5}$ is a linear homogeneous recurrence relation of **degree five**.

Solving Recurrence Relations

Basically, when solving such recurrence relations, we try to find solutions of the form $a_n = r^n$, where r is a constant.

$a_n = r^n$ is a solution of the recurrence relation

$a_n = c_1a_{n-1} + c_2a_{n-2} + \dots + c_ka_{n-k}$ if and only if

$r^n = c_1r^{n-1} + c_2r^{n-2} + \dots + c_kr^{n-k}$.

Divide this equation by r^{n-k} and subtract the right-hand side from the left:

$$r^k - c_1r^{k-1} - c_2r^{k-2} - \dots - c_{k-1}r - c_k = 0$$

This is called the **characteristic equation** of the recurrence relation.

Solving Recurrence Relations

The solutions of this equation are called the **characteristic roots** of the recurrence relation.

Let us consider linear homogeneous recurrence relations of **degree two**.

Theorem: Let c_1 and c_2 be real numbers. Suppose that $r^2 - c_1r - c_2 = 0$ has two distinct roots r_1 and r_2 . Then the sequence $\{a_n\}$ is a solution of the recurrence relation $a_n = c_1a_{n-1} + c_2a_{n-2}$ if and only if $a_n = \alpha_1r_1^n + \alpha_2r_2^n$ for $n = 0, 1, 2, \dots$, where α_1 and α_2 are constants.

See pp. 321 and 322 for the proof.

Solving Recurrence Relations

Example: What is the solution of the recurrence relation $a_n = a_{n-1} + 2a_{n-2}$ with $a_0 = 2$ and $a_1 = 7$?

Solution: The characteristic equation of the recurrence relation is $r^2 - r - 2 = 0$.

Its roots are $r = 2$ and $r = -1$.

Hence, the sequence $\{a_n\}$ is a solution to the recurrence relation if and only if:

$a_n = \alpha_1 2^n + \alpha_2 (-1)^n$ for some constants α_1 and α_2 .

Solving Recurrence Relations

Given the equation $a_n = \alpha_1 2^n + \alpha_2 (-1)^n$ and the initial conditions $a_0 = 2$ and $a_1 = 7$, it follows that

$$a_0 = 2 = \alpha_1 + \alpha_2$$

$$a_1 = 7 = \alpha_1 \cdot 2 + \alpha_2 \cdot (-1)$$

Solving these two equations gives us

$$\alpha_1 = 3 \text{ and } \alpha_2 = -1.$$

Therefore, the solution to the recurrence relation and initial conditions is the sequence $\{a_n\}$ with

$$a_n = 3 \cdot 2^n - (-1)^n.$$

Solving Recurrence Relations

$a_n = r^n$ is a solution of the linear homogeneous recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

if and only if

$$r^n = c_1 r^{n-1} + c_2 r^{n-2} + \dots + c_k r^{n-k}.$$

Divide this equation by r^{n-k} and subtract the right-hand side from the left:

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \dots - c_{k-1} r - c_k = 0$$

This is called the **characteristic equation** of the recurrence relation.

Solving Recurrence Relations

The solutions of this equation are called the **characteristic roots** of the recurrence relation.

Let us consider linear homogeneous recurrence relations of **degree two**.

Theorem: Let c_1 and c_2 be real numbers. Suppose that $r^2 - c_1r - c_2 = 0$ has two distinct roots r_1 and r_2 . Then the sequence $\{a_n\}$ is a solution of the recurrence relation $a_n = c_1a_{n-1} + c_2a_{n-2}$ if and only if $a_n = \alpha_1r_1^n + \alpha_2r_2^n$ for $n = 0, 1, 2, \dots$, where α_1 and α_2 are constants.

See pp. 321 and 322 for the proof.

Solving Recurrence Relations

Example: Give an explicit formula for the Fibonacci numbers.

Solution: The Fibonacci numbers satisfy the recurrence relation $f_n = f_{n-1} + f_{n-2}$ with initial conditions $f_0 = 0$ and $f_1 = 1$.

The characteristic equation is $r^2 - r - 1 = 0$.

Its roots are

$$r_1 = \frac{1+\sqrt{5}}{2}, \quad r_2 = \frac{1-\sqrt{5}}{2}$$

Solving Recurrence Relations

Therefore, the Fibonacci numbers are given by

$$f_n = \alpha_1 \left(\frac{1+\sqrt{5}}{2} \right)^n + \alpha_2 \left(\frac{1-\sqrt{5}}{2} \right)^n$$

for some constants α_1 and α_2 .

We can determine values for these constants so that the sequence meets the conditions $f_0 = 0$ and $f_1 = 1$:

$$f_0 = \alpha_1 + \alpha_2 = 0$$

$$f_1 = \alpha_1 \left(\frac{1+\sqrt{5}}{2} \right) + \alpha_2 \left(\frac{1-\sqrt{5}}{2} \right) = 1$$

Solving Recurrence Relations

The unique solution to this system of two equations and two variables is

$$\alpha_1 = \frac{1}{\sqrt{5}}, \quad \alpha_2 = -\frac{1}{\sqrt{5}}$$

So finally we obtained an explicit formula for the Fibonacci numbers:

$$f_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

Solving Recurrence Relations

But what happens if the characteristic equation has only one root?

How can we then match our equation with the initial conditions a_0 and a_1 ?

Theorem: Let c_1 and c_2 be real numbers with $c_2 \neq 0$. Suppose that $r^2 - c_1r - c_2 = 0$ has only one root r_0 . A sequence $\{a_n\}$ is a solution of the recurrence relation $a_n = c_1a_{n-1} + c_2a_{n-2}$ if and only if $a_n = \alpha_1r_0^n + \alpha_2nr_0^n$, for $n = 0, 1, 2, \dots$, where α_1 and α_2 are constants.

Solving Recurrence Relations

Example: What is the solution of the recurrence relation $a_n = 6a_{n-1} - 9a_{n-2}$ with $a_0 = 1$ and $a_1 = 6$?

Solution: The only root of $r^2 - 6r + 9 = 0$ is $r_0 = 3$. Hence, the solution to the recurrence relation is $a_n = \alpha_1 3^n + \alpha_2 n 3^n$ for some constants α_1 and α_2 .

To match the initial condition, we need

$$a_0 = 1 = \alpha_1$$

$$a_1 = 6 = \alpha_1 \cdot 3 + \alpha_2 \cdot 3$$

Solving these equations yields $\alpha_1 = 1$ and $\alpha_2 = 1$.

Consequently, the overall solution is given by

$$a_n = 3^n + n 3^n.$$

Solving Recurrence Relations

So what does
 $T(n) = T(n-1) + n$
look like anyway?

Recurrence Relations

- Can easily describe the runtime of recursive algorithms
- Can then be expressed in a closed form (not defined in terms of itself)
- Consider the linear search:

Eg. 1 - Linear Search

- Recursively
- Look at an element (constant work, c), then search the remaining elements...



- $T(n) = T(n-1) + c$
- “The cost of searching n elements is the cost of looking at 1 element, plus the cost of searching $n-1$ elements”

Linear Search (cont)

Caveat:

- You need to convince yourself (and others) that the single step, examining an element, **is** done in constant time.
- Can I get to the i^{th} element in constant time, either directly, or from the $(i-1)^{\text{th}}$ element?
- Look at the code

Methods of Solving Recurrence Relations

- Substitution (we'll work on this one in this lecture)
- Accounting method
- Draw the recursion tree, think about it
- The Master Theorem*
- Guess at an upper bound, prove it

* See Cormen, Leiserson, & Rivest, [Introduction to Algorithms](#)

Linear Search (cont.)

- We'll "unwind" a few of these

$$T(n) = T(n-1) + c \quad (1)$$

But, $T(n-1) = T(n-2) + c$, from above

Substituting back in:

$$T(n) = T(n-2) + c + c$$

Gathering like terms

$$T(n) = T(n-2) + 2c \quad (2)$$

Linear Search (cont.)

- Keep going:

$$T(n) = T(n-2) + 2c$$

$$T(n-2) = T(n-3) + c$$

$$T(n) = T(n-3) + c + 2c$$

$$T(n) = T(n-3) + 3c \tag{3}$$

- One more:

$$T(n) = T(n-4) + 4c \tag{4}$$

Looking for Patterns

- Note, the intermediate results are enumerated
- We need to pull out patterns, to write a general expression for the k^{th} unwinding
 - This requires practise. It is a little bit art. The brain learns patterns, over time. Practise.
- Be careful while simplifying after substitution

Eg. 1 – list of intermediates

| Result at i^{th} unwinding | i |
|---|-----------------------|
| $T(n) = T(n-1) + 1c$ | 1 |
| $T(n) = T(n-2) + 2c$ | 2 |
| $T(n) = T(n-3) + 3c$ | 3 |
| $T(n) = T(n-4) + 4c$ | 4 |

Linear Search (cont.)

- An expression for the k th unwinding:
$$T(n) = T(n-k) + kc$$
- We have 2 variables, k and n , but we have a relation
- $T(d)$ is constant (can be determined) for some constant d (we know the algorithm)
- Choose any convenient # to stop.

Linear Search (cont.)

- Let's decide to stop at $T(0)$. When the list to search is empty, you're done...
- 0 is convenient, in this example...

Let $n-k = 0 \Rightarrow n=k$

- Now, substitute n in everywhere for k :

$$T(n) = T(n-n) + nc$$

$$T(n) = T(0) + nc = nc + c_0 = O(n)$$

($T(0)$ is some constant, c_0)

Binary Search

- Algorithm – “check middle, then search lower $\frac{1}{2}$ or upper $\frac{1}{2}$ ”
- $T(n) = T(n/2) + c$
where c is some constant, the cost of checking the middle...
- Can we really find the middle in constant time? (Make sure.)

Binary Search (cont)

Let's do some quick substitutions:

$$T(n) = T(n/2) + c \quad (1)$$

but $T(n/2) = T(n/4) + c$, so

$$T(n) = T(n/4) + c + c$$

$$T(n) = T(n/4) + 2c \quad (2)$$

$$T(n/4) = T(n/8) + c$$

$$T(n) = T(n/8) + c + 2c$$

$$T(n) = T(n/8) + 3c \quad (3)$$

Binary Search (cont.)

| Result at i^{th} unwinding | i |
|-------------------------------------|-----|
| $T(n) = T(n/2) + c$ | 1 |
| $T(n) = T(n/4) + 2c$ | 2 |
| $T(n) = T(n/8) + 3c$ | 3 |
| $T(n) = T(n/16) + 4c$ | 4 |

Binary Search (cont)

- We need to write an expression for the k^{th} unwinding (in n & k)
 - Must find patterns, changes, as $i=1, 2, \dots, k$
 - This can be the hard part
 - Do not get discouraged! Try something else...
 - We'll re-write those equations...
- We will then need to relate n and k

Binary Search (cont)

| Result at i^{th} unwinding | | | i |
|-------------------------------------|--|-------------------|-----|
| $T(n) = T(n/2) + c$ | | $= T(n/2^1) + 1c$ | 1 |
| $T(n) = T(n/4) + 2c$ | | $= T(n/2^2) + 2c$ | 2 |
| $T(n) = T(n/8) + 3c$ | | $= T(n/2^3) + 3c$ | 3 |
| $T(n) = T(n/16) + 4c$ | | $= T(n/2^4) + 4c$ | 4 |

Binary Search (cont)

- After k unwindings:

$$T(n) = T(n/2^k) + kc$$

- Need a convenient place to stop unwinding – need to relate k & n
- Let's pick $T(0) = c_0$ So,

$$n/2^k = 0 \Rightarrow$$

$$n=0$$

Hmm. Easy, but not real useful...

Binary Search (cont)

- Okay, let's consider $T(1) = c_0$
- So, let:

$$n/2^k = 1 \Rightarrow$$

$$n = 2^k \Rightarrow$$

$$k = \log_2 n = \lg n$$

Binary Search (cont.)

- Substituting back in (getting rid of k):

$$\begin{aligned} T(n) &= T(1) + c \lg(n) \\ &= c \lg(n) + c_0 \\ &= O(\lg(n)) \end{aligned}$$

Recursion

Sections 7.1 and 7.2 of Rosen

Fall 2008

CSCE 235 Introduction to Discrete Structures

Course web-page: cse.unl.edu/~cse235

Questions: cse235@cse.unl.edu

Outline

- Introduction, Motivating Example
- Recurrence Relations
 - Definition, general form, initial conditions, terms
- Linear Homogeneous Recurrences
 - Form, solution, characteristic equation, characteristic polynomial, roots
 - Second order linear homogeneous recurrence
 - Double roots, solution, examples
 - Single root, example
 - General linear homogeneous recurrences: distinct roots, any multiplicity
- Linear Nonhomogenous Recurrences
- Other Methods
 - Backward substitution
 - Recurrence trees
 - Cheating with Maple

Recursive Algorithms

- A recursive algorithm is one in which objects are defined in terms of other objects of the same type
- Advantages:
 - Simplicity of code
 - Easy to understand
- Disadvantages
 - Memory
 - Speed
 - Possibly redundant work
- Tail recursion offers a solution to the memory problem, but really, do we need recursion?

Recursive Algorithms: Analysis

- We have already discussed how to analyze the running time of (iterative) algorithms
- To analyze recursive algorithms, we require more sophisticated techniques
- Specifically, we study how to defined & solve recurrence relations

Motivating Examples: Factorial

- Recall the factorial function:

$$n! = \begin{cases} 1 & \text{if } n=1 \\ n.(n-1) & \text{if } n > 1 \end{cases}$$

- Consider the following (recursive) algorithm for computing $n!$

FACTORIAL

Input: $n \in \mathbb{N}$

Output: $n!$

1. **If** $n=1$
2. **Then Return** 1
3. **Else Return** $n \times \text{FACTORIAL}(n-1)$
4. **Endif**
5. **End**

Factorial: Analysis

How many multiplications $M(x)$ does factorial perform?

- When $n=1$ we don't perform any
- Otherwise, we perform one...
- ... plus how ever many multiplications we perform in the recursive call FACTORIAL($n-1$)
- The number of multiplications can be expressed as a formula (similar to the definition of $n!$)

$$M(0) = 0$$

$$M(n) = 1 + M(n-1)$$

- This relation is known as a recurrence relation

Recurrence Relations

- **Definition:** A recurrence relation for a sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more of the previous terms in the sequence:

$$a_0, a_1, a_2, \dots, a_{n-1}$$

for all integers $n \geq n_0$ where n_0 is a nonnegative integer.

- A sequence is called a solution of a recurrence if its terms satisfy the recurrence relation

Recurrence Relations: Solutions

- Consider the recurrence relation $a_n = 2a_{n-1} - a_{n-2}$
- It has the following sequences a_n as solutions
 - $a_n = 3n$
 - $a_n = n+1$
 - $a_n = 5$
- The initial conditions + recurrence relation
uniquely determine the sequence

Recurrence Relations: Example

- The Fibonacci numbers are defined by the recurrence

$$F(n) = F(n-1) + F(n-2)$$

$$F(1) = 1$$

$$F(0) = 1$$

- The solution to the Fibonacci recurrence is

$$f_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

(The solution is derived in your textbook.)

Outline

- Introduction, Motivating Example
- **Recurrence Relations**
 - **Definition, general form, initial conditions, terms**
- Linear Homogeneous Recurrences
 - Form, solution, characteristic equation, characteristic polynomial, roots
 - Second order linear homogeneous recurrence
 - Double roots, solution, examples
 - Single root, example
 - General linear homogeneous recurrences: distinct roots, any multiplicity
- Linear Nonhomogenous Recurrences
- Other Methods
 - Backward substitution
 - Recurrence trees
 - Cheating with Maple

Recurrence Relations: General Form

- More generally, recurrences can have the form

$$T(n) = \alpha T(n-\beta) + f(n), T(\delta) = c$$

or

$$T(n) = \alpha T(n/\beta) + f(n), T(\delta) = c$$

- Note that it may be necessary to define several $T(\delta)$, which are the initial conditions

Recurrence Relations: Initial Conditions

- The initial conditions specify the value of the first few necessary terms in the sequence.
- In the Fibonacci numbers, we needed two initial conditions: $F(0)=F(1)=1$ since $F(n)$ is defined by the two previous terms in the sequence
- Initial conditions are also known as boundary conditions (as opposed to general conditions)
- From now on, we will use the subscript notation, so the Fibonacci numbers are:

$$f_n = f_{n-1} + f_{n-2}$$

$$f_1 = 1$$

$$f_0 = 1$$

Recurrence Relations: Terms

- Recurrence relations have two parts: **recursive terms** and **non-recursive terms**

$$T(n) = 2T(n-2) + n^2 - 10$$

- **Recursive terms** come from when an algorithm calls itself
- **Non-recursive terms** correspond to the non-recursive cost of the algorithm: work the algorithm performs within a function
- We will see examples later. First, we need to know how to solve recurrences.

Solving Recurrences

- There are several methods for solving recurrences
 - Characteristic Equations
 - Forward Substitution
 - Backward Substitution
 - Recurrence Trees
 - ... Maple!

Outline

- Introduction, Motivating Example
- Recurrence Relations
 - Definition, general form, initial conditions, terms
- **Linear Homogeneous Recurrences**
 - **Form, solution, characteristic equation, characteristic polynomial, roots**
 - **Second order linear homogeneous recurrence**
 - Double roots, solution, examples
 - Single root, example
 - **General linear homogeneous recurrences: distinct roots, any multiplicity**
- Linear Nonhomogenous Recurrences
- Other Methods
 - Backward substitution
 - Recurrence trees
 - Cheating with Maple

Linear Homogeneous Recurrences

- **Definition:** A linear homogeneous recurrence relation of degree k with constant coefficients is a recurrence relation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

with $c_1, c_2, \dots, c_k \in \mathbb{R}$, $c_k \neq 0$.

- Linear: RHS is a sum of multiples of previous terms of the sequence (linear combination of previous terms). The coefficients are all constants (not functions depending on n)
- Homogeneous: no terms occur that are not multiples of a_j 's
- Degree k : a_n is expressed in terms of k terms of the sequences

Linear Homogeneous Recurrences: Examples

- The Fibonacci sequence is a linear homogeneous recurrence relation
- So are the following relations:

$$a_n = 4a_{n-1} + 5a_{n-2} + 7a_{n-3}$$

$$a_n = 2a_{n-2} + 4a_{n-4} + 8a_{n-8}$$

How many initial conditions do we need to specify for these relations?

As many as the degree k : $k=3, 8$ respectively

- So, how do solve linear homogeneous recurrences?

Solving Linear Homogeneous Recurrences

- We want a solution of the form $a_n = r^n$ where r is some real constant
- We observe that $a_n = r^n$ is a solution to a linear homogeneous recurrence *if and only if*

$$r^n = c_1 r^{n-1} + c_2 r^{n-2} + \dots + c_k r^{n-k}$$

- We can now divide both sides by r^{n-k} , collect terms and we get a k -degree polynomial

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \dots - c_k = 0$$

- This equation is called the characteristic equation of the recurrence relation
- The roots of this polynomial are called the characteristics roots of the recurrence relation. They can be used to find the solutions (if they exist) to the recurrence relation. We will consider several cases.

Second Order Linear Homogeneous Recurrences

- A second order linear homogeneous recurrence is a recurrence of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2}$$

- **Theorem** (Theorem 1, page 462): Let $c_1, c_2 \in \mathbb{R}$ and suppose that $r^2 - c_1 r - c_2 = 0$ is the characteristic polynomial of a 2nd order linear homogeneous recurrence which has two distinct* roots r_1, r_2 , then $\{a_n\}$ is a solution if and only if

$$a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$$

for $n=0,1,2,\dots$ where α_1, α_2 are constants dependent upon the initial conditions

* We discuss single root later

Second Order Linear Homogeneous Recurrences: Example A (1)

- Find a solution to

$$a_n = 5a_{n-1} - 6a_{n-2}$$

with initial conditions $a_0=1, a_1=4$

- The characteristic equation is

$$r^2 - 5r + 6 = 0$$

- The roots are $r_1=2, r_2=3$

$$r^2 - 5r + 6 = (r-2)(r-3)$$

- Using the 2nd order theorem we have a solution

$$a_n = \alpha_1 2^n + \alpha_2 3^n$$

Second Order Linear Homogeneous Recurrences: Example A (2)

- Given the solution

$$a_n = \alpha_1 2^n + \alpha_2 3^n$$

- We plug in the two initial conditions to get a system of linear equations

$$a_0 = \alpha_1 2^0 + \alpha_2 3^0$$

$$a_1 = \alpha_1 2^1 + \alpha_2 3^1$$

- Thus:

$$1 = \alpha_1 + \alpha_2$$

$$4 = 2\alpha_1 + 3\alpha_2$$

Second Order Linear Homogeneous Recurrences: Example A (3)

$$1 = \alpha_1 + \alpha_2$$

$$4 = 2\alpha_1 + 3\alpha_2$$

- Solving for $\alpha_1 = (1 - \alpha_2)$, we get

$$4 = 2\alpha_1 + 3\alpha_2$$

$$4 = 2(1 - \alpha_2) + 3\alpha_2$$

$$4 = 2 - 2\alpha_2 + 3\alpha_2$$

$$2 = \alpha_2$$

- Substituting for α_1 : $\alpha_1 = -1$
- Putting it back together, we have

$$a_n = \alpha_1 2^n + \alpha_2 3^n$$

$$a_n = -1 \cdot 2^n + 2 \cdot 3^n$$

Second Order Linear Homogeneous Recurrences: Example B (1)

- Solve the recurrence

$$a_n = -2a_{n-1} + 15a_{n-2}$$

with initial conditions $a_0 = 0, a_1 = 1$

- If we did it right, we have

$$a_n = \frac{1}{8}(3)^n - \frac{1}{8}(-5)^n$$

- To check ourselves, we verify a_0, a_1 , we compute a_3 with both equations, then maybe a_4 , etc.

Single Root Case

- We can apply the theorem if the roots are distincts, i.e. $r_1 \neq r_2$
- If the roots are not distinct ($r_1 = r_2$), we say that one characteristic root has multiplicity two. In this case, we apply a different theorem
- **Theorem** (Theorem 2, page 464)

Let $c_1, c_2 \in \mathbb{R}$ and suppose that $r^2 - c_1r - c_2 = 0$ has only one distinct root, r_0 , then $\{a_n\}$ is a solution to $a_n = c_1a_{n-1} + c_2a_{n-2}$ if and only if

$$a_n = \alpha_1 r_0^n + \alpha_2 n r_0^n$$

for $n=0,1,2,\dots$ where α_1, α_2 are constants depending upon the initial conditions

Single Root Case: Example (1)

- What is the solution to the recurrence relation

$$a_n = 8a_{n-1} - 16a_{n-2}$$

with initial conditions $a_0= 1$, $a_1= 7$?

- The characteristic equation is:

$$r^2 - 8r + 16 = 0$$

- Factoring gives us:

$$r^2 - 8r + 16 = (r-4)(r-4), \text{ so } r_0=4$$

- Applying the theorem we have the solution:

$$a_n = \alpha_1(4)^n + \alpha_2 n(4)^n$$

Single Root Case: Example (2)

- Given: $a_n = \alpha_1(4)^n + \alpha_2 n(4)^n$
- Using the initial conditions, we get:

$$a_0 = 1 = \alpha_1(4)^0 + \alpha_2 0(4)^0 = \alpha_1$$

$$a_1 = 7 = \alpha_1(4) + \alpha_2 1(4)^1 = 4\alpha_1 + 4\alpha_2$$

- Thus: $\alpha_1 = 1, \alpha_2 = 3/4$
- The solution is

$$a_n = (4)^n + \frac{3}{4} n (4)^n$$

- Always check yourself...

General Linear Homogeneous Recurrences

- There is a straightforward generalization of these cases to higher-order linear homogeneous recurrences
- Essentially, we simply define higher degree polynomials
- The roots of these polynomials lead to a general solution
- The general solution contains coefficients that depend only on the initial conditions
- In the general case, the coefficients form a system of linear equalities

General Linear Homogeneous Recurrences: Distinct Roots

- **Theorem** (Theorem 3, page 465)

Let $c_1, c_2, \dots, c_k \in \mathbb{R}$ and suppose that the characteristic equation

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \dots - c_k = 0$$

has k distinct roots r_1, r_2, \dots, r_k . Then a sequence $\{a_n\}$ is a solution of the recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

if and only if

$$a_n = \alpha_1 r_1^n + \alpha_2 r_2^n + \dots + \alpha_k r_k^n$$

for $n=0,1,2,\dots$ where $\alpha_1, \alpha_2, \dots, \alpha_k$ are constants depending upon the initial conditions

General Linear Homogeneous Recurrences: Any Multiplicity

- **Theorem** (Theorem 3, page 465)

Let $c_1, c_2, \dots, c_k \in R$ and suppose that the characteristic equation

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \dots - c_k = 0$$

has t roots with multiplicities m_1, m_2, \dots, m_t . Then a sequence $\{a_n\}$ is a solution of the recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

if and only if $a_n = (\alpha_{1,0} + \alpha_{1,1}n + \dots + \alpha_{1,m_1-1}n^{m_1-1}) r_1^n +$
 $(\alpha_{2,0} + \alpha_{2,1}n + \dots + \alpha_{2,m_2-1}n^{m_2-1}) r_2^n + \dots$
 $(\alpha_{t,0} + \alpha_{t,1}n + \dots + \alpha_{t,m_t-1}n^{m_t-1}) r_t^n$

for $n=0,1,2,\dots$ where $\alpha_{i,j}$ are constants for $1 \leq i \leq t$ and
 $0 \leq j \leq m_i-1$ depending upon the initial conditions

Outline

- Introduction, Motivating Example
- Recurrence Relations
 - Definition, general form, initial conditions, terms
- Linear Homogeneous Recurrences
 - Form, solution, characteristic equation, characteristic polynomial, roots
 - Second order linear homogeneous recurrence
 - Double roots, solution, examples
 - Single root, example
 - General linear homogeneous recurrences: distinct roots, any multiplicity
- **Linear Nonhomogenous Recurrences**
- Other Methods
 - Backward substitution
 - Recurrence trees
 - Cheating with Maple

Linear NonHomogeneous Recurrences

- For recursive algorithms, cost function are often not homogeneous because there is usually a non-recursive cost depending on the input size
- Such a recurrence relation is called a linear nonhomogeneous recurrence relation
- Such functions are of the form
$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} + f(n)$$
- $f(n)$ represents a non-recursive cost. If we chop it off, we are left with
$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$
which is the associated homogeneous recurrence relation
- Every solution of a linear nonhomogeneous recurrence relation is the sum of
 - a particular relation and
 - a solution to the associated linear homogeneous recurrence relation

Solving Linear NonHomogeneous Recurrences (1)

- **Theorem** (Theorem 5, p468)

If $\{a_n^{(p)}\}$ is a particular solution of the nonhomogeneous linear recurrence relation with constant coefficients

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} + f(n)$$

then every solution is of the form $\{a_n^{(p)} + a_n^{(h)}\}$ where $\{a_n^{(h)}\}$ is a solution of the associated homogeneous recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

Solving Linear NonHomogeneous Recurrences (2)

- There is no general method for solving such relations.
- However, we can solve them for special cases
- In particular, if $f(n)$ is
 - a polynomial function
 - exponential function, or
 - the product of a polynomial and exponential functions,

then there is a general solution

Solving Linear NonHomogeneous Recurrences (3)

- **Theorem** (Theorem 6, p469)

Suppose $\{a_n\}$ satisfies the linear nonhomogeneous recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} + f(n)$$

where $c_1, c_2, \dots, c_k \in R$ and

$$f(n) = (b_t n^t + b_{t-1} n^{t-1} + \dots + b_1 n + b_0) s^n$$

where $b_0, b_1, \dots, b_n, s \in R$

... continues

Solving Linear NonHomogeneous Recurrences (4)

- **Theorem** (Theorem 6, p469)... continued

When s is not a root of the characteristic equation of the associated linear homogeneous recurrence relation, there is a particular solution of the form

$$(p_t n^t + p_{t-1} n^{t-1} + \dots + p_1 n + p_0) s^n$$

When s is a root of this characteristic equation and its multiplicity is m, there is a particular solution of the form

$$n^m (p_t n^t + p_{t-1} n^{t-1} + \dots + p_1 n + p_0) s^n$$

Linear NonHomogeneous Recurrences: Examples

- The examples in the textbook are quite good (see pp467–470) and illustrate how to solve simple nonhomogeneous relations
- We may go over more examples if time allows
- Also read up on generating functions in Section 7.4 (though we may return to this subject)
- However, there are alternate, more intuitive methods

Outline

- Introduction, Motivating Example
- Recurrence Relations
 - Definition, general form, initial conditions, terms
- Linear Homogeneous Recurrences
 - Form, solution, characteristic equation, characteristic polynomial, roots
 - Second order linear homogeneous recurrence
 - Double roots, solution, examples
 - Single root, example
 - General linear homogeneous recurrences: distinct roots, any multiplicity
- Linear Nonhomogenous Recurrences
- **Other Methods**
 - **Backward substitution**
 - **Recurrence trees**
 - **Cheating with Maple**

Other Methods

- When analyzing algorithms, linear homogeneous recurrences of order greater than 2 hardly ever arise in practice
- We briefly describe two unfolding methods that work for a lot of cases
 - Backward substitution: this works exactly as its name suggests. Starting from the equation itself, work backwards, substituting values of the function for previous ones
 - Recurrence trees: just as powerful, but perhaps more intuitive, this method involves mapping out the recurrence tree for an equation. Starting from the equation, you unfold each recursive call to the function and calculate the non-recursive cost at each level of the tree. Then, you find a general formula for each level and take a summation over all such levels

Backward Substitution: Example (1)

- Give a solution to

$$T(n) = T(n-1) + 2n$$

where $T(1)=5$

- We begin by unfolding the recursion by a simple substitution of the function values
- We observe that

$$T(n-1) = T((n-1)-1) + 2(n-1) = T(n-2) + 2(n-1)$$

- Substituting into the original equation

$$T(n) = T(n-2) + 2(n-1) + 2n$$

Backward Substitution: Example (2)

- If we continue to do that we get

$$T(n) = T(n-2) + 2(n-1) + 2n$$

$$T(n) = T(n-3) + 2(n-2) + 2(n-1) + 2n$$

$$T(n) = T(n-4) + 2(n-3) + 2(n-2) + 2(n-1) + 2n$$

.....

$$T(n) = T(n-i) + \sum_{j=0}^{i-1} 2(n - j) \quad \text{function's value at the } i^{\text{th}} \text{ iteration}$$

- Solving the sum we get

$$T(n) = T(n-i) + 2n(i-1) - 2(i-1)(i-1+1)/2 + 2n$$

$$T(n) = T(n-i) + 2n(i-1) - i^2 + i + 2n$$

Backward Substitution: Example (3)

- We want to get rid of the recursive term
- To do that, we need to know at what iteration we reach our base case, i.e. for what value of i can we use the initial condition $T(1)=5$?
- We get the base case when $n-i=1$ or $i=n-1$
- Substituting in the equation above we get

$$T(n) = 5 + 2n(n-1-1) - (n-1)^2 + (n-1) + 2n$$

$$T(n) = 5 + 2n(n-2) - (n^2-2n+1) + (n-1) + 2n = n^2 + n + 3$$

Recurrence Trees (1)

- When using recurrence trees, we graphically represent the recursion
- Each node in the tree is an instance of the function. As we progress downward, the size of the input decreases
- The contribution of each level to the function is equivalent to the number of nodes at that level times the non-recursive cost on the size of the input at that level
- The tree ends at the depth at which we reach the base case
- As an example, we consider a recursive function of the form

$$T(n) = \alpha T(n/\beta) + f(n), \quad T(\delta) = c$$

Recurrence Trees (2)

Iteration

0

$T(n)$

1

$T(n/\beta)$

$T(n/\beta)$

$\dots \alpha \dots$

$T(n/\beta)$

Cost
 $f(n)$

$\alpha f(n/\beta)$

2

$T(n/\beta^2)$

$\dots \alpha \dots$

$T(n/\beta^2)$

$T(n/\beta^2)$

$\dots \alpha \dots$

$T(n/\beta^2)$

$\alpha^2 f(n/\beta^2)$

i

$\alpha^i f(n/\beta^i)$

$\log_\beta n$

$\alpha^{\log_\beta n} \cdot T(\delta)$

Recurrence Trees (3)

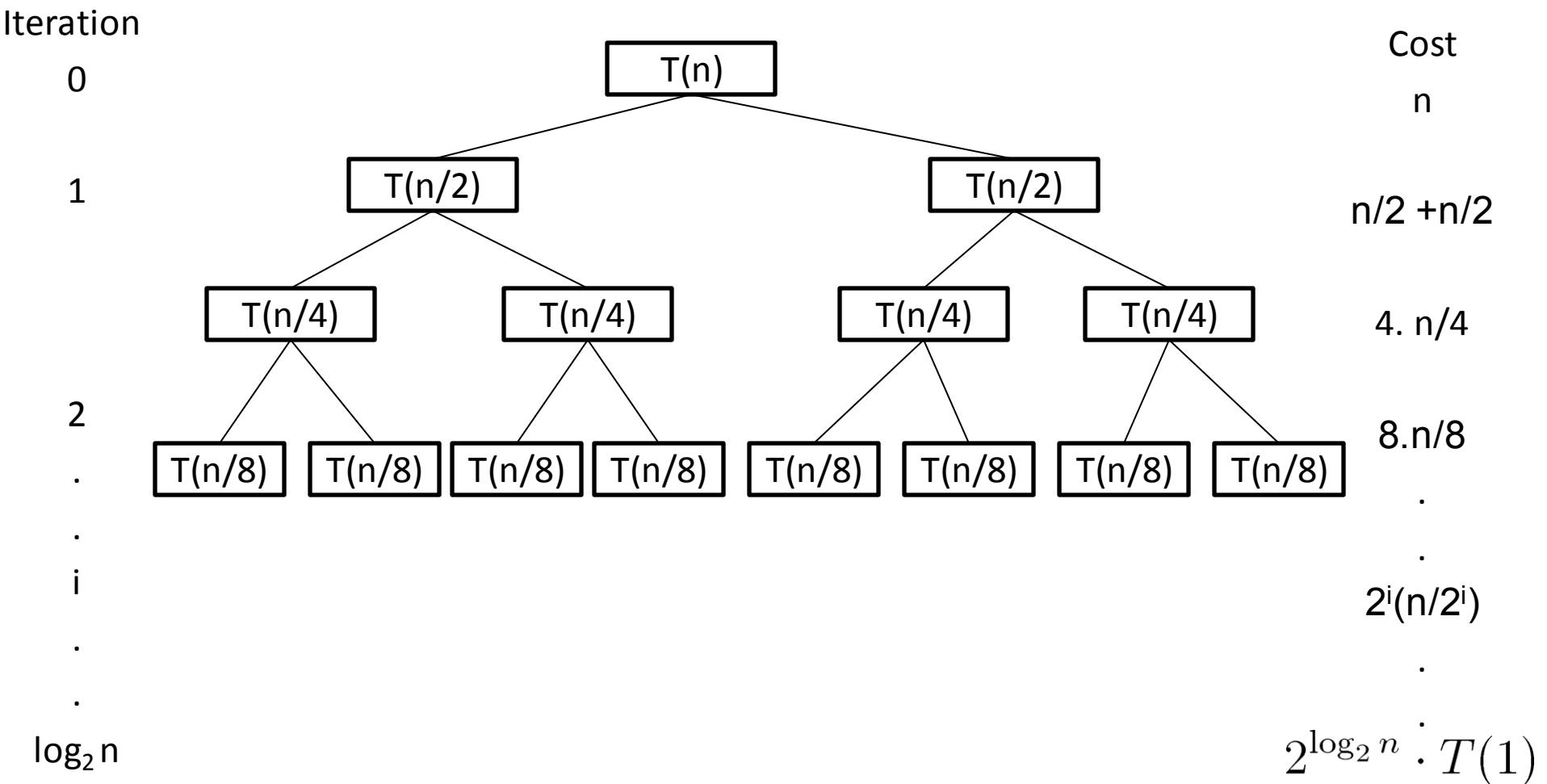
- The total value of the function is the summation over all levels of the tree

$$T(n) = \sum_{i=0}^{\log_\beta n} \alpha^i f(n/\beta^i)$$

- Consider the following concrete example

$$T(n) = 2T(n/2) + n, \quad T(1)=4$$

Recurrence Tree: Example (2)



Recurrence Trees: Example (3)

- The value of the function is the summation of the value of all levels.
- We treat the last level as a special case since its non-recursive cost is different

$$T(n) = 4n + \sum_{i=0}^{(\log_2 n - 1)} 2^i \frac{n}{2^i} = n(\log n) + 4n$$

Smoothness Rule

- In the previous example, we make the following assumption
 n has a power of two ($n=2^k$)
This assumption is necessary to get a nice depth of $\log(n)$ and a full tree
- We can restrict consideration to certain powers because of the smoothness rule, which is not studied in this course.
- For more information about that rule, consult pages 481—483 of the textbook “The Design & Analysis of Algorithms” by Anany Levitin

How to Cheat with Maple (1)

- Maple and other math tools are great resources. However, they are no substitutes for knowing how to solve recurrences yourself
- As such, you should only use Maple to check your answers
- Recurrence relations can be solved using the `rsolve` command and giving Maple the proper parameters
- The arguments are essentially a comma-delimited list of equations
 - General and boundary conditions
 - Followed by the ‘name’ and variables of the function

How to Cheat with Maple (2)

```
> rsolve({T(n)= T(n-1)+2*n, T(1)=5}, T(n));  
1+2 (n+1) (1/2 n+1)-2 n
```

- You can clean up Maple's answer a bit by encapsulating it in the `simplify` command

```
> simplify(rsolve({T(n)= T(n-1) + 2*n, T(1) = 5},  
T(n)));  
3 + n2 + n
```

Summary

- Introduction, Motivating Example
- Recurrence Relations
 - Definition, general form, initial conditions, terms
- Linear Homogeneous Recurrences
 - Form, solution, characteristic equation, characteristic polynomial, roots
 - Second order linear homogeneous recurrence
 - Double roots, solution, examples
 - Single root, example
 - General linear homogeneous recurrences: distinct roots, any multiplicity
- Linear Nonhomogenous Recurrences
- Other Methods
 - Backward substitution
 - Recurrence trees
 - Cheating with Maple