

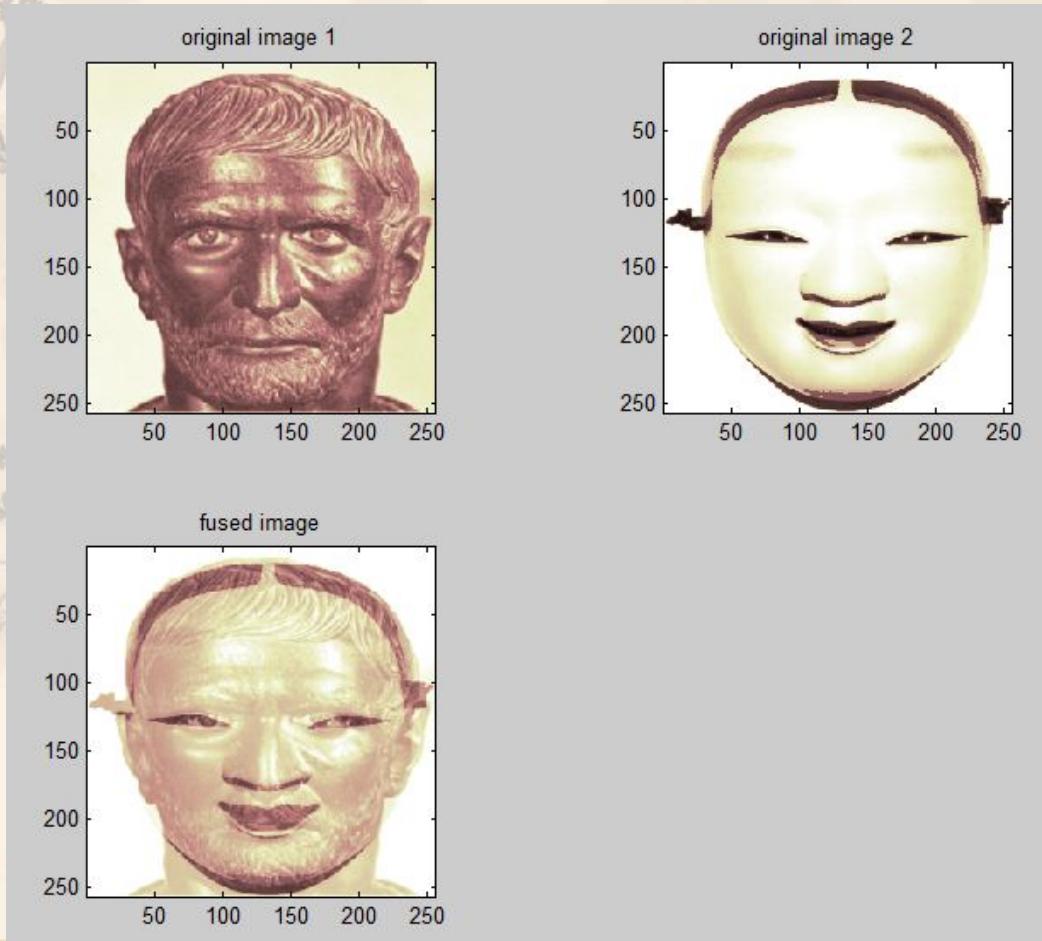
LECTURE ONE

INTRODUCTION TO ALGORITHM

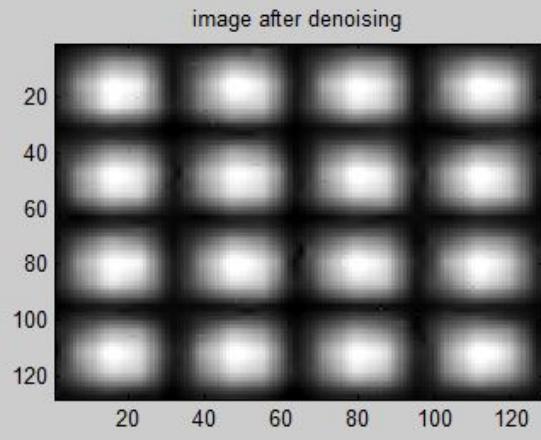
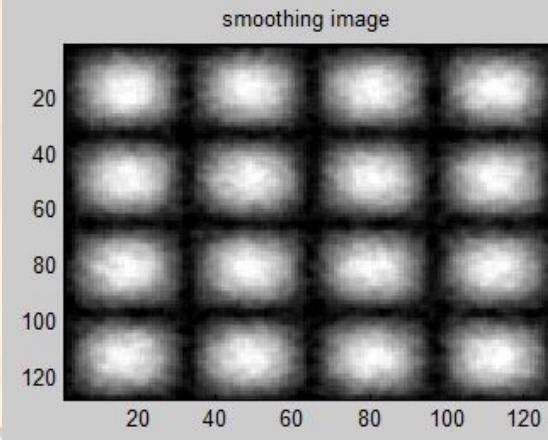
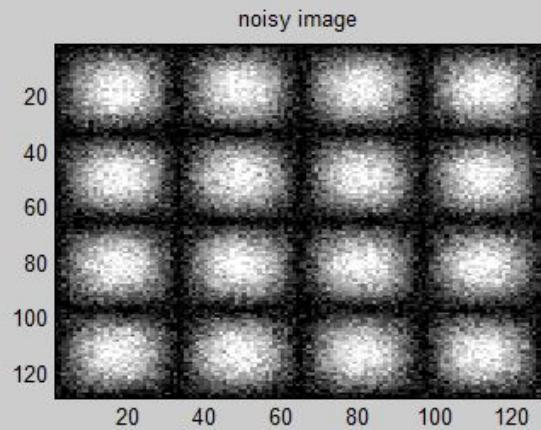
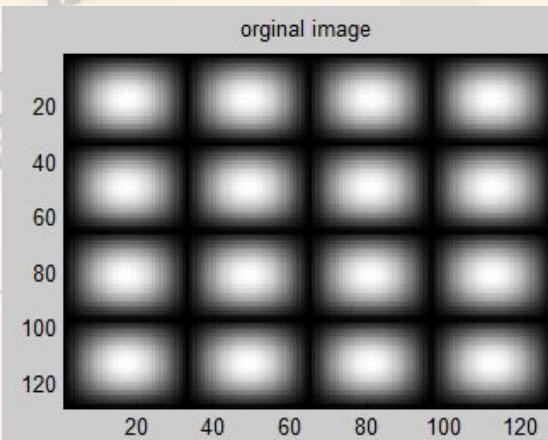
Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

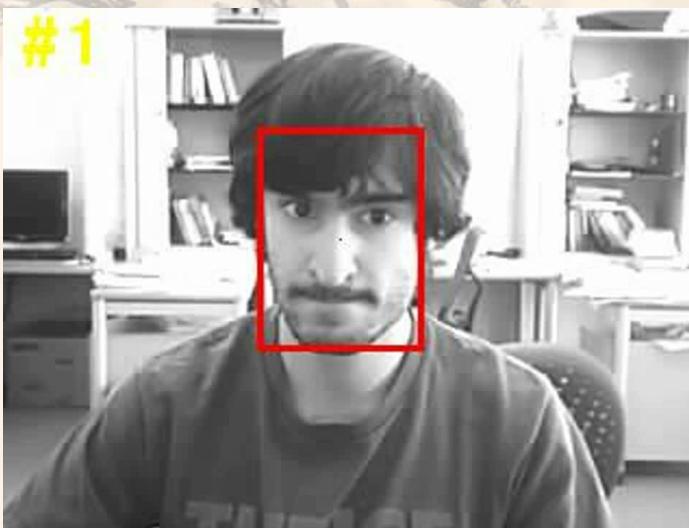
Some examples: fusion



Some examples: denoising



Some examples: object tracking

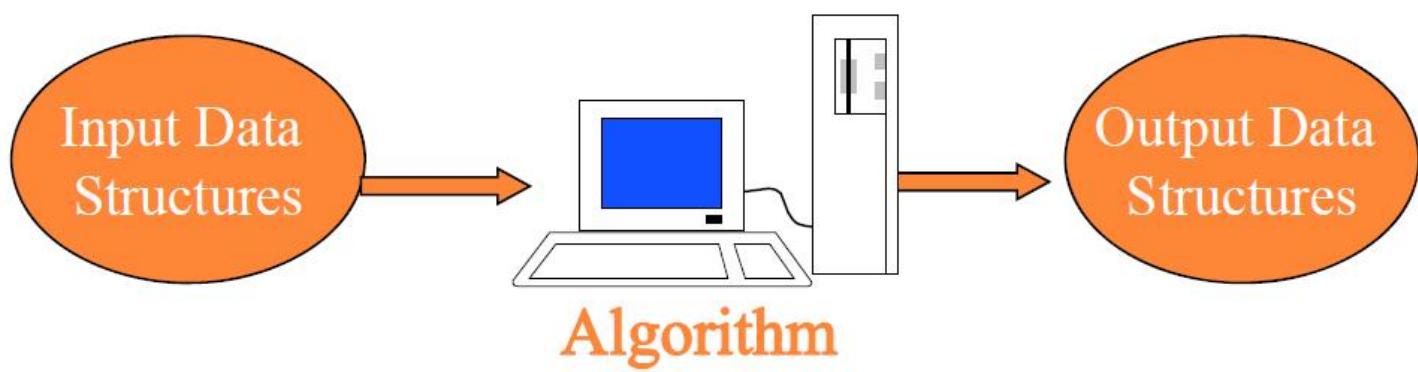
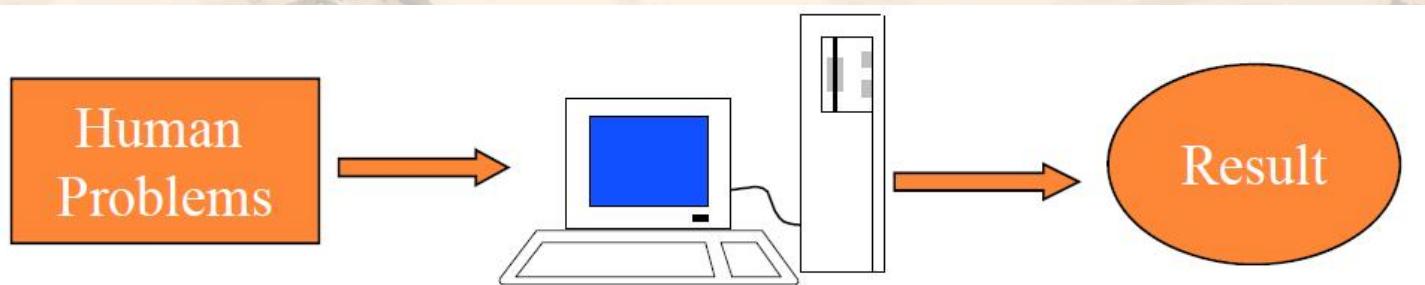


Algorithms

- What?
- Why?
- How?

What are algorithms?

An algorithm is a sequence of computational steps that transform the input into the output.



One example: sorting problem

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$

Output: A permutation (recording) sequence $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that

$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

Human Problems

- Path searching

Input: a road map on which the distance between each pair of adjacent intersections is marked

Output: the shortest route from one intersection to another

- Human genome project

Input: 100,000 genes in human DNA.

Output: sequences of 3 billion chemical base pairs.

- Internet information management

Why?-The Reasons to Study Algorithm

- Become the skilled programmer from the novice.
- Cookbook
- Master the skills to analysis and design algorithms with high efficiency for new problems

Efficiency

Input: -1, 1, -1, 1, ..., $(-1)^n$

Output: the sum of this sequences.

What do you think of this question?

Algorithm 1

```
sum = 0;  
for (i=1; i<=n; i++)  
{  
    sum = sum+(-1)^n;  
}  
Execute n+1 times.
```

Algorithm 2

```
if(n%2==0)  
    sum=0;  
else  
    sum=-1;  
Execute 1 time
```

Efficiency

- Baseline to evaluate the algorithm

- Inserting sort

Cost time: $C_1 n^2$

$$C_1 < C_2$$

$$C_1 n^2 \leq C_2 n \lg n ?$$

- Merge sort

Cost time: $C_2 n \lg n$

● References

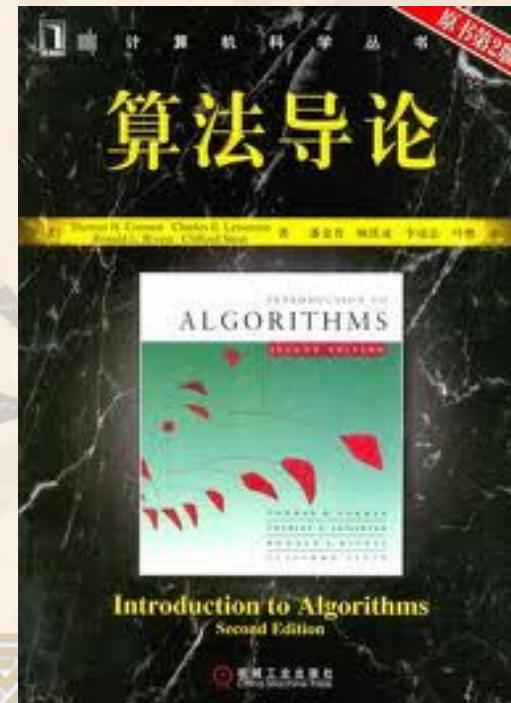
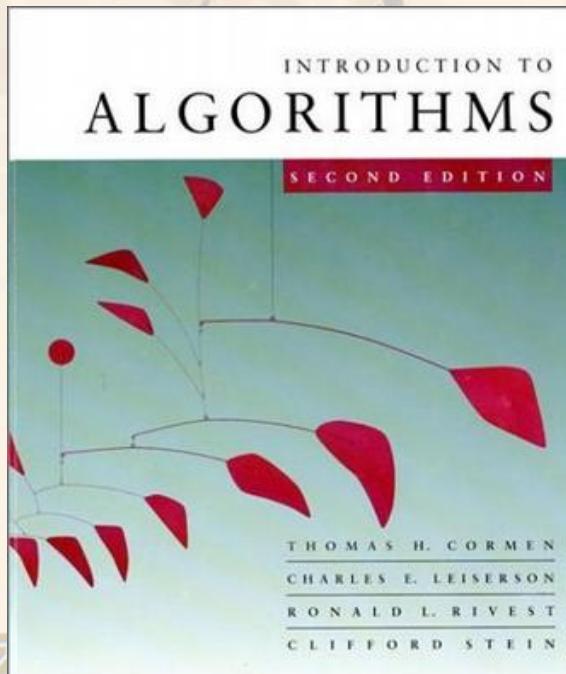
- 计算机算法设计与分析(第2版)
王晓东编著



The Trivial

- Text book:
 - Introduction to Algorithm (Second Edition)

Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein



- Prerequisites
 - Knowledge of Data Structure
 - A programming Language: C, Java, ..., etc.

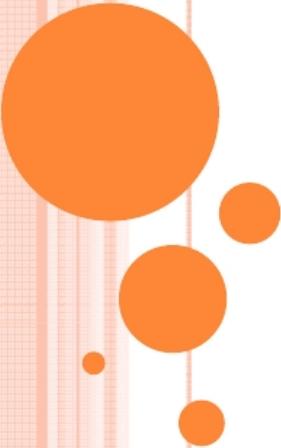
Schedule

- Introduction to Algorithm
- Recurrences and Divide-and-Conquer
- Quicksort & Sorting in Linear Time
- Elementary Data Structures & Hash Function
- Binary Search Tree & Red Black Tree & B-Tree
- Augmenting Data Structure (Interval Tree)
- Dynamic Programming
- Greedy Algorithm
- Linear Programming
- ...

Grading

- Class participation 5%
- Project & Homework 35%
- Exam 60%

RECURRENCE & DIVIDE-AND-CONQUER



Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

OUTLINE

Sort Example and Asymptotic Analysis

Recurrence and Divide-and-Conquer

Three recurrence solving methods

Substitution method

Recursion-tree method

Master method

Divide-and-Conquer example

Big Integer Multiplication

Strassen Matrix Multiplication

Chessboard Cover

Order Statistic



SORT EXAMPLE AND ASYMPTOTIC ANALYSIS

SORTING PROBLEM

Description:

Input: sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that
 $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

INSERTION SORT

Pseudocode

InsertionSort (A, n)

for $j \leftarrow 2$ **to** n

do

Insert $A[j]$ into the sorted sequence

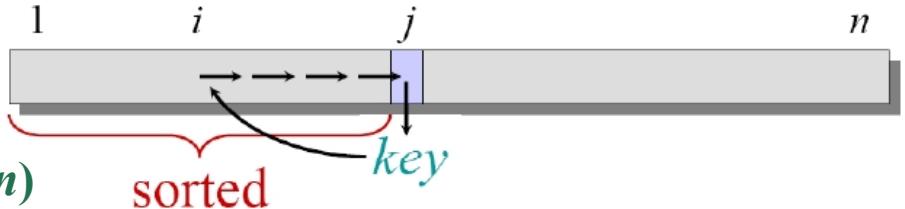
$A[1 .. j - 1]$.



INSERTION SORT

Pseudocode

InsertionSort (A, n)



```
for  $j \leftarrow 2$  to  $n$ 
do
    key  $\leftarrow A[j]$ 
    //Insert  $A[j]$  into the sorted sequence  $A[1 .. j - 1]$ .
     $i \leftarrow j - 1$ 
    while  $i > 0$  and  $A[i] > \text{key}$ 
    do
         $A[i + 1] \leftarrow A[i]$ 
         $i \leftarrow i - 1$ 
     $A[i + 1] \leftarrow \text{key}$ 
```



INSERTION SORT

Example:

8

2

4

9

3

6

INSERTION SORT

Example:



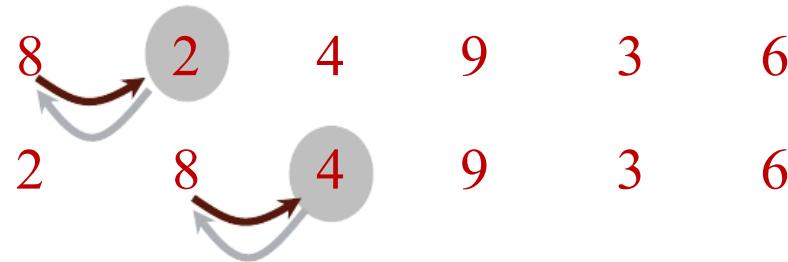
INSERTION SORT

Example:



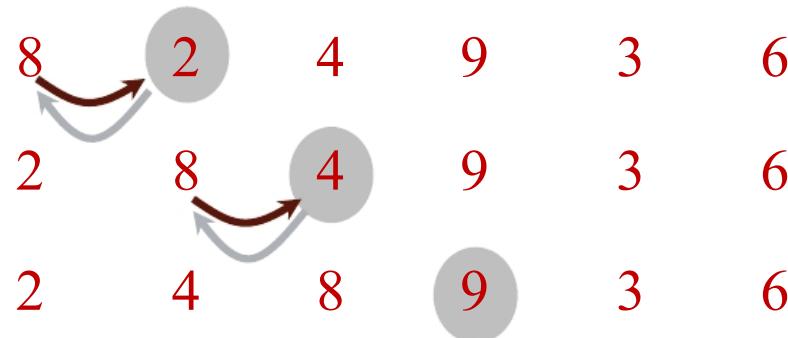
INSERTION SORT

Example:



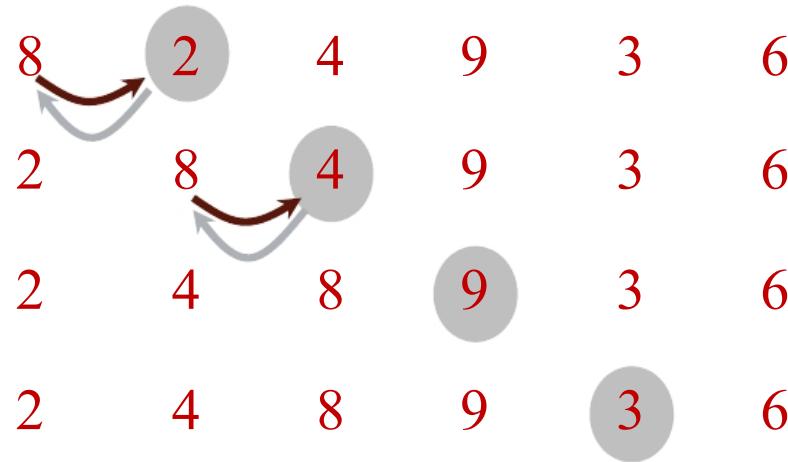
INSERTION SORT

Example:



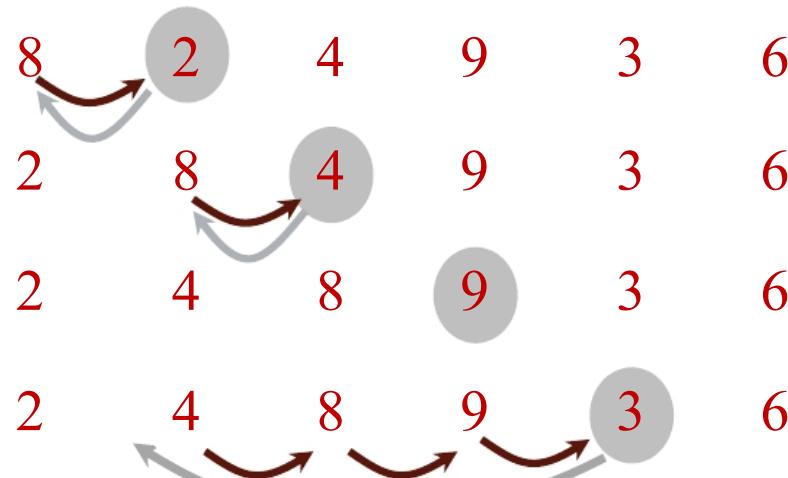
INSERTION SORT

Example:



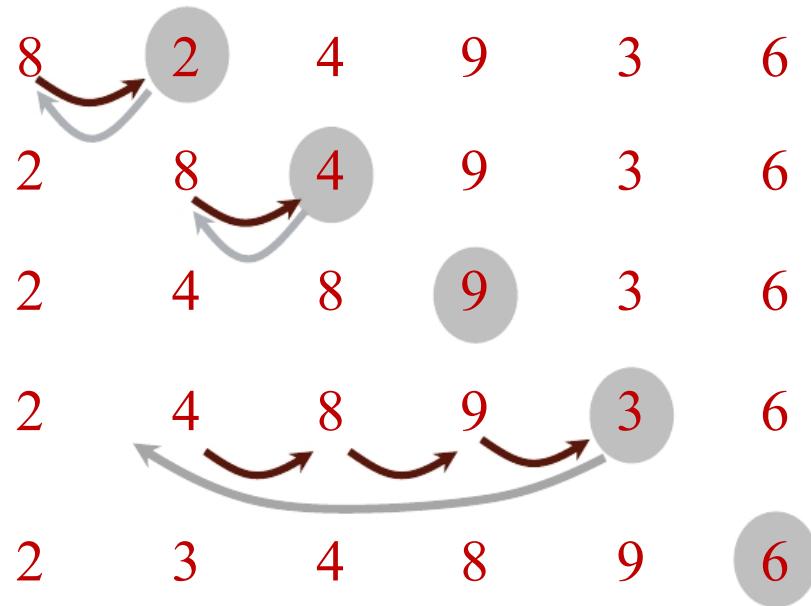
INSERTION SORT

Example:



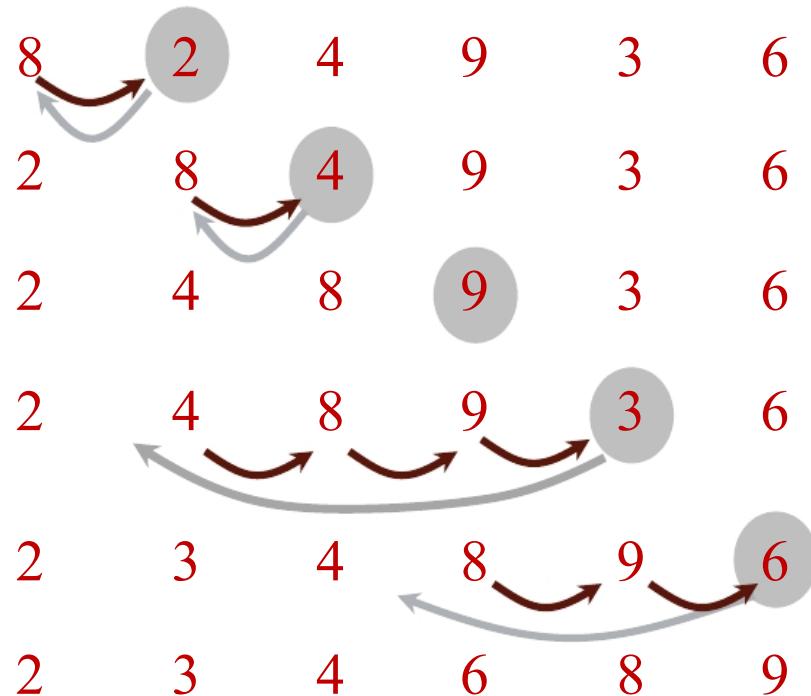
INSERTION SORT

Example:



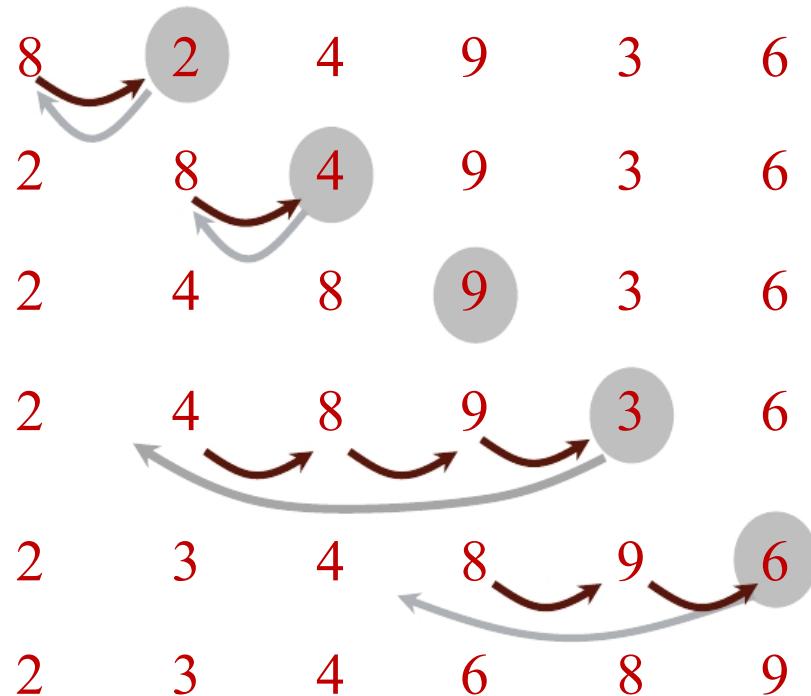
INSERTION SORT

Example:



INSERTION SORT

Example:



CORRECTNESS OF A ALGORITHM

For such an incremental algorithm, we can use **loop invariants** to prove the correctness of the algorithm.

Loop invariants:

Initialization

It is true at the first loop

Maintenance

It is true before an iteration of loop, then true before next iteration

Termination

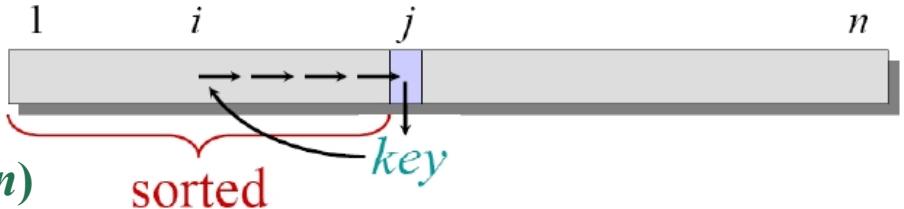
The invariant guarantee the correctness at last iteration.



INSERTION SORT

Pseudocode

```
for  $j \leftarrow 2$  to  $n$ 
do
    key  $\leftarrow A[j]$ 
    //Insert  $A[j]$  into the sorted sequence  $A[1 .. j - 1]$ .
     $i \leftarrow j - 1$ 
    while  $i > 0$  and  $A[i] > \text{key}$ 
    do
         $A[i + 1] \leftarrow A[i]$ 
         $i \leftarrow i - 1$ 
     $A[i + 1] \leftarrow \text{key}$ 
```



RUNNING TIME

The running time depends on **the input**

An already sorted sequence is easier to sort.

Parameterize the running time by the size of the input---***n***, since short sequences are easier to be sorted than the longer ones.

Generally, we seek **upper bounds** on the running time, because everybody likes a guarantee---
worst case.



KINDS OF ANALYSES

Worst-case: (usually)

$T(n) = \text{maximum}$ time of algorithm on any input of size n .

Average-case: (sometimes)

$T(n) = \text{expected}$ time of algorithm over all inputs of size n .

Need assumption of statistical distribution of inputs.

Best-case: (bogus)

Cheat with a slow algorithm that works fast on some input.

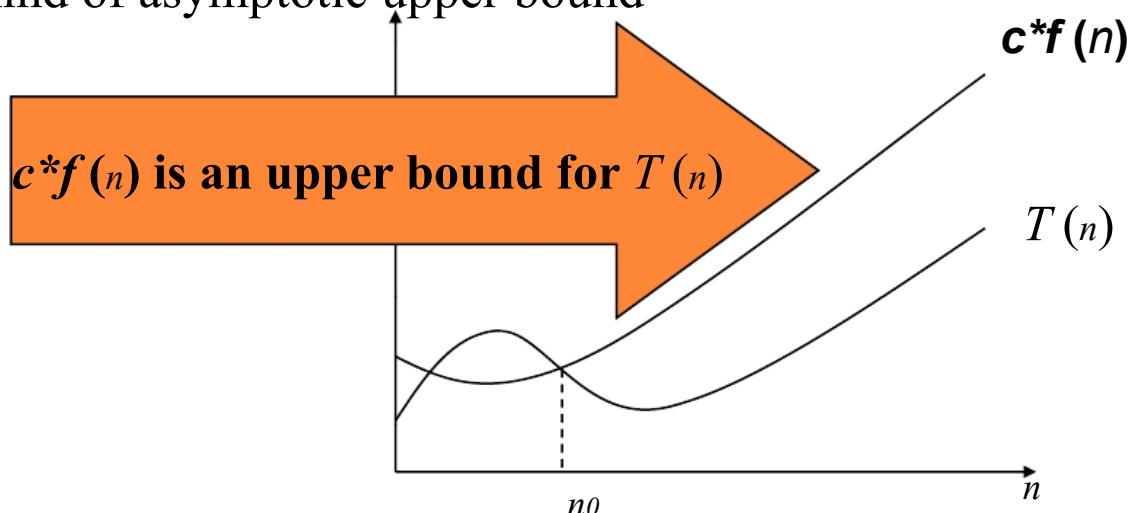


ASYMPTOTIC TIME ANALYSIS

Big O time complexity

$T(n) = O(f(n))$ if there exist positive constant c and n_0 such that $T(n) \leq cf(n)$ when $n \geq n_0$.

A kind of asymptotic upper bound

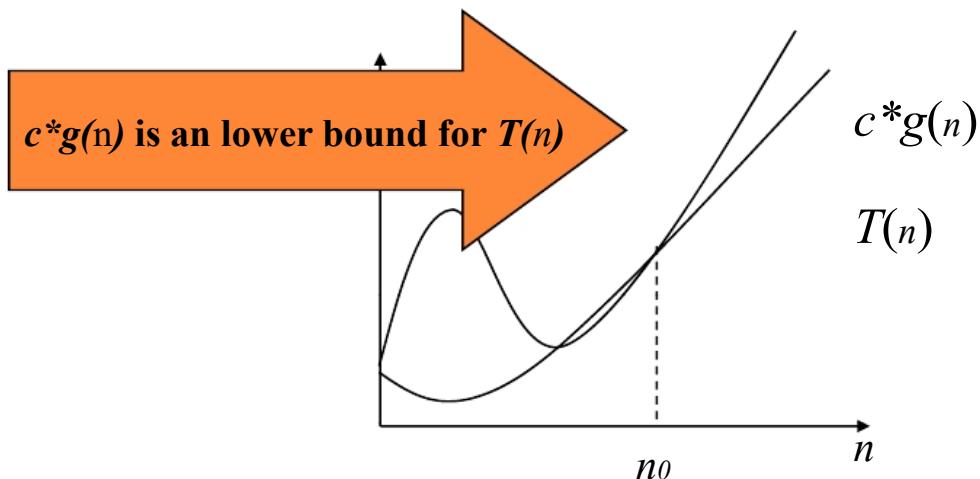


ASYMPTOTIC TIME ANALYSIS

Big Ω time complexity

$T(n) = \Omega(g(n))$ if there exist positive constant c and n_0 such that $T(n) \geq cg(n)$ when $n \geq n_0$.

A kind of asymptotic lower bound



ASYMPTOTIC TIME ANALYSIS

$T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

$T(n) = o(f(n))$ if $T(n) = O(f(n))$ and $T(n) \neq \Theta(f(n))$

We can determine the relative growth rates of $f(n)$ and $g(n)$ by computing $\lim_{n \rightarrow \infty} f(n) / g(n)$

0: $f(n) = o(g(n))$

a constant c : $f(n) = \Theta(g(n))$

infinite: $g(n) = o(f(n))$

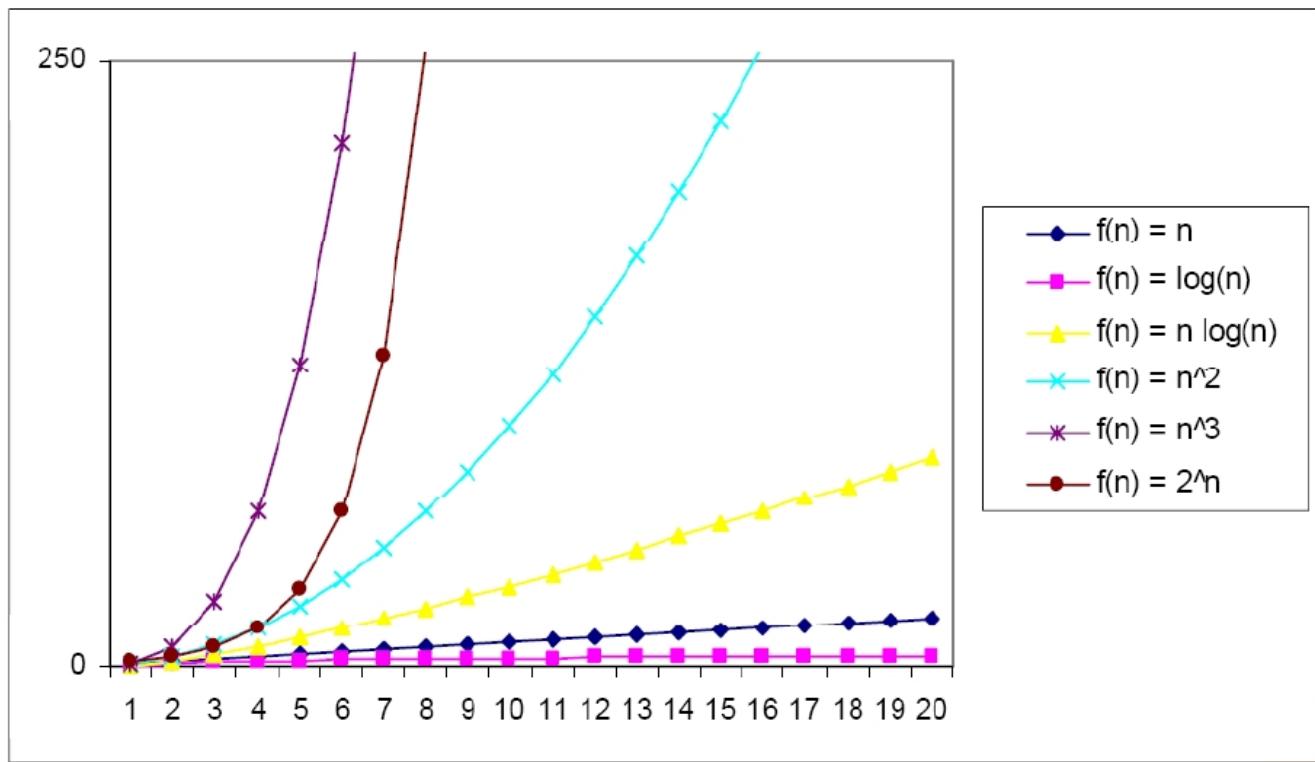
If $T_1(n) = \Theta(f(n))$ and $T_2(n) = \Theta(g(n))$

$$T_1(n) + T_2(n) = \max(\Theta(f(n)), \Theta(g(n)))$$

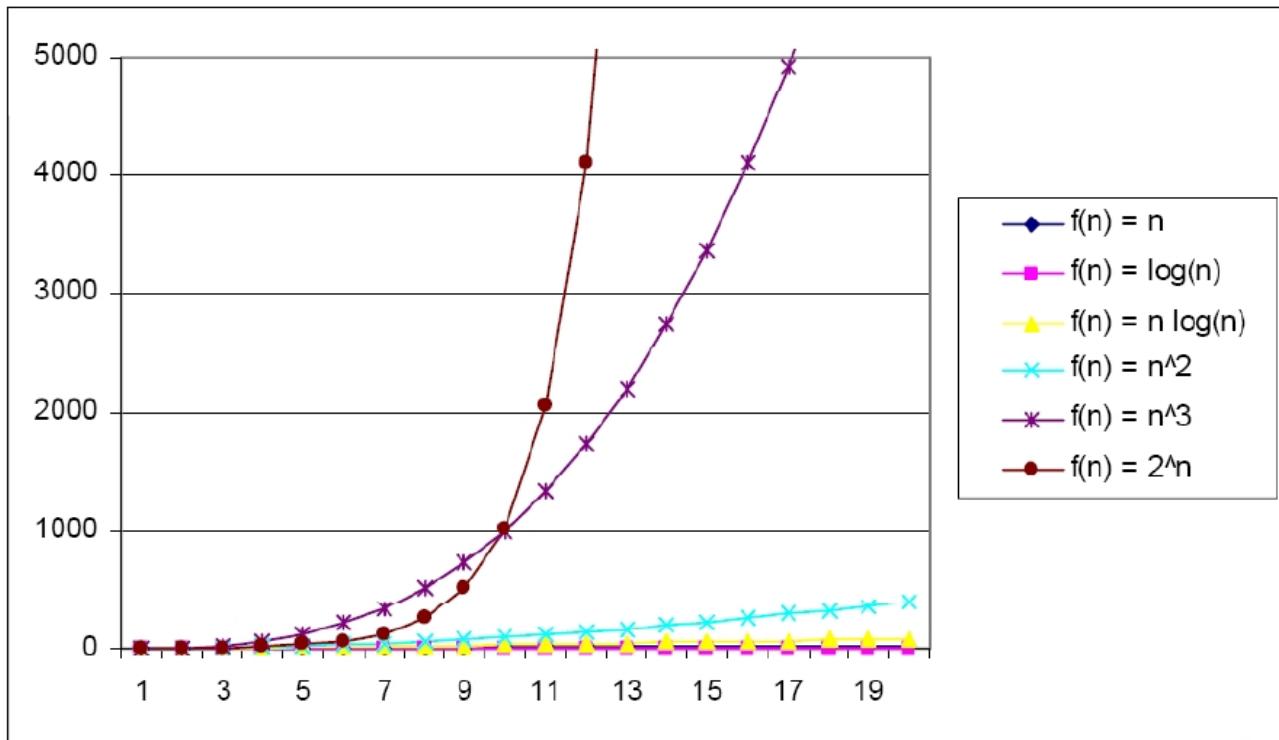
$$T_1(n) * T_2(n) = \Theta(f(n)) * \Theta(g(n))$$



PRACTICAL EXAMPLE



PRACTICAL EXAMPLE



INSERTION SORT ANALYSIS

Cost and times

InsertionSort (A, n)

```
for  $j \leftarrow 2$  to  $n$ 
do
    key  $\leftarrow A[1]
    i \leftarrow j - 1
    \text{while } i > 0 \text{ and } A[i] > \text{key}
    \text{do}
        A[i + 1] \leftarrow A[i]
        i \leftarrow i - 1
    A[i + 1] \leftarrow \text{key}$ 
```

cost times

c_1	n
c_2	$n - 1$
c_3	$n - 1$
c_4	$\sum_{j=2}^n t_j$
c_5	$\sum_{j=2}^n (t_j - 1)$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$n - 1$

INSERTION SORT ANALYSIS

○ Worst Case: decreasing order

- $t_j = j$ $\sum_{j=2}^n t_j = \frac{n(n+1)}{2} - 1$ $\sum_{j=2}^n (t_j - 1) = \frac{n(n-1)}{2}$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

$$T(n) = an^2 + bn + c \quad \Rightarrow \quad T(n) = \Theta(n^2)$$



INSERTION SORT ANALYSIS

Worst Case: decreasing order

InsertionSort (A, n)

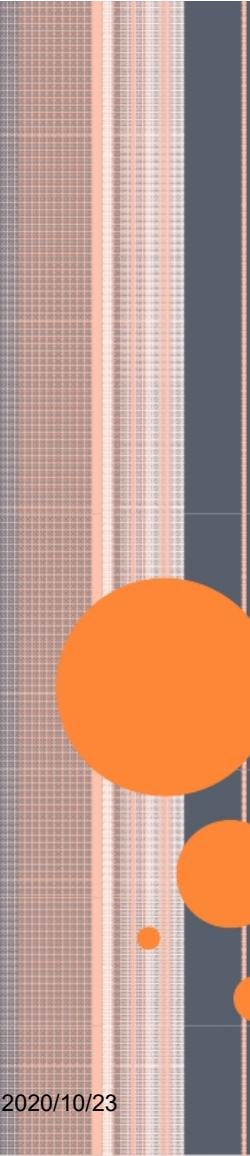
for $j \leftarrow 2$ to n

do

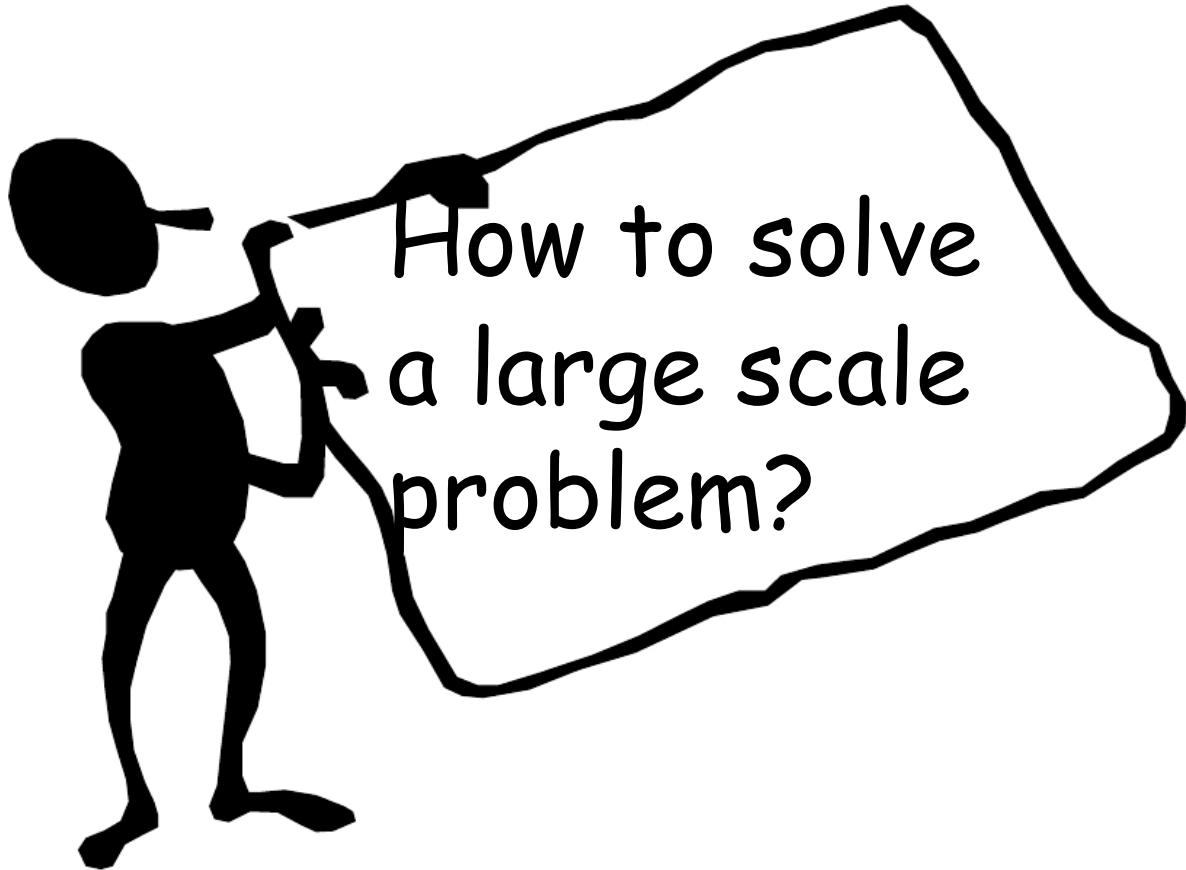
Insert $A[j]$ into the sorted sequence $A[1 .. j - 1]$.

$$T(n) = \sum_{j=2}^n \Theta(j) = \Theta(n^2)$$





RECURRENCE & DIVIDE-AND-CONQUER



How to solve
a large scale
problem?

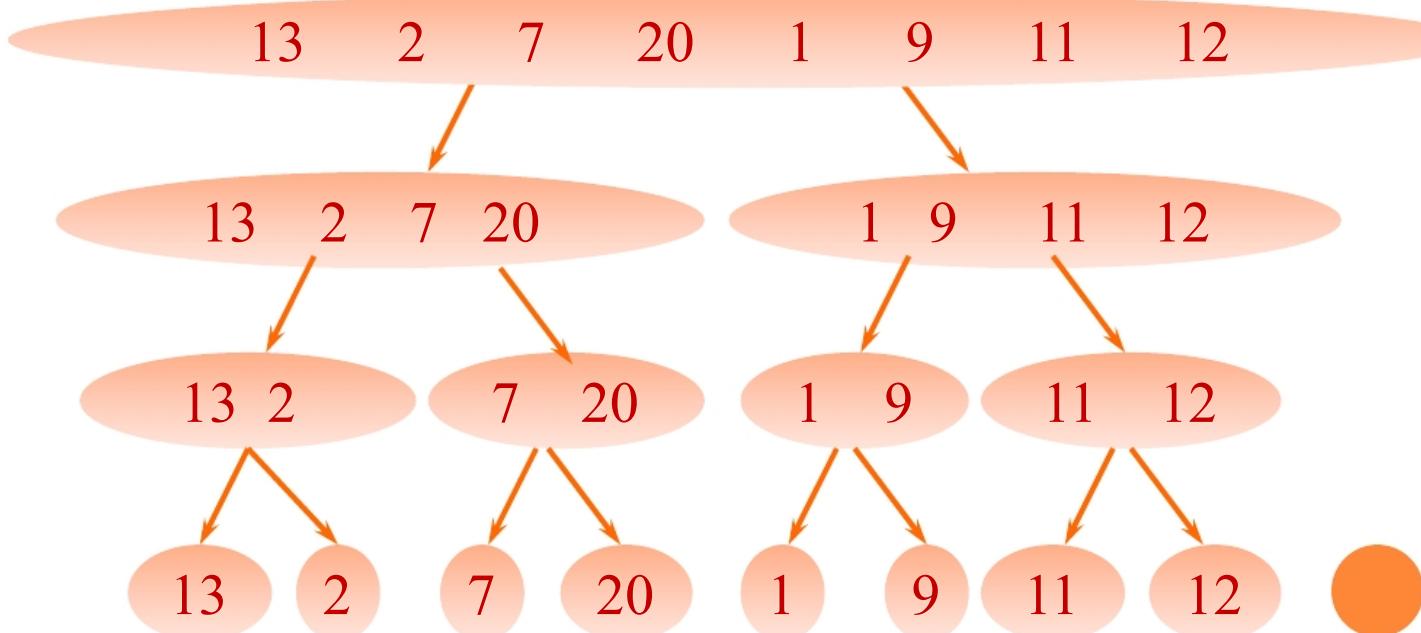


MERGE SORT

Example:

Input: 13, 2, 7, 20, 1, 9, 11, 12

Divide

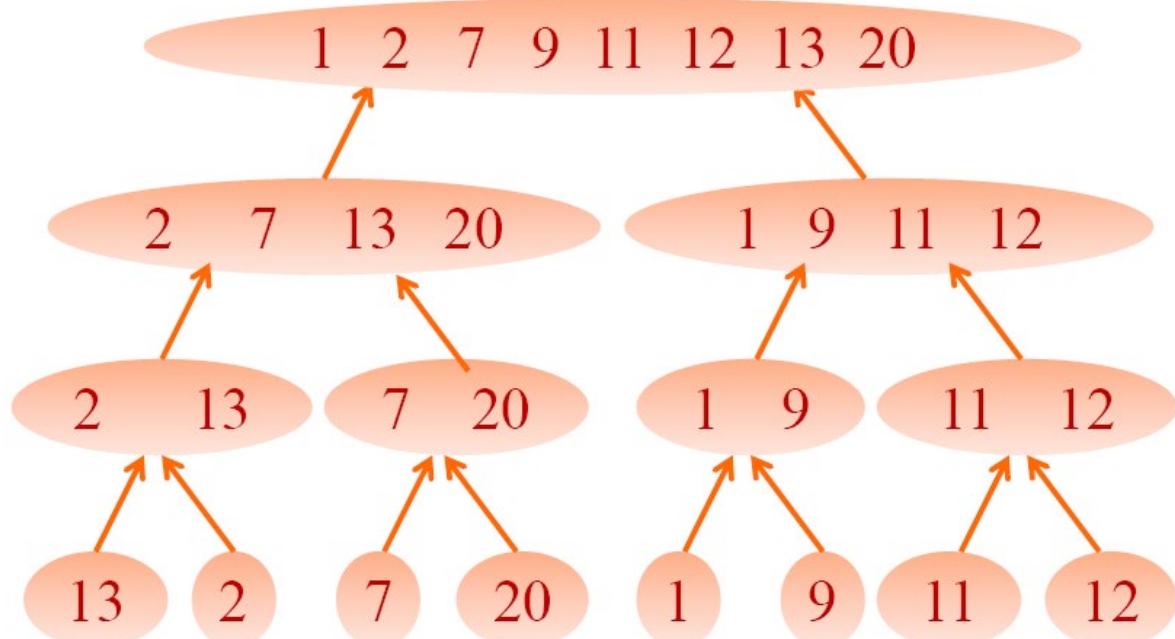


MERGE SORT

Example:

Input: 13, 2, 7, 20, 1, 9, 11, 12

Merge



MERGE SORT

Merge Sort

$\text{MergeSort}(A, p, r)$

if $p < r$

then

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

$\text{MergeSort}(A, p, q)$

$\text{MergeSort}(A, q+1, r)$

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



DIVIDE-AND-CONQUER

Recursive problems

Call themselves recursively one or more times to deal with closely related **subproblems**.

Divide and Conquer

Break the problem into several **subproblems**

Subproblems are similar to the original problem but smaller in size

Conquer the subproblems by solving subproblems
recursively

Then **combine** these solutions to create a solution to the original problem.



MERGE SORT

Divide

Trivial.

Conquer

Recursively sort 2 subarrays.

$$2T(n/2)$$

Combine

Merge Sort

MergeSort(A, p, r)

if $p < r$

then

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

MergeSort(A, p, q)

MergeSort($A, q+1, r$)

Merge(A, p, q, r)



MERGE IN LINEAR TIME

20 12

13 11

7 9

2 1



MERGE IN LINEAR TIME

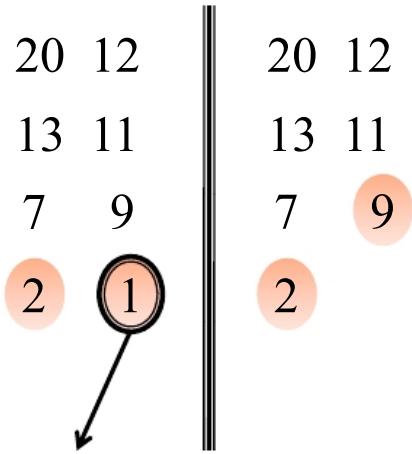
20 12

13 11

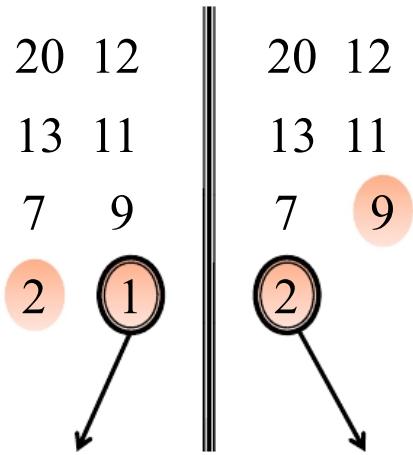
7 9



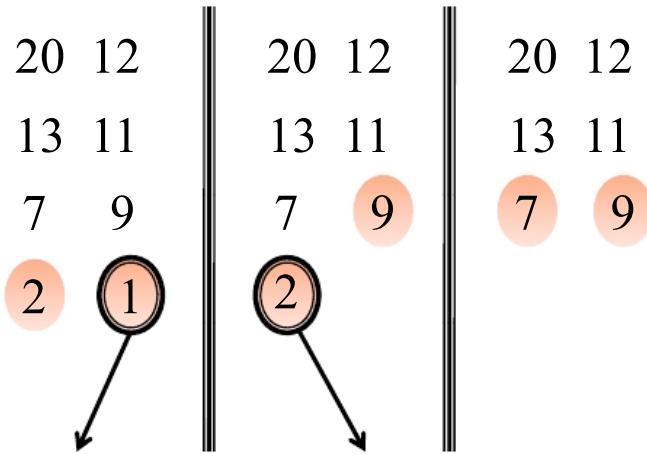
MERGE IN LINEAR TIME



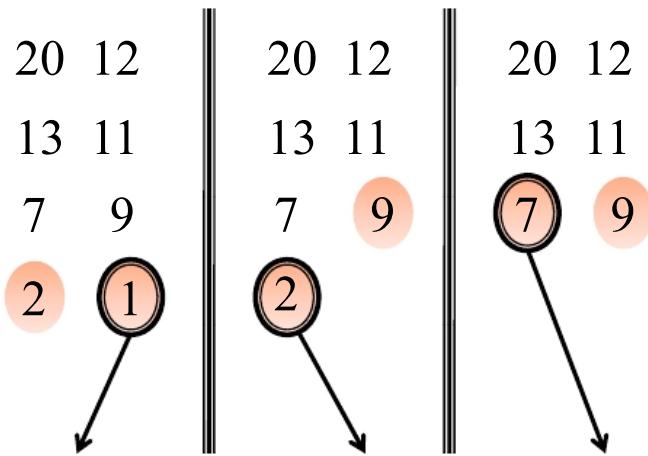
MERGE IN LINEAR TIME



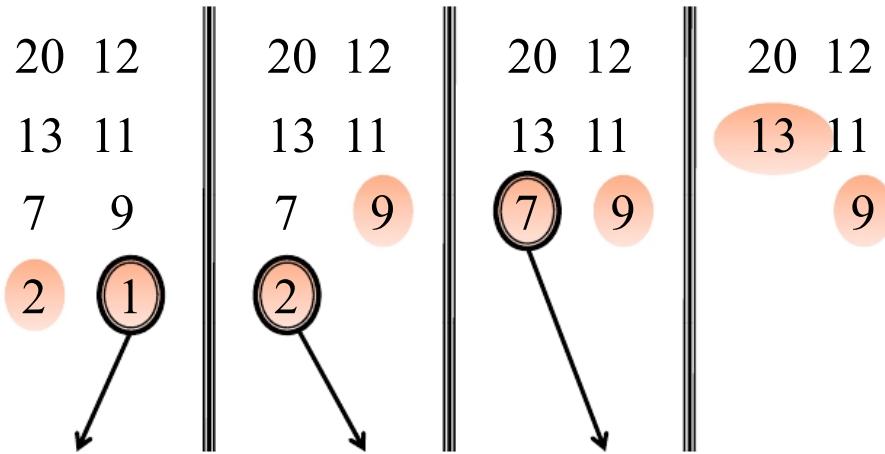
MERGE IN LINEAR TIME



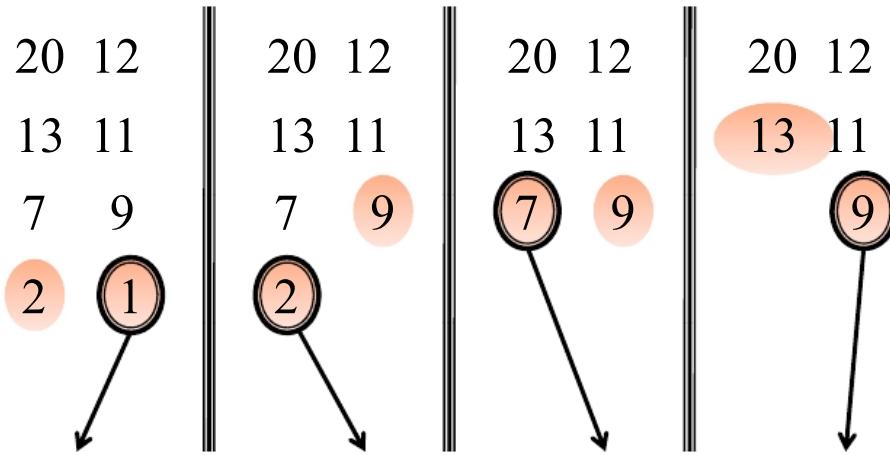
MERGE IN LINEAR TIME



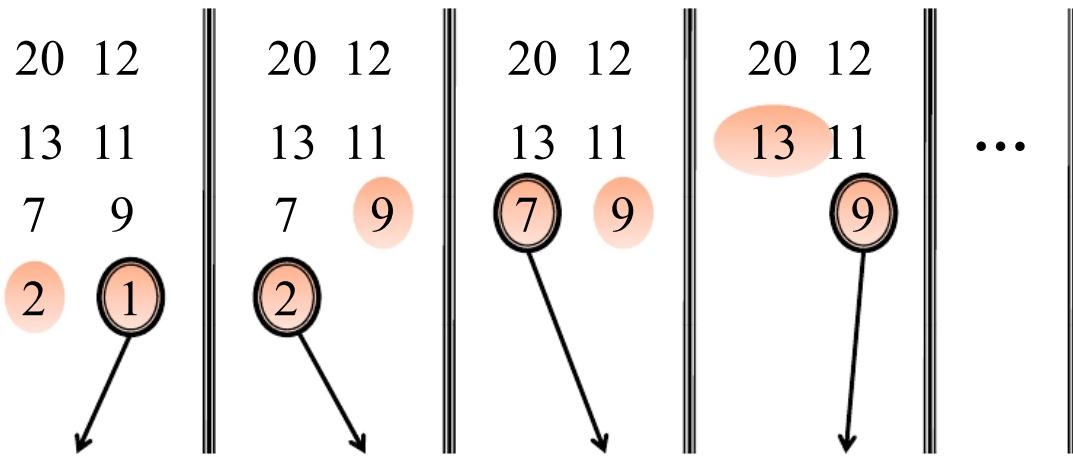
MERGE IN LINEAR TIME



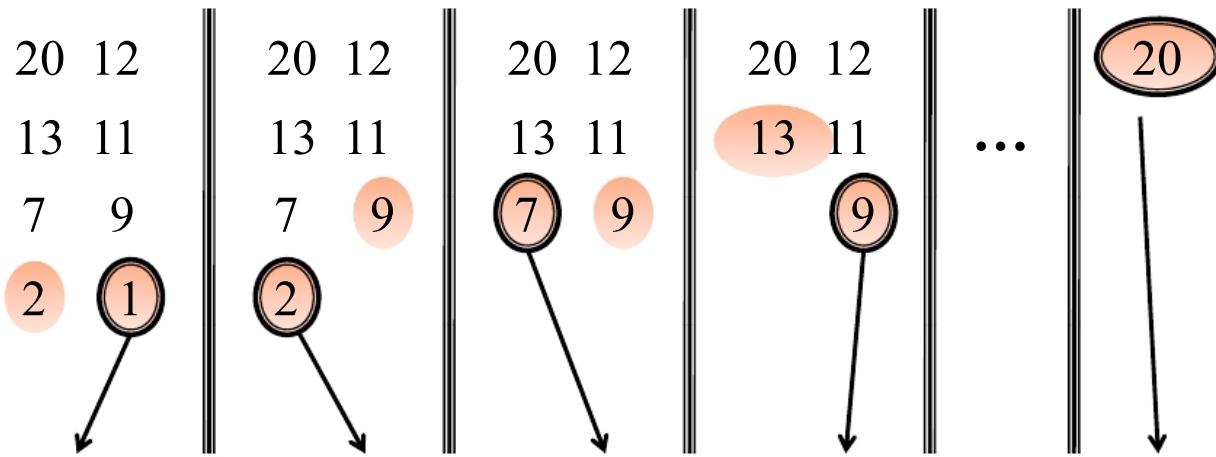
MERGE IN LINEAR TIME



MERGE IN LINEAR TIME



MERGE IN LINEAR TIME



MERGE SORT

Divide

Trivial.

Conquer

Recursively sort 2 subarrays.

$$2T(n/2)$$

Combine

Merge in linear time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \neq 1 \end{cases}$$

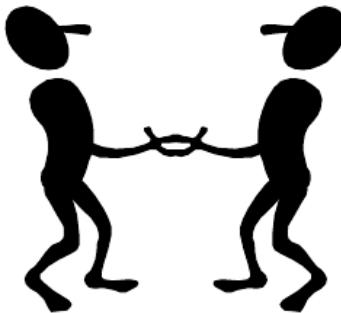
RECURRENCES AND DIVIDE-AND-CONQUER

Recurrence:

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

Recurrence and Divide and Conquer

Twins



???How to resolve a Recurrence?



RECURRENCES

Three methods:

Substitution method

Recursion tree method

Master method



SUBSTITUTION METHOD

The most general method:

1. *Guess* the form of the solution.
2. *Verify* by induction.
3. *Solve* for constants.

Example: $T(n) = 4T(n/2) + n$

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \leq ck^3$ for $k < n$.
- Prove $T(n) \leq cn^3$ by induction.



Example of substitution

- We must also handle the initial conditions, that is, ground the induction with base cases.
- **Base:** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \leq n < n_0$, we have “ $\Theta(1)$ ” $\leq cn^3$, if we pick c big enough.



Example (continued)

$$\begin{aligned} T(n) &= 4T(n / 2) + n \\ &\leq 4c(n / 2)^3 + n \\ &= (c / 2)n^3 + n \\ &= cn^3 - ((c / 2)n^3 - n) \quad \leftarrow \textit{desired - residual} \\ &\leq cn^3 \quad \leftarrow \textit{desired} \end{aligned}$$

Whenever $(c/2)n^3 - n \geq 0$, for example,
If $c \geq 2$ and $n \geq 1$. ← residual

This bound is not tight!



A tighter upper bound?

We shall prove that $T(n) = O(n^2)$.

Assume that $T(k) \leq ck^2$ for $k < n$:

$$T(n) = 4T(n/2) + n$$

$$\leq cn^2 + n$$

$$= cn^2 - (-n)$$

$$\leq cn^2$$

for no choice n when $c > 0$. Lose!



A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.

Inductive hypothesis: $T(k) \leq c_1 k^2 - c_2 k$ for $k < n$.

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4(c_1(n/2)^2 - c_2(n/2)) + n \\ &= c_1n^2 - 2c_2n + n \\ &= c_1n^2 - c_2n - (c_2n - n) \\ &\leq c_1n^2 - c_2n \quad \text{if } c_2 > 1. \end{aligned}$$

Pick c_1 big enough to handle the initial conditions.

Recursion-tree method

- A recursion tree **models** the costs (time) of a recursive execution of an algorithm.
- The **recursion tree method** is good for generating guesses for the **substitution method**.



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

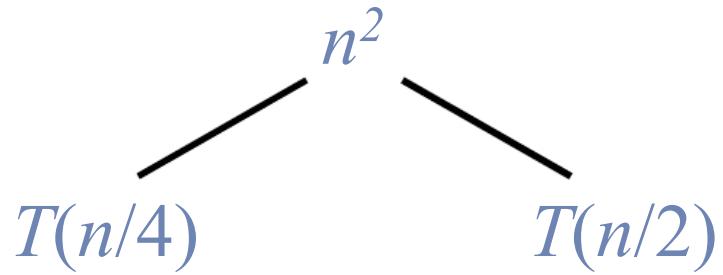
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$T(n)$$



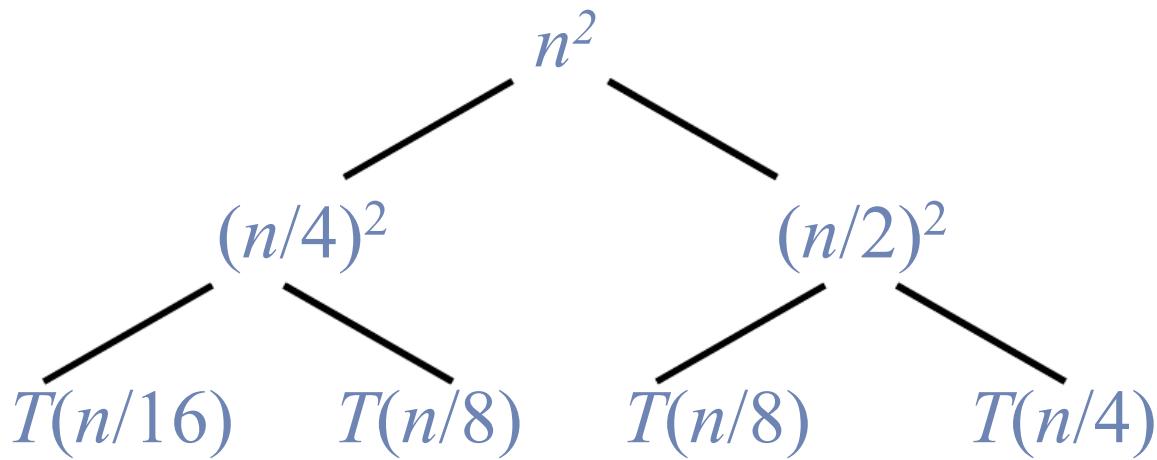
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



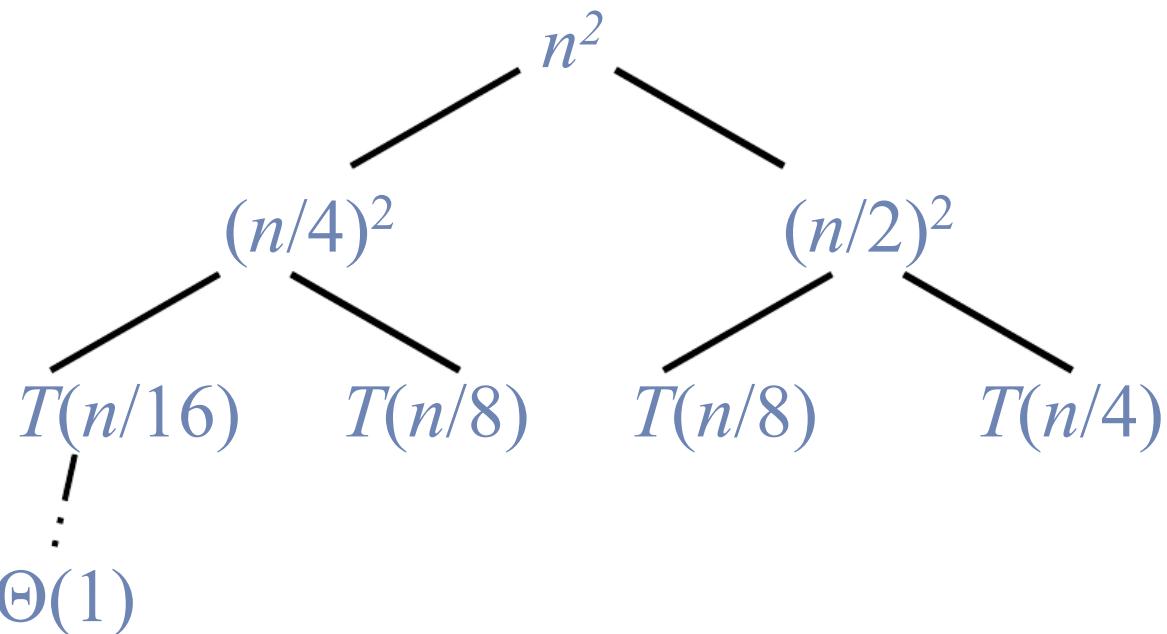
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



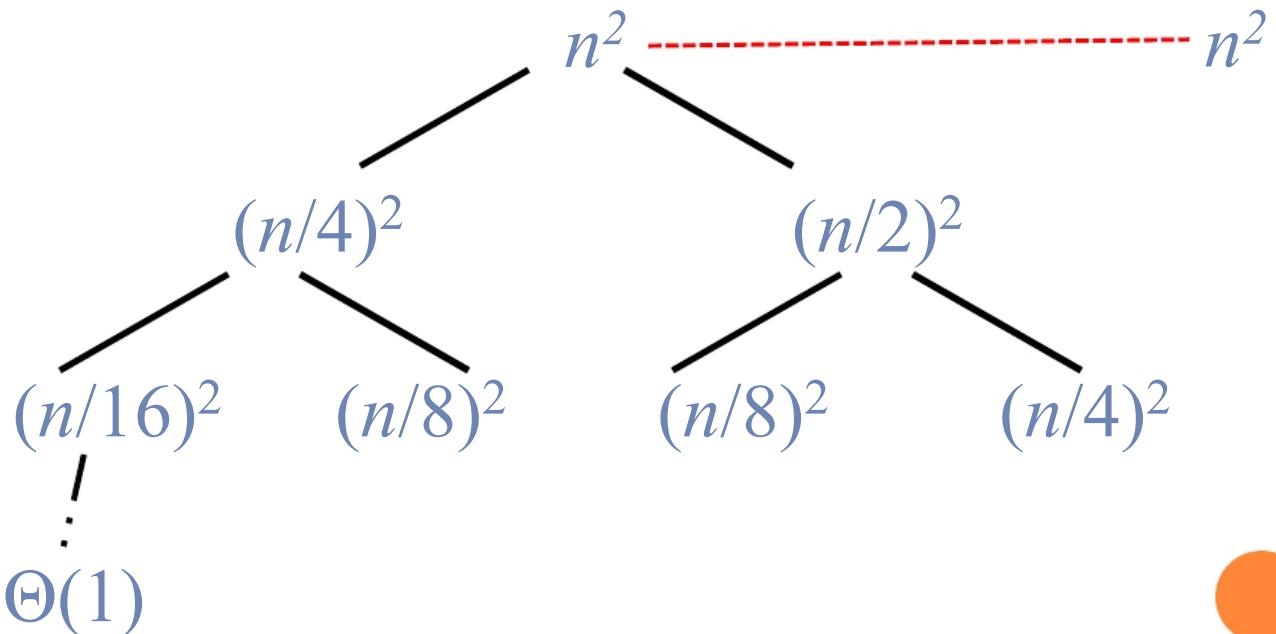
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



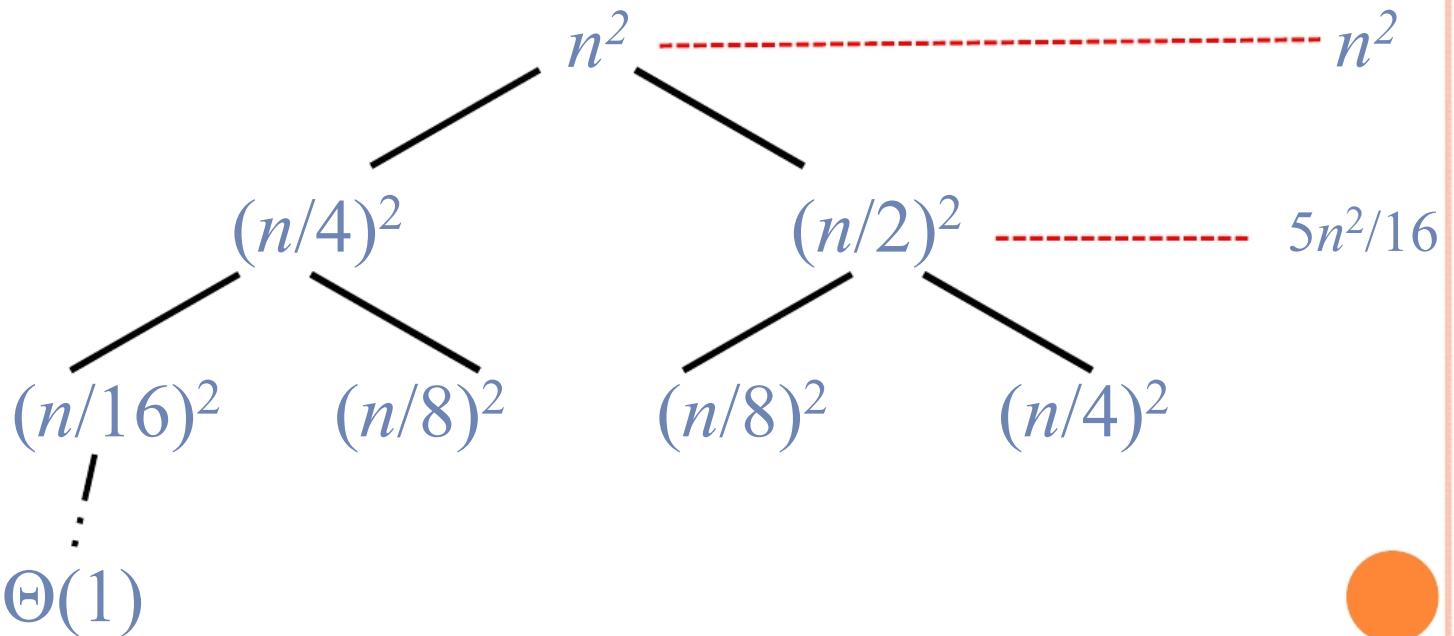
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



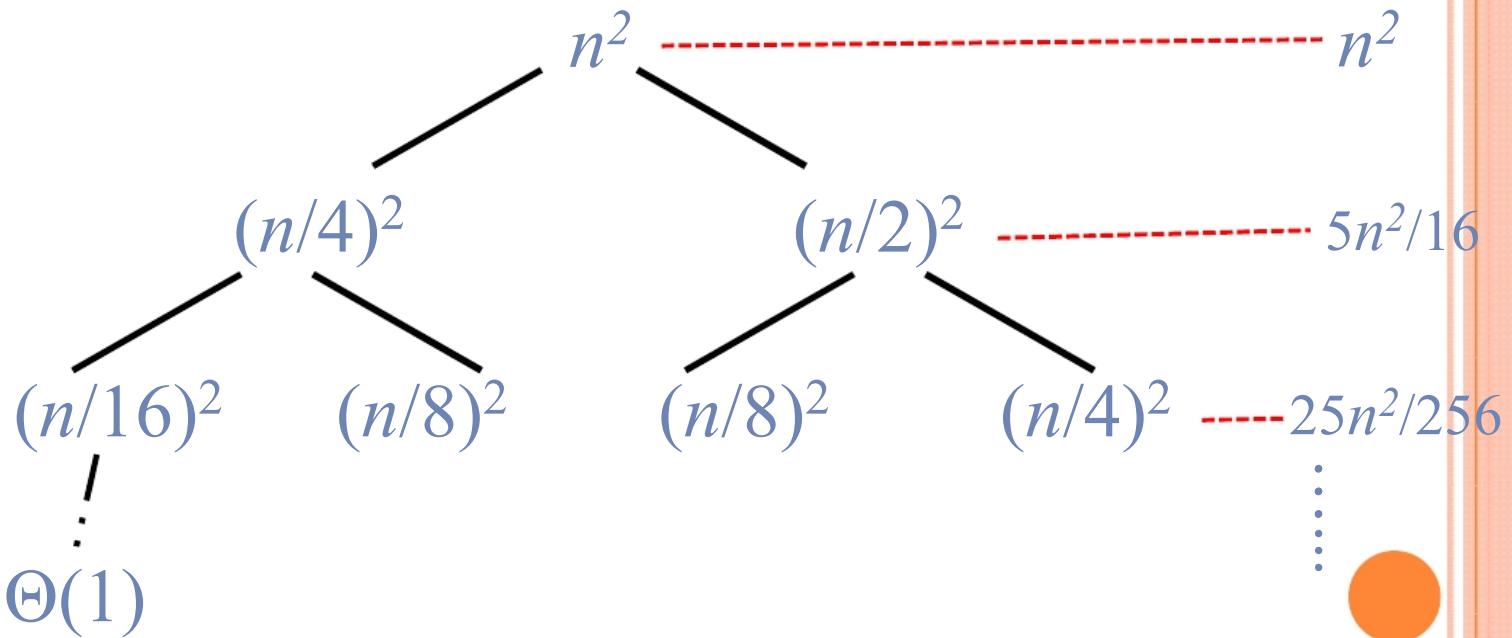
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



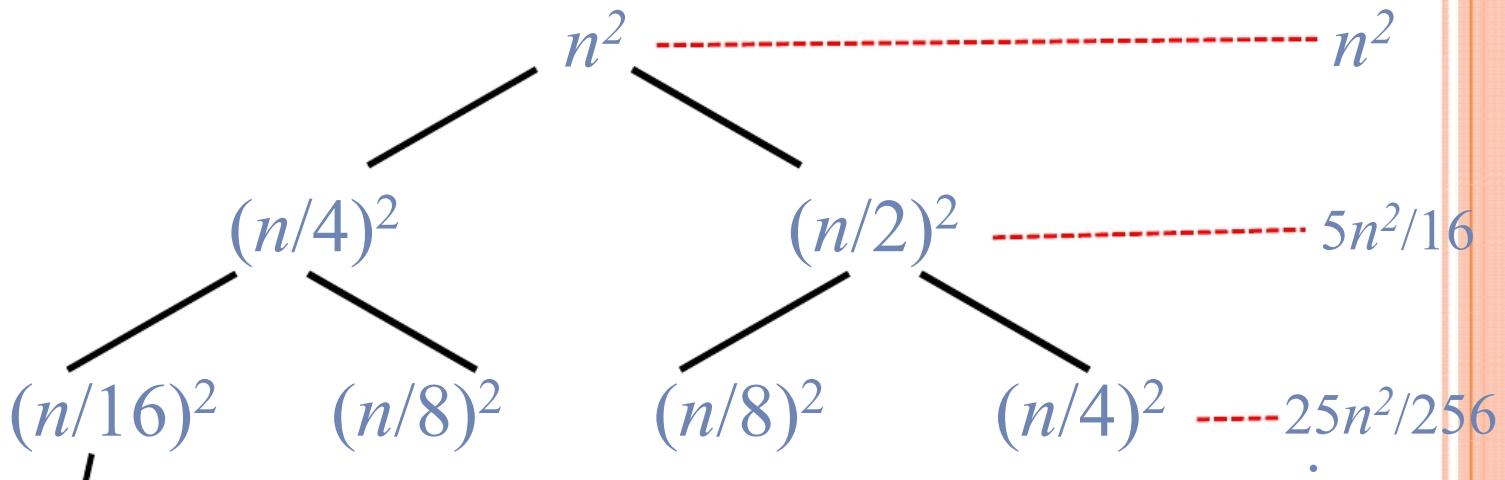
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$\Theta(1)$

$$\begin{aligned} \text{Total} &= n^2 (1 + 5/16 + (5/16)^2 + (5/16)^3 + \dots) \\ &= \Theta(n^2) \end{aligned}$$

geometric series

The master method

The master method applies to recurrences of the form

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

where $a \geq 1$, $b > 1$



证明.

Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

- $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^ε factor).

Solution: $T(n) = \Theta(n^{\log_b a})$.

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.

- $f(n)$ and $n^{\log_b a} \lg^k n$ grow at similar rates.

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.



Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

- $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^ε factor),

and $f(n)$ satisfies the *regularity condition* that $a f(n/b) \leq c f(n)$ for some constant $c < 1$.

Solution: $T(n) = \Theta(f(n))$.



Examples

Ex. $T(n) = 4T(n/2) + n$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$$

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.
 $\therefore T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$$

CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.
 $\therefore T(n) = \Theta(n^2 \lg n)$.



Examples

Ex. $T(n) = 4T(n/2) + n^3$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3.$$

CASE 3: $f(n) = \Omega(n^{2 - \varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.

$$\therefore T(n) = \Theta(n^3).$$

Ex. $T(n) = 4T(n/2) + n^2/\lg n$

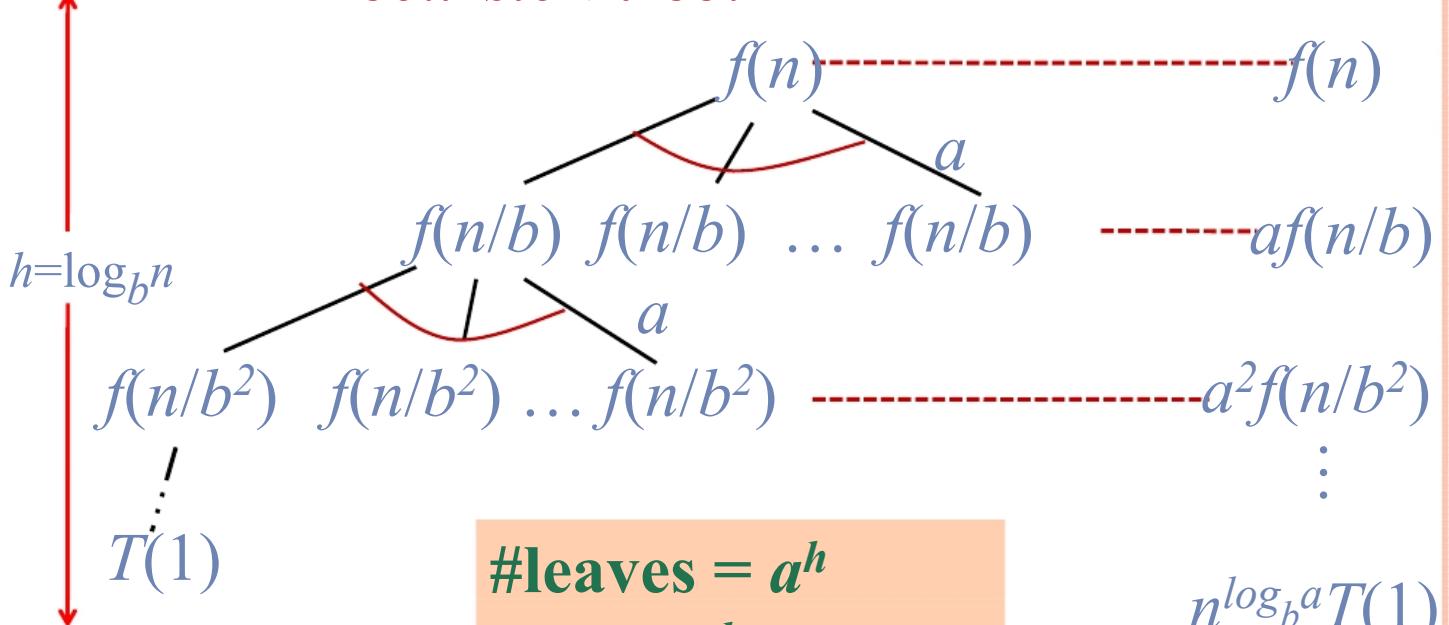
$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2 / \lg n.$$

Master method does not apply.



Idea of master theorem

Recursion tree:



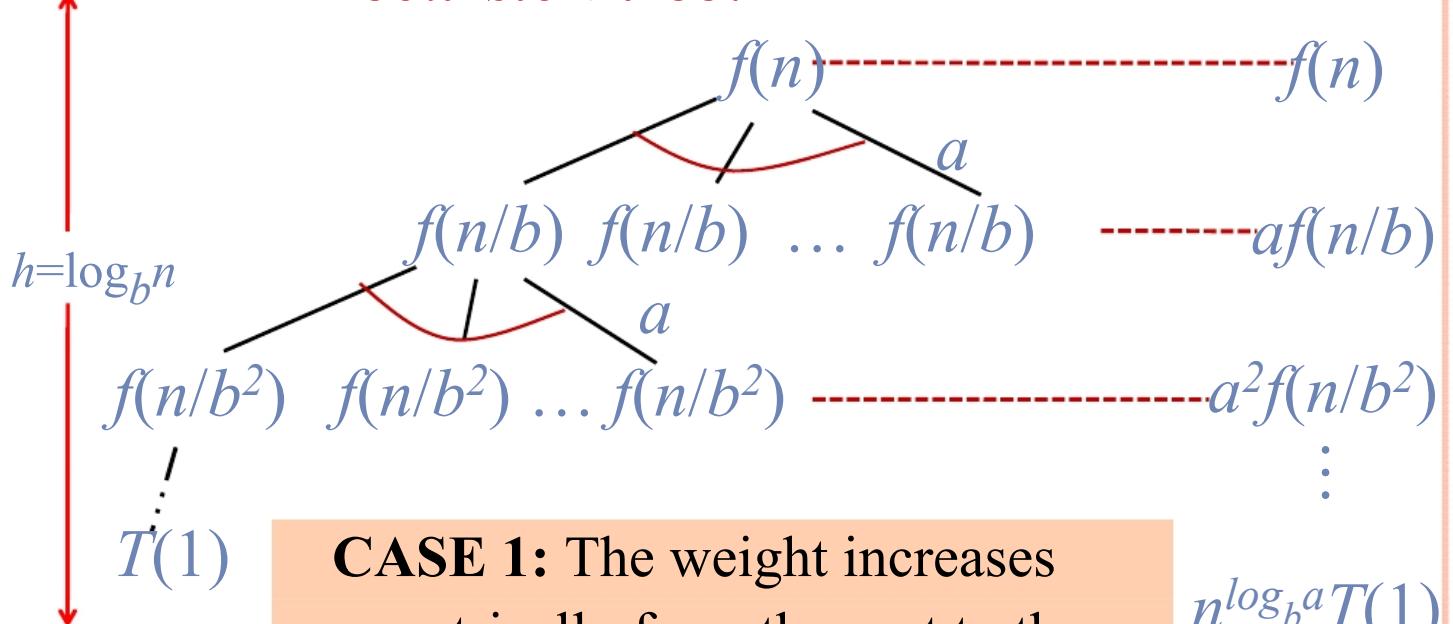
$$\begin{aligned}\#\text{leaves} &= a^h \\ &= a^{\log_b n} \\ &= n^{\log_b a}\end{aligned}$$

$$n^{\log_b a} T(1)$$



Idea of master theorem

Recursion tree:

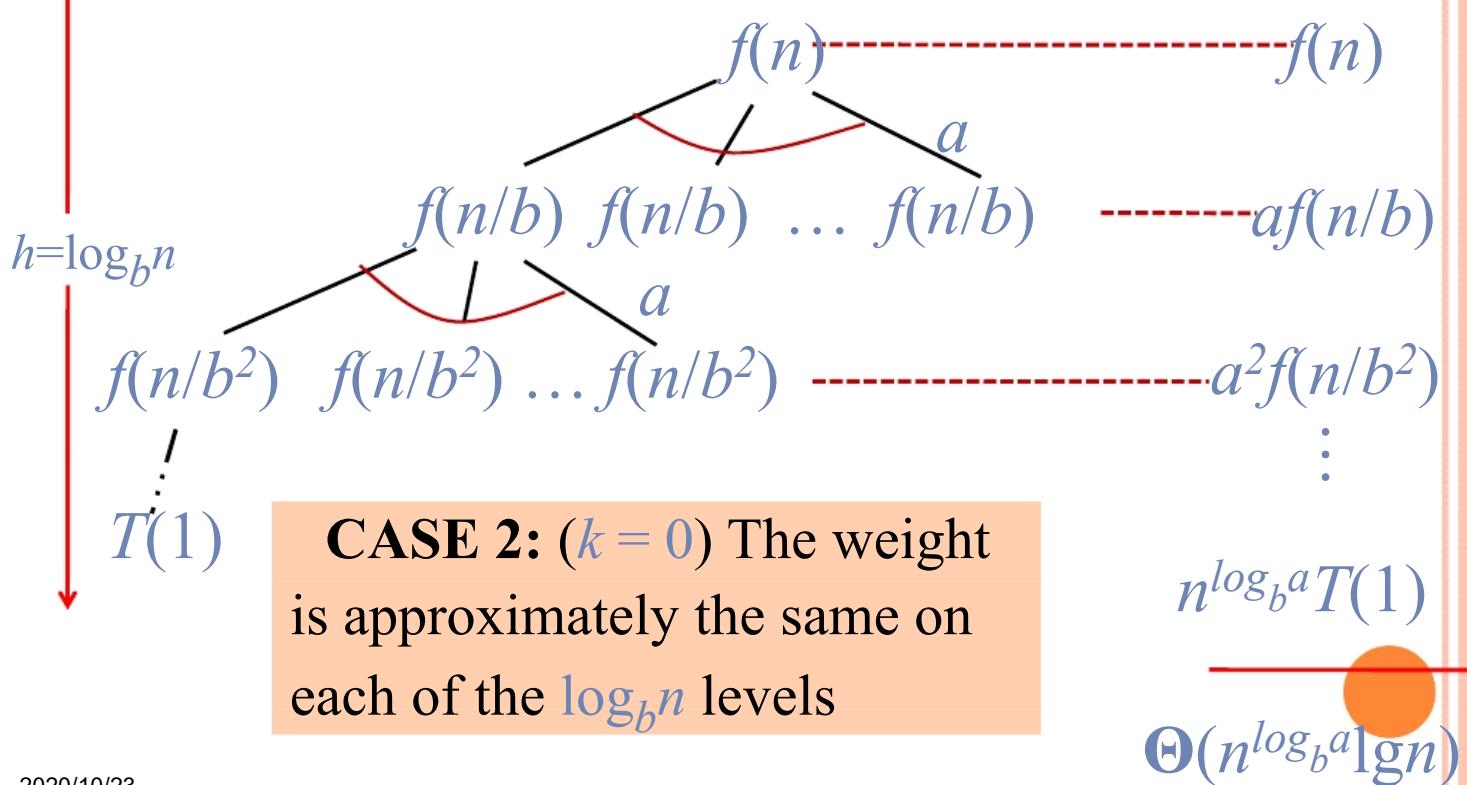


CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight

$n^{\log_b a} T(1)$
 $\Theta(n^{\log_b a})$

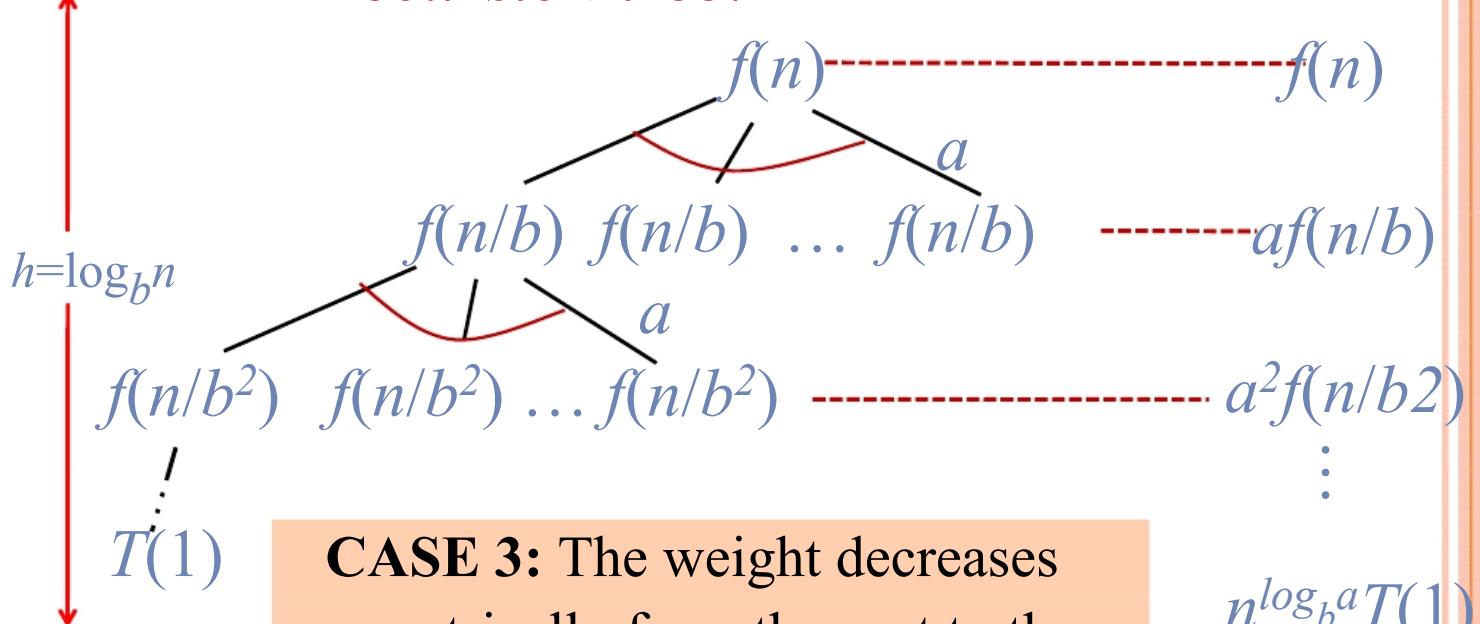
Idea of master theorem

Recursion tree:

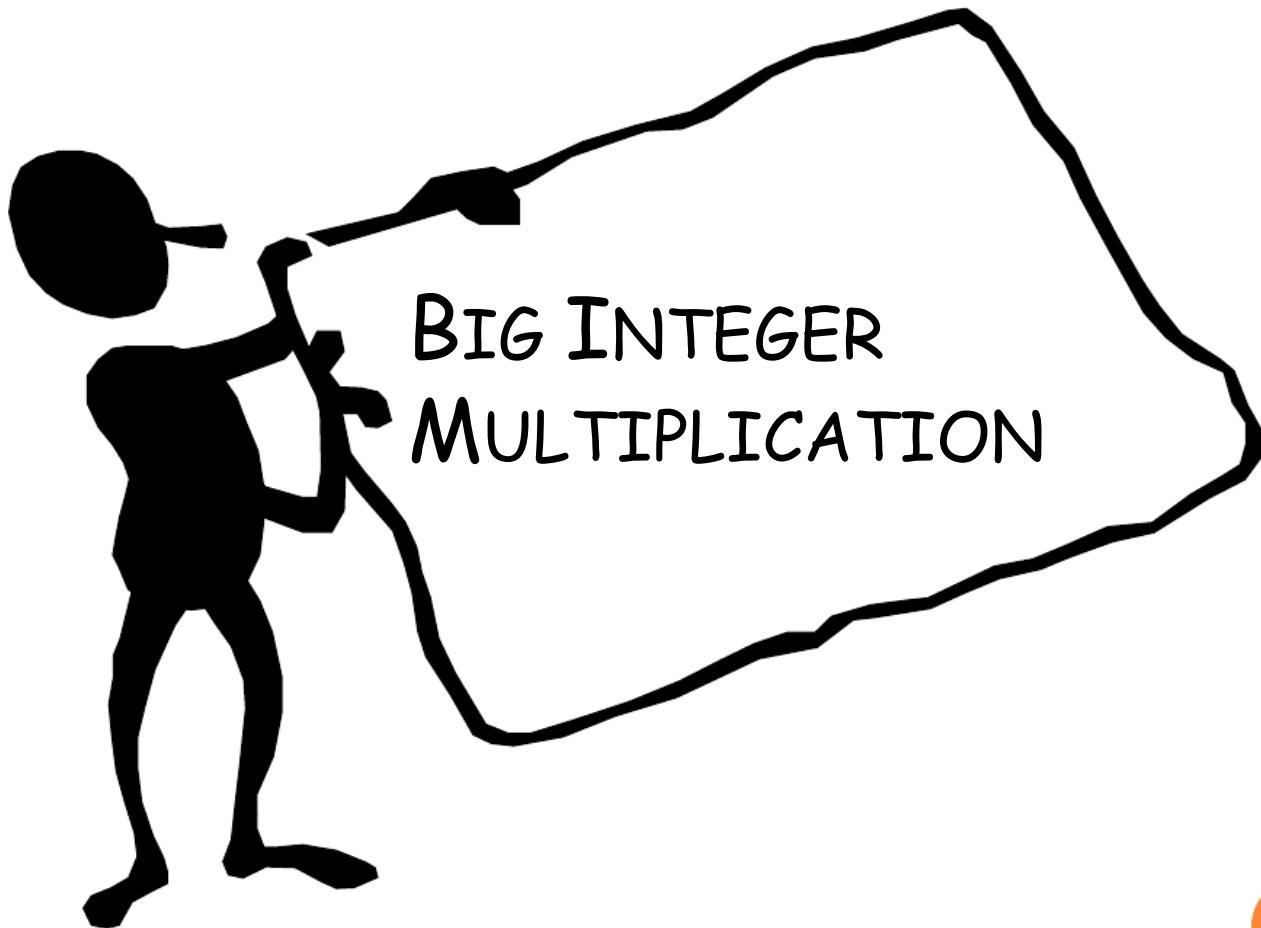


Idea of master theorem

Recursion tree:



CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.



BIG INTEGER MULTIPLICATION

Input: two n-bit integer X and Y

Output: the product of X and Y

Traditional method : $O(n^2)$

low efficiency

Divide-and-conquer:

X =



Y =



$$X = a \cdot 2^{n/2} + b \quad Y = c \cdot 2^{n/2} + d$$

$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$



BIG INTEGER MULTIPLICATION

Input: two n-bit integer X and Y

Output: the product of X and Y

Traditional method : $O(n^2)$

low efficiency

Divide-and-conquer

X =

Y =

X =

Complexity

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 4T(n/2) + \Theta(n) & n > 1 \end{cases}$$

$$T(n) = \Theta(n^2) \quad \text{no improvement}$$

$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$

BIG INTEGER MULTIPLICATION

$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$

Reduce the times of multiplication

1. $XY = ac \cdot 2^n + ((a - c)(b - d) + ac + bd) \cdot 2^{n/2} + bd$
2. $XY = ac \cdot 2^n + ((a + c)(b + d) - ac - bd) \cdot 2^{n/2} + bd$

**Notice: we do not use euqation 2, for summation
may conduct n+1 bits number.**

BIG INTEGER MULTIPLICATION

$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$

Reduce the times of multiplication

1. $XY = ac \cdot 2^n + ((a - b)(d - c) + ac + bd) \cdot 2^{n/2} + bd$
2. $XY = ac \cdot 2^n + ((a + b)(c + d) - ac - bd) \cdot 2^{n/2} + bd$

Complexity

Not
may

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 3T(n/2) + \Theta(n) & n > 1 \end{cases}$$

$$T(n) = \Theta(n^{\log 3}) = \Theta(n^{1.59})$$

improved

BIG INTEGER MULTIPLICATION

Even faster algorithm:

- Divide into more pieces, and use the complex methods to merge, may leads to more optimal algorithm.
- This idea conduct Fast Fourier Transform(FFT)。 FFT can be seen as a complex Divide-and-Conquer method. For Multiple it solve in $\Theta(n \log n)$ 。





STRASSEN MATRIX MULTIPLICATION

$$C = AB \text{ where } C[i][j] = \sum_{k=1}^n A[i][k]B[k][j]$$

Traditional algorithm: $T(n) = \Theta(n^3)$

Divide and Conquer: $T(n) = \Theta(n^{\log 7}) = \Theta(n^{2.81})$

Now the best performance is $\Theta(\textcolor{red}{n^{2.376}})$

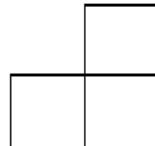
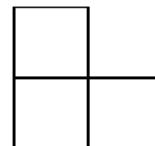
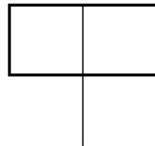
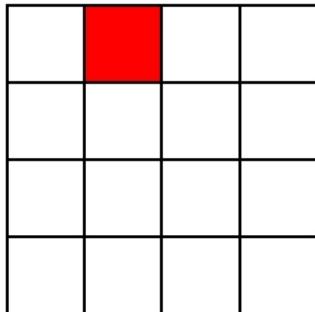




Chessboard Cover

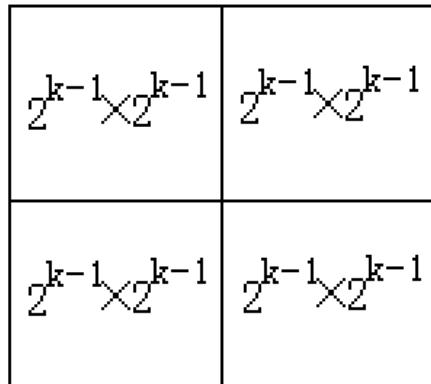
CHESS BOARD COVER

On a $2^k \times 2^k$ chessboard , only one square is different, called *specific*. In the chessboard cover problem, we use the following four kinds of L-shape cards to cover the whole chessboard squares except the specific, and request that there is no overlapping.

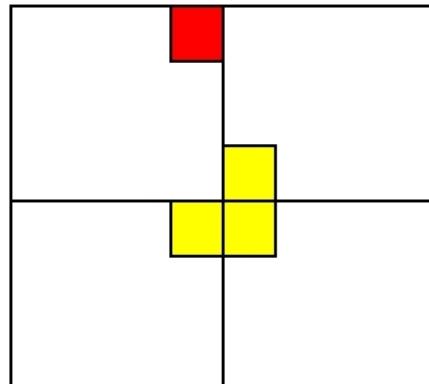


Chessboard Cover

- When $k > 0$, partition $2^k \times 2^k$ chessboard into four $2^{k-1} \times 2^{k-1}$ sub-chessboard
 - The *specific* must be in one of the four sub-chessboard, and the other three have no specific.
- Now lay a L-shaped cards on the joint of the three sub-chessboard.
 - Then we get four smaller chessboard cover problem.
- Do recursively until we get 1×1 chessboard.



(a)



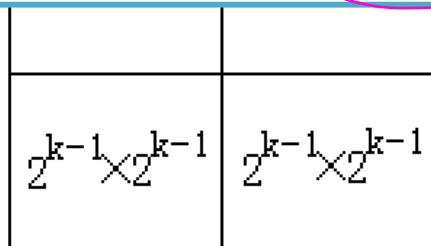
(b)

Chessboard Cover

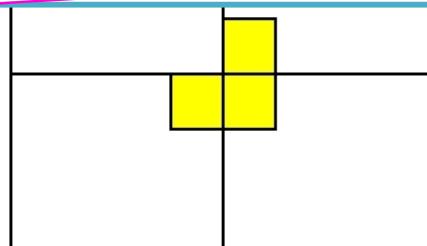
- When $k > 0$, partition $2^k \times 2^k$ chessboard into four $2^{k-1} \times 2^{k-1}$ sub-chessboard
 - The *specific* must be in one of the four sub-chessboard, and the other three have
- Now lay
 - Then
- Do recur

Complexity

$$T(k) = \begin{cases} \Theta(1) & k = 0 \\ 4T(k-1) + \Theta(1) & k > 0 \end{cases}$$
$$T(k) = \Theta(4^k)$$



(a)



(b)

Binary Searching

Given n elements arranged in ascending order, find a particular element K .

Compare the middle element with the particular look up element X , if X is equal to the middle element, then the searching is successful and this algorithm is terminated; if X is less than the middle element, continue the searching in the first half of the sequence; otherwise, continue the searching in the second half of the sequence.



Searching 17 in the sequence [5,8,15,17,25,30,34,39,45,52,60] . Here, variables “low” and “high” stands for the searching scope, “mid” stands for the middle of the searching scope. In fact, $\text{mid}=(\text{low}+\text{high})/2$

0	1	2	3	4	5	6	7	8	9	10
5	8	15	17	25	30	34	39	45	52	60

mid=3 low=3 high=4

Successfully complete the searching, terminate the searching algorithm.

SORTING ALGORITHMS



Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

OUTLINE

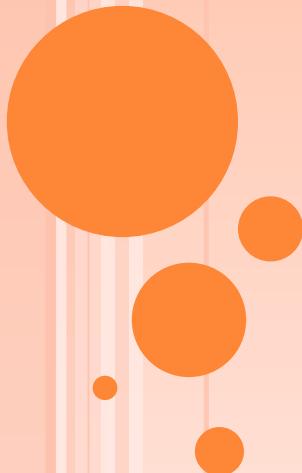
证明 比较排序复杂度 $O(n \log n)$

决策树

- Why We Do Sorting?
- Review of Learned Sorting Algorithms
- How Fast can We Sort so far?
- Linear-Time Sorting Algorithms
 - ❖ Counting Sort
 - ❖ Radix Sort
 - ❖ Bucket Sort



WHY WE DO SORTING



WHY WE DO SORTING?

- Commonly encountered programming task in computing.
- Examples of sorting:
 - ❖ List containing exam scores sorted from Lowest to Highest or from Highest to Lowest
 - ❖ List Rank of Milk Powder Producer Brands in China from the Star to Notorious
 - ❖ List Search Results of a User Query by Relevance from the WWW.



WHY WE DO SORTING?

- Searching for an element in an array will be more efficient. (example: looking up for information like phone number).
- It's always nice to see data in a sorted display. (example: spreadsheet or database application).
- Computers sort things much faster.



SORTING PROBLEM

Description:

- **Input:** sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.
- **Output:** A permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that
 $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

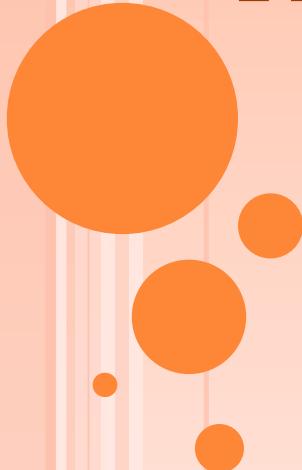
Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9



LEARNED SORTING ALGORITHMS



LEARNED SORTING ALGORITHMS

- ❖ Selection Sort, Bubble Sort
- ❖ Insertion Sort, Binary Insertion Sort,
Shell Sort
- ❖ Merge Sort, Heap Sort, Quick Sort



SHELL SORT

- Shell sort
 - ❖ An algorithm that first beats the $O(N^2)$ barrier
 - ❖ Suitable performance for general use
- Very popular
 - ❖ It is the basis of **the default R sort()** function
- Tunable algorithm
 - ❖ Can use different orderings for comparisons



SHELL SORT

- Donald L. Shell (1959)
 - ❖ A High-Speed Sorting Procedure
Communications of the Association for Computing Machinery 2:30-32
 - ❖ Systems Analyst working at GE
 - ❖ Back then, most computers read punch-cards
- Also called:
 - ❖ Diminishing increment sort
 - ❖ “Comb” sort
 - ❖ “Gap” sort



INTUITION

- Insertion sort is effective:
 - ❖ For small datasets
 - ❖ For data that is nearly sorted
- Insertion sort is inefficient when:
 - ❖ Elements must move far in array



THE IDEA ...

- Allow elements to move large steps
- Bring elements close to final location
 - ❖ First, ensure array is nearly sorted ...
 - ❖ then, run insertion sort ...

How?

Sort interleaved arrays first



SHELL SORT RECIPE

- Decreasing sequence of step sizes h
 - ❖ Every sequence must end at 1
 - ❖ ... , 8, 4, 2, 1
- For each h , sort sub-arrays that start at arbitrary element and include every h th element
 - ❖ If $h = 4$
 - ❖ Sub-array with elements 1, 5, 9, 13 ...
 - ❖ Sub-array with elements 2, 6, 10, 14 ...
 - ❖ Sub-array with elements 3, 7, 11, 15 ...
 - ❖ Sub-array with elements 4, 8, 12, 16 ...



SHELL SORT NOTES

- Any decreasing sequence that ends at 1 will do...
 - ❖ The final pass ensures array is sorted
- Different sequences can dramatically increase (or decrease) performance
- Code is similar to insertion sort



SUB-ARRAYS WHEN INCREMENT IS 5

5-sorting an array



Elements in each subarray color coded



C Code: Shellsort

```
void sort(Item a[], int sequence[], int start, int stop)
{
    int step, i;

    for (int step = 0; sequence[step] >= 1; step++)
    {
        int inc = sequence[step];

        for (i = start + inc; i <= stop; i++)
        {
            int j = i;
            Item val = a[i];

            while ((j >= start + inc) && val < a[j - inc])
            {
                a[j] = a[j - inc];
                j -= inc;
            }

            a[j] = val;
        }
    }
}
```

```
#include "stdlib.h"
#include "stdio.h"

#define Item int

void sort(Item a[], int sequence[], int start, int stop);

int main(int argc, char * argv[])
{
    printf("This program uses shell sort to sort a random array\n\n");
    printf(" Parameters: [array-size]\n\n");

    int size = 100;
    if (argc > 1) size = atoi(argv[1]);

    int sequence[] = { 364, 121, 40, 13, 4, 1, 0};
    int * array = (int *) malloc(sizeof(int) * size);

    srand(123456);
    printf("Generating %d random elements ...\n", size);
    for (int i = 0; i < size; i++)
        array[i] = rand();

    printf("Sorting elements ...\n");
    sort(array, sequence, 0, size - 1);

    printf("The sorted array is ...\n");
    for (int i = 0; i < size; i++)
        printf("%d ", array[i]);
    printf("\n");
    free(array);
}
```

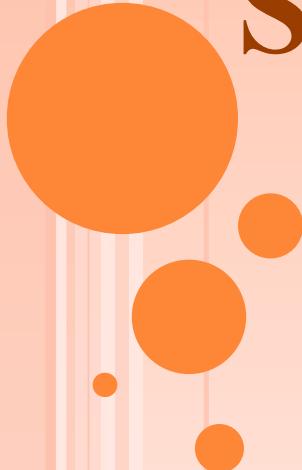


ANALYSIS OF SHELL SORT

- The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed important. It's also an excellent choice for repetitive sorting of smaller lists.
- For good increment sequences, requires time proportional to
 - ❖ $O(N (\log N)^2)$ ($n \rightarrow \infty$)
 - ❖ $O(N^{1.25})$
- 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.



HOW FAST CAN WE SORT BY NOW



HOW FAST CAN WE SORT?

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

- E.g., insertion sort, merge sort, quicksort, heapsort.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Q: Is $O(n \lg n)$ the best we can do?

A: Yes, as long as we use comparison sorts

TODAY: Prove any comparison sort has $\Omega(n \lg n)$ worst case running time



THE TIME COMPLEXITY OF A PROBLEM

The minimum time needed by an algorithm to solve it.

Upper Bound:

Problem P is solvable in time $T_{\text{upper}}(n)$

if there is an algorithm A which

- outputs the correct answer
- in this much time

$\exists A, \forall I, A(I)=P(I)$ and $\text{Time}(A,I) \leq T_{\text{upper}}(|I|)$

Lower Bound:

Time $T_{\text{lower}}(n)$ is a lower bound for problem
if no algorithm solve the problem faster.

$\forall A, \exists I, A(I) \neq P(I)$ or $\text{Time}(A,I) \geq T_{\text{lower}}(|I|)$



TODAY:

PROVED A LOWER BOUND FOR
ANY COMPARISON BASED ALGORITHM
FOR THE SORTING PROBLEM

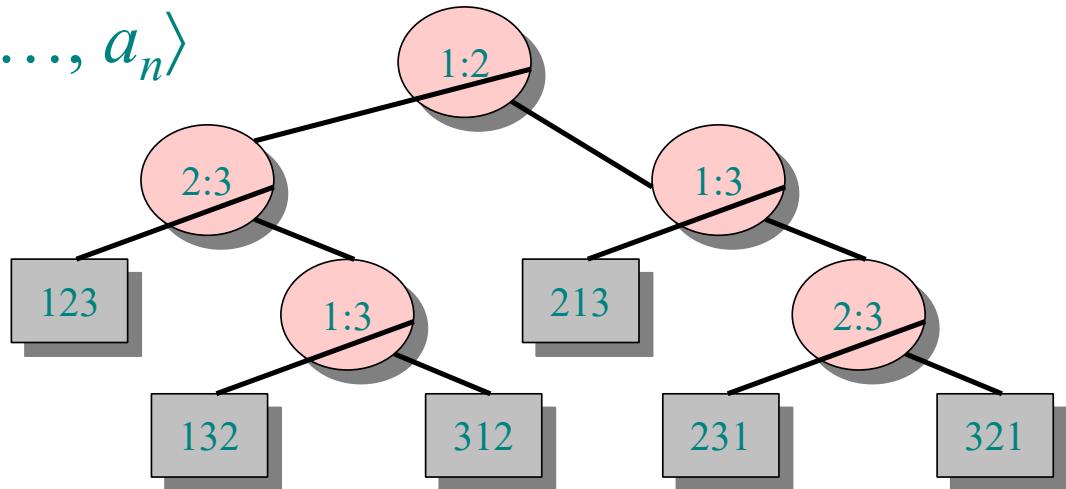
How?

Decision trees help us.



DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, \dots, a_n \rangle$



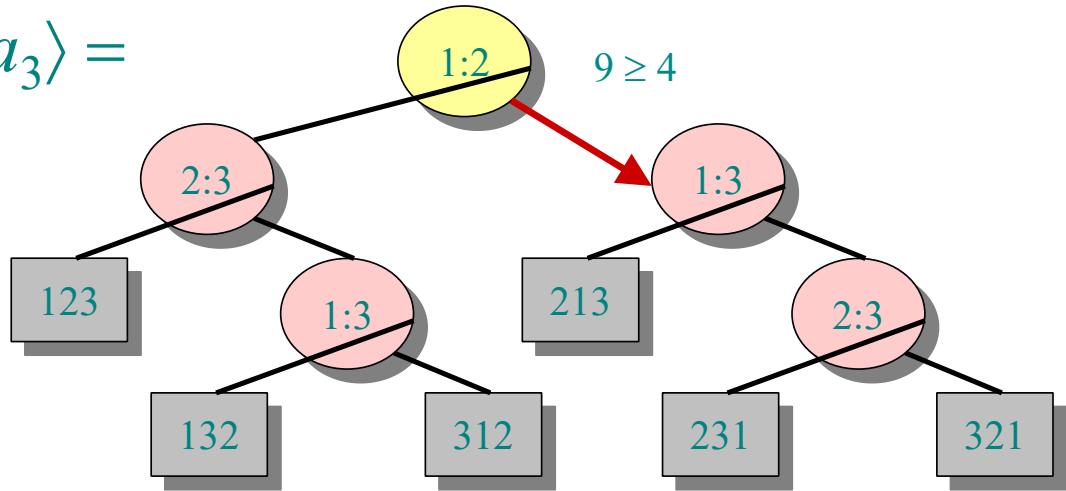
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:



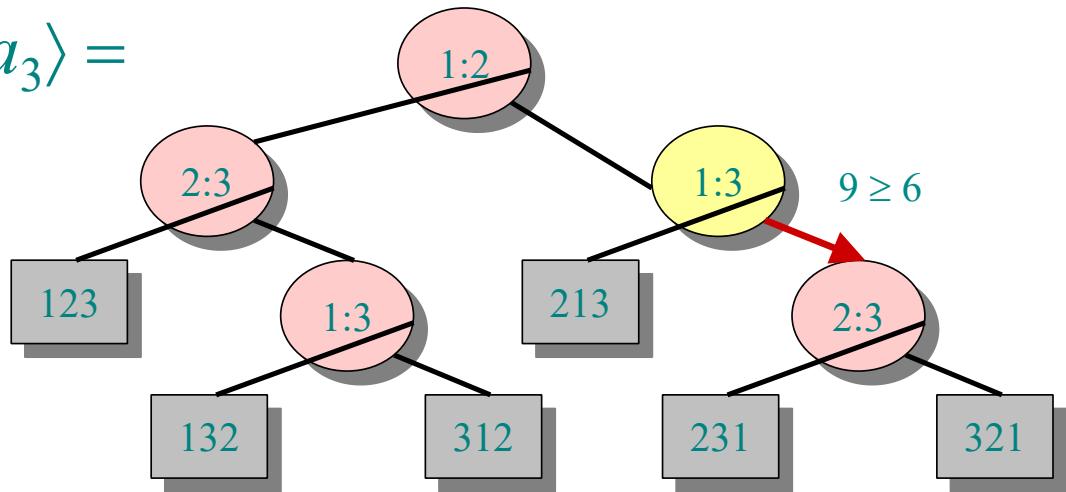
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:



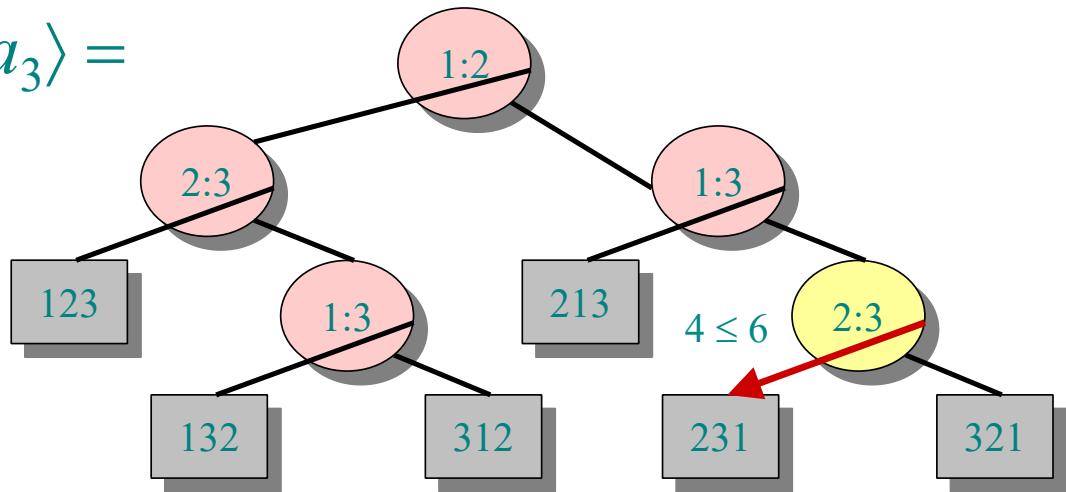
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:



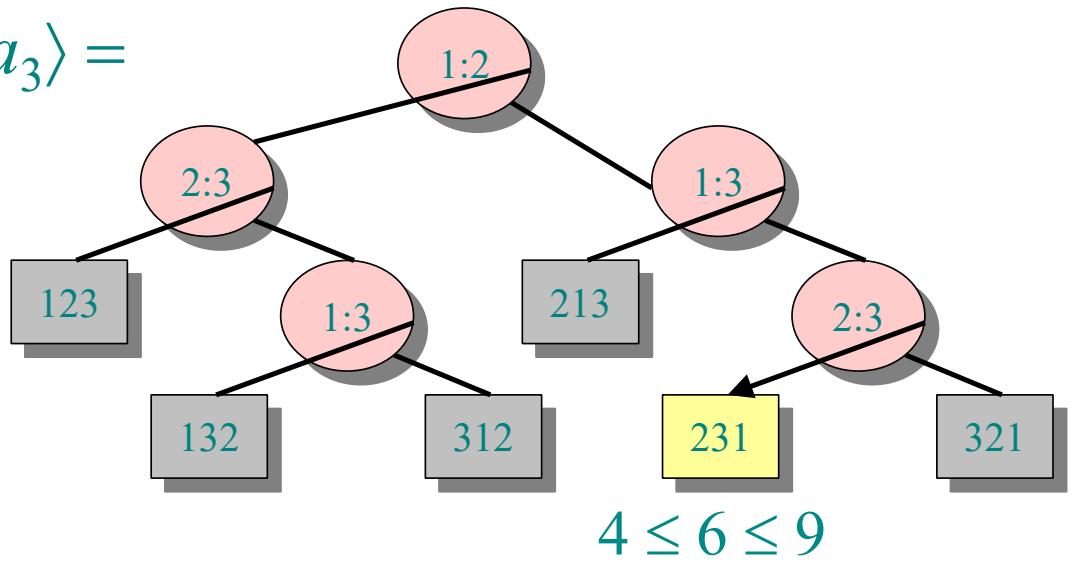
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ has been established.

DECISION-TREE MODEL

A decision tree can model the execution of any comparison sort:

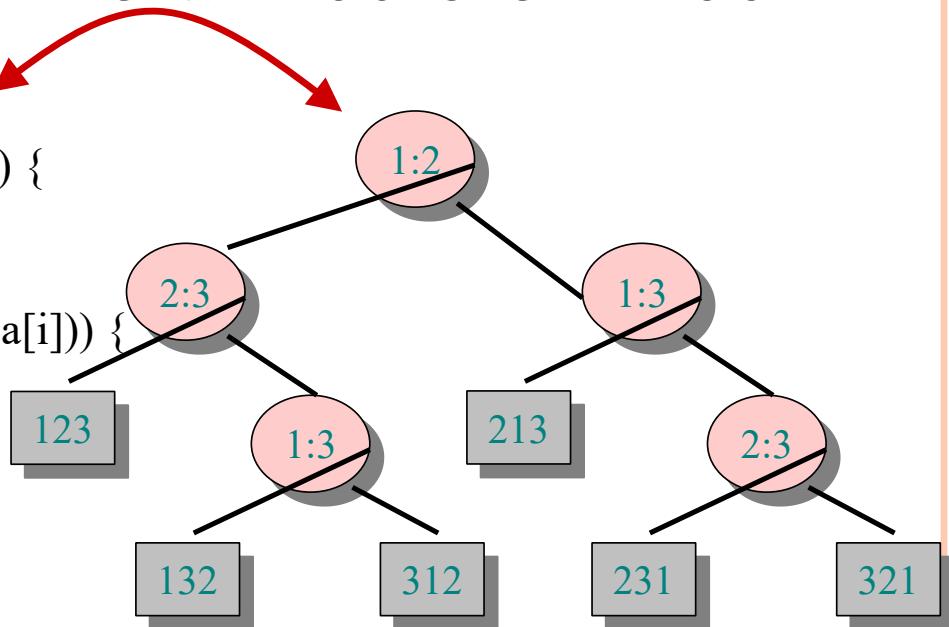
- One tree for each input size n .
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm $=$ the length of the path taken.
- Worst-case running time $=$ height of tree.



Any comparison sort

Can be turned into a Decision tree

```
class InsertionSortAlgorithm {  
    for (int i = 1; i < a.length; i++) {  
        int j = i;  
        while ((j > 0) && (a[j-1] > a[i])) {  
            a[j] = a[j-1];  
            j--; }  
        a[j] = B; } }
```



*What do the leaves represent?
How many leaves must there be?*



LOWER BOUND FOR DECISION-TREE SORTING

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

- *What's the minimum # of leaves?*
- *What's the maximum # of leaves of a binary tree of height h ?*

Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height- h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.



LOWER BOUND FOR DECISION-TREE SORTING

- So we have...

$$n! \leq 2^h$$

- Taking logarithms:

$$\lg(n!) \leq h$$

- Stirling's approximation tells us:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n$$

- Thus: $h \geq \lg\left(\frac{n}{e}\right)^n$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$



LOWER BOUND FOR COMPARISON SORTING

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. 



Sorting Lower Bound

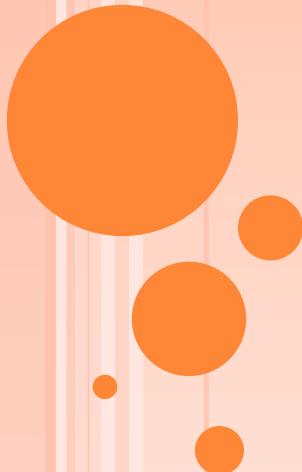
Is there a faster algorithm?

If different model of computation?

```
class InsertionSortAlgorithm {  
    for (int i = 1; i < a.length; i++) {  
        int j = i;  
        while ((j > 0) && (a[j-1] > a[i])) {  
            a[j] = a[j-1];  
            j--; }  
        a[j] = B; } }
```



LINEAR TIME SORTING ALGORITHMS



SORTING IN LINEAR TIME

Counting sort: No comparisons between elements.

- ***Input:*** $A[1 \dots n]$, where $A[j] \in \{1, 2, \dots, k\}$.
- ***Output:*** $B[1 \dots n]$, sorted.
- ***Auxiliary storage:*** $C[1 \dots k]$.

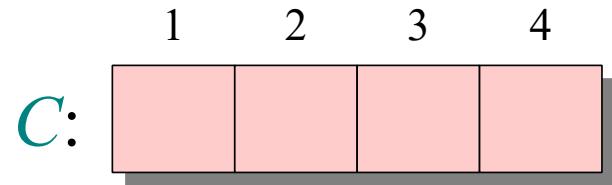
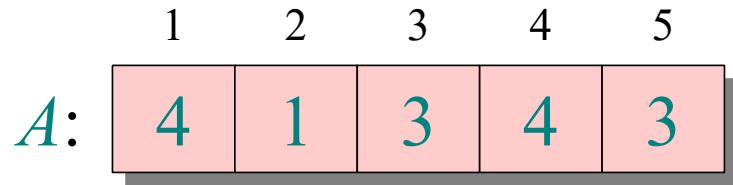


COUNTING SORT

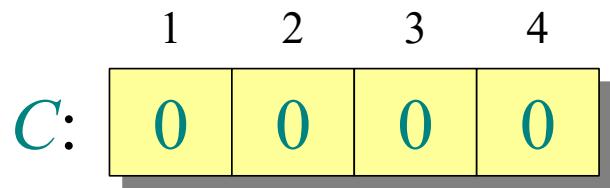
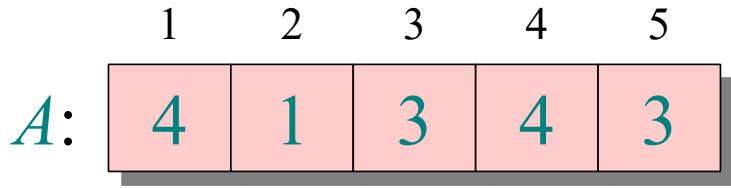
```
for  $i \leftarrow 1$  to  $k$ 
    do  $C[i] \leftarrow 0$ 
for  $j \leftarrow 1$  to  $n$ 
    do  $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright C[i] = |\{key = i\}|$ 
for  $i \leftarrow 2$  to  $k$ 
    do  $C[i] \leftarrow C[i] + C[i-1]$   $\triangleright C[i] = |\{key \leq i\}|$ 
for  $j \leftarrow n$  down to 1
    do  $B[C[A[j]]] \leftarrow A[j]$ 
         $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



COUNTING-SORT EXAMPLE



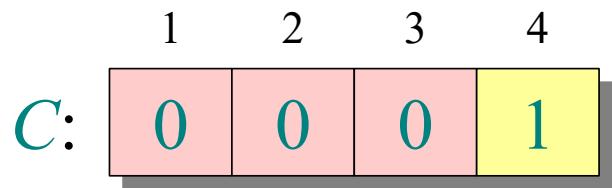
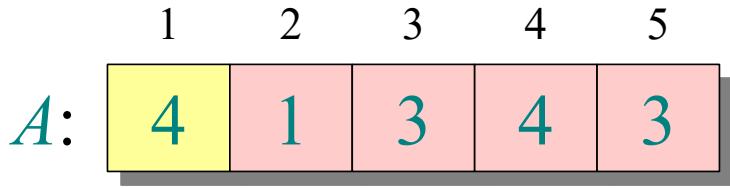
LOOP 1



```
for  $i \leftarrow 1$  to  $k$ 
  do  $C[i] \leftarrow 0$ 
```



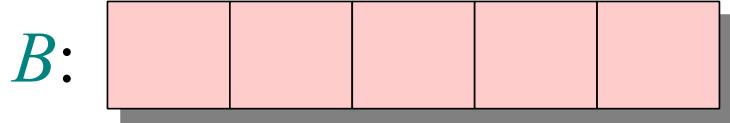
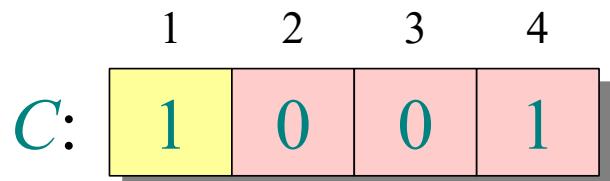
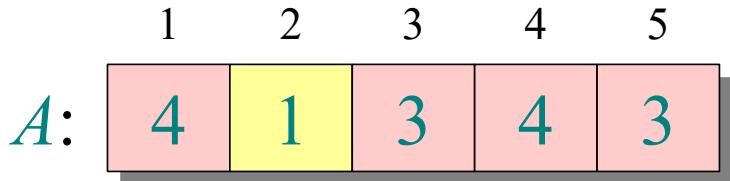
LOOP 2



for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{ \text{key} = i \}|$



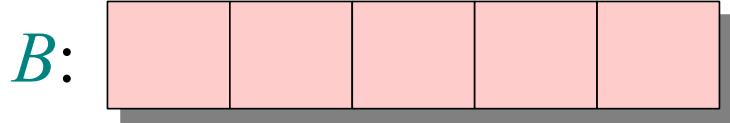
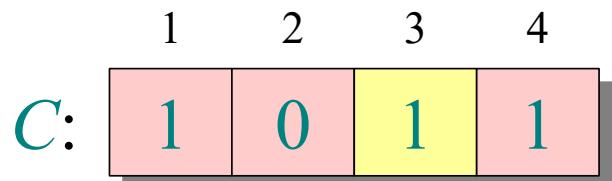
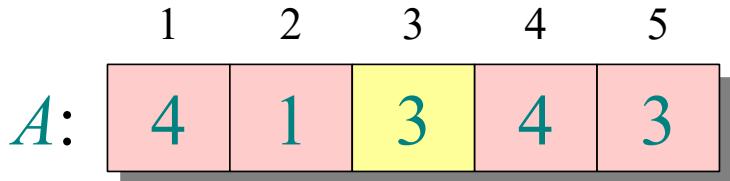
LOOP 2



for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{ \text{key} = i \}|$



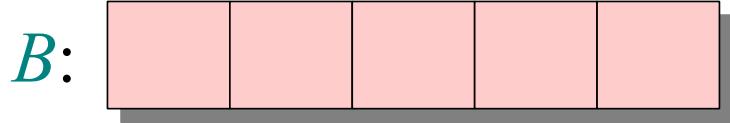
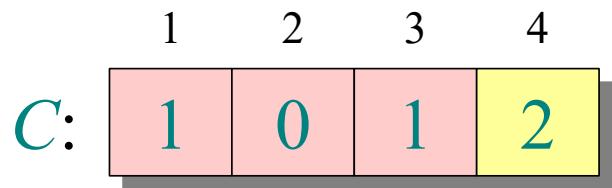
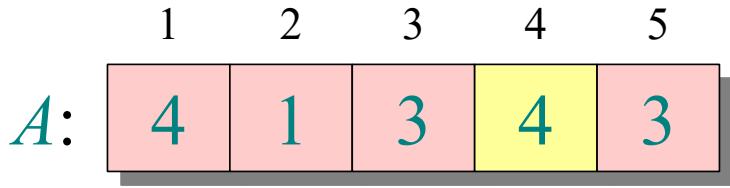
LOOP 2



for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{ \text{key} = i \}|$



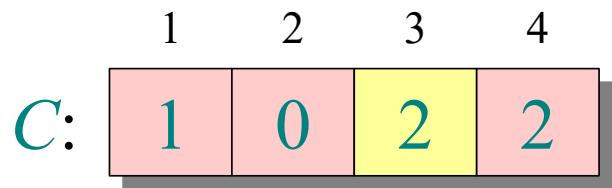
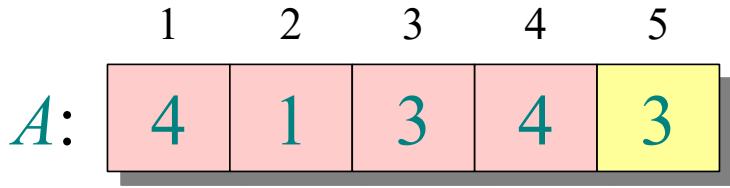
LOOP 2



for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{key = i\}|$



LOOP 2



for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i] = |\{ \text{key} = i \}|$



LOOP 3

	1	2	3	4	5
$A:$	4	1	3	4	3

	1	2	3	4
$C:$	1	0	2	2

$B:$					

$C':$	1	1	2	2

for $i \leftarrow 2$ **to** k
do $C[i] \leftarrow C[i] + C[i-1]$ $\blacktriangleright C[i] = |\{key \leq i\}|$



LOOP 3

	1	2	3	4	5
$A:$	4	1	3	4	3

	1	2	3	4
$C:$	1	0	2	2

$B:$					

$C':$	1	1	3	2

for $i \leftarrow 2$ **to** k
do $C[i] \leftarrow C[i] + C[i-1]$ $\blacktriangleright C[i] = |\{key \leq i\}|$



LOOP 3

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

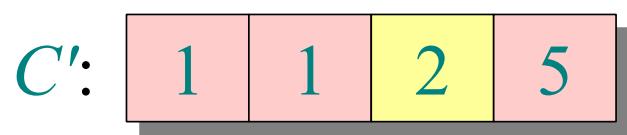
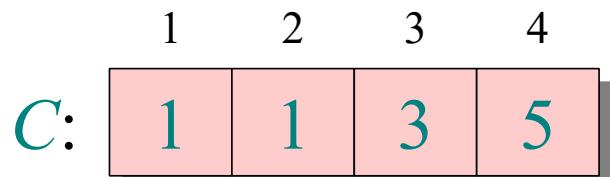
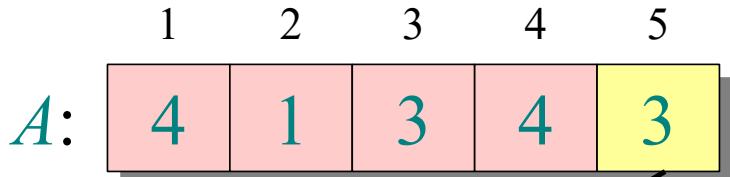
B:					

C':	1	1	3	5

for $i \leftarrow 2$ **to** k
do $C[i] \leftarrow C[i] + C[i-1]$ $\blacktriangleright C[i] = |\{key \leq i\}|$



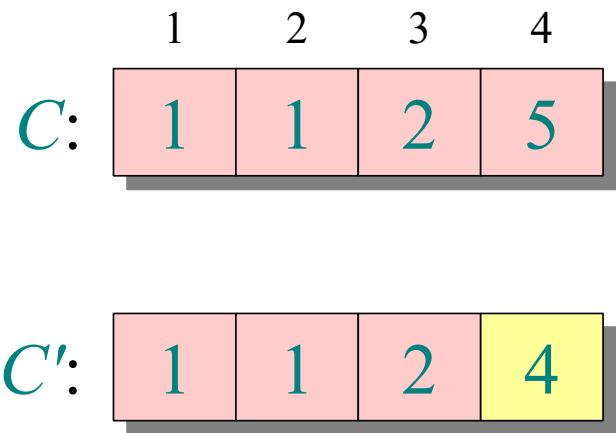
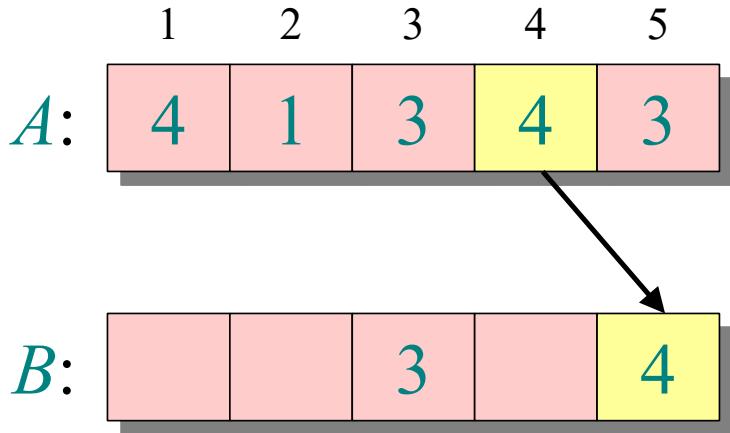
LOOP 4



for $j \leftarrow n$ down to 1
do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$



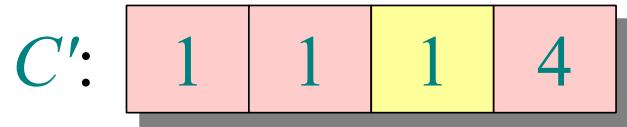
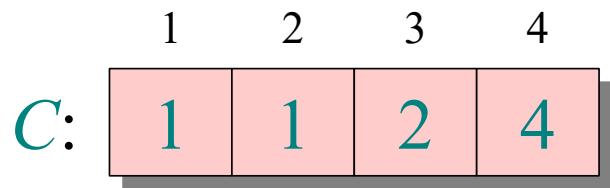
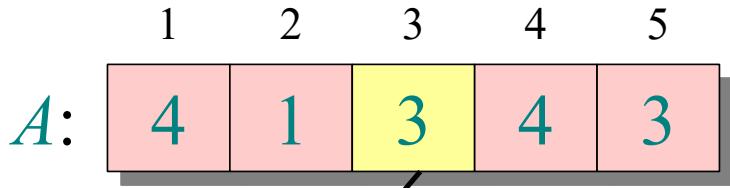
LOOP 4



for $j \leftarrow n$ **down to** 1
do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$



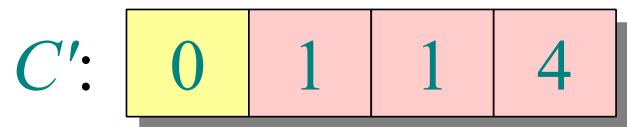
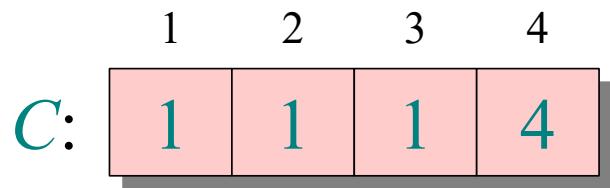
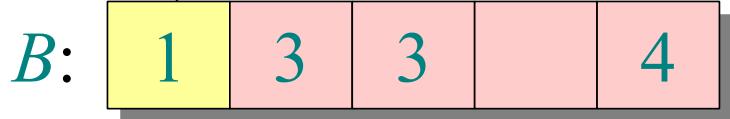
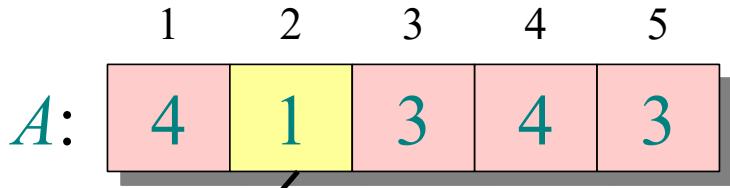
LOOP 4



```
for  $j \leftarrow n$  down to 1  
do  $B[C[A[j]]] \leftarrow A[j]$   
 $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



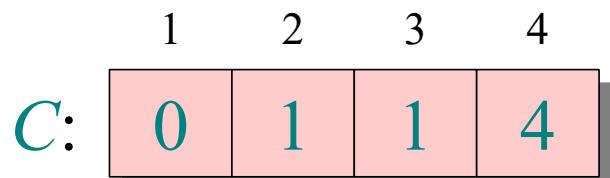
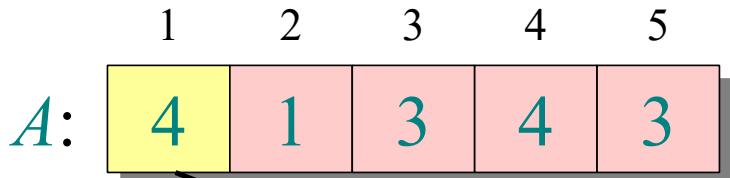
LOOP 4



```
for  $j \leftarrow n$  down to 1  
do  $B[C[A[j]]] \leftarrow A[j]$   
 $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



LOOP 4



for $j \leftarrow n$ down to 1
do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$



ANALYSIS

$\Theta(k)$

{ **for** $i \leftarrow 1$ **to** k
 do $C[i] \leftarrow 0$

$\Theta(n)$

{ **for** $j \leftarrow 1$ **to** n
 do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$

{ **for** $i \leftarrow 2$ **to** k
 do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$

{ **for** $j \leftarrow n$ **down to** 1
 do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$



RUNNING TIME

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

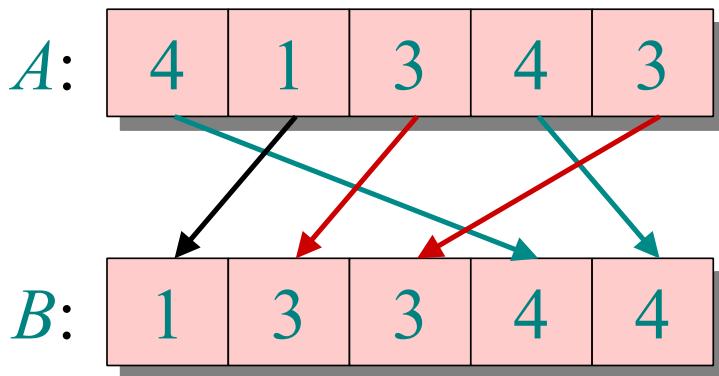
- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!



資料 stable 會 與 INTL

STABLE SORTING

Counting sort is a *stable* sort: it preserves the input order among equal elements.



Exercise: What other sorts have this property?



COUNTING SORT

- *Why don't we always use counting sort?*
 - Because it depends on range k of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
 - Answer: no, k too large ($2^{32} = 4,294,967,296$)



IMPROVEMENT BY RADIX SORT

- In fact, each number is composed of digits.
 - The range of each digit is limited.
 - We can run counting sort on each digit.

8 3 9

4 3 6

7 2 0

3 5 5

3 2 9

4 5 7

6 5 7

8 3 **9**

4 3 **6**

7 2 **0**

3 5 **5**

3 2 **9**

4 5 **7**

6 5 **7**

8 3 9

4 3 6

7 2 0

3 5 5

3 2 9

4 5 7

6 5 7

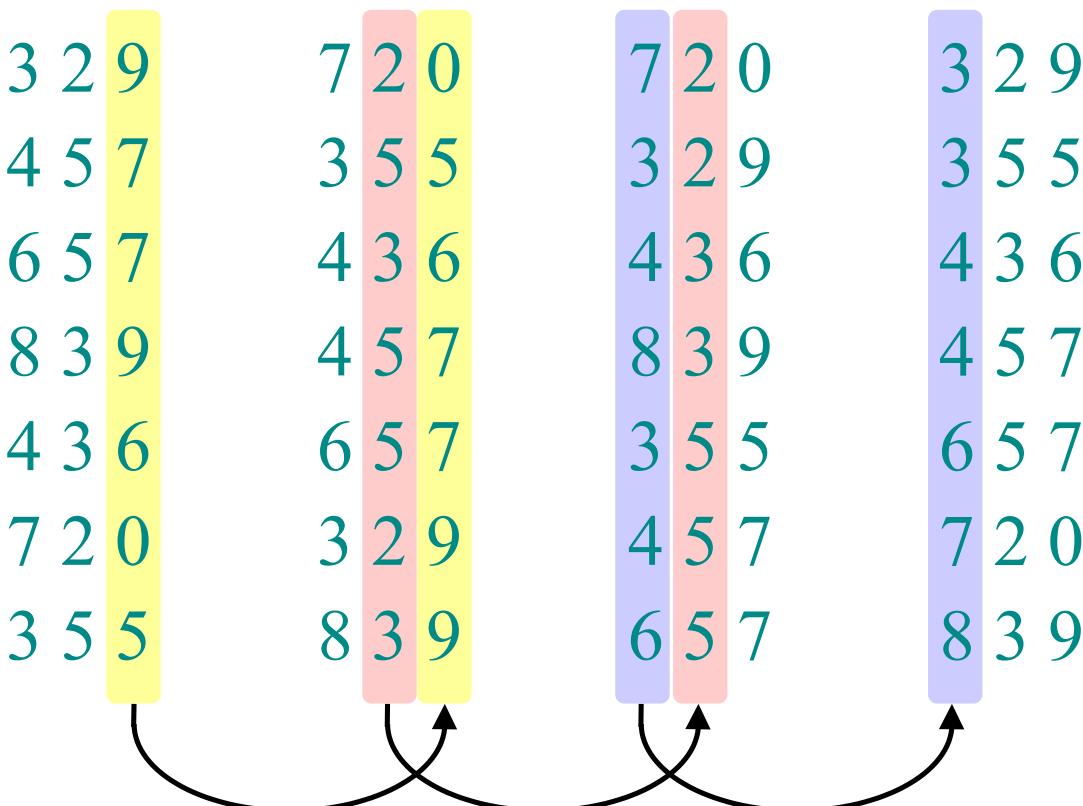


RADIX SORT

- *Origin*: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on *least-significant digit first* with auxiliary *stable* sort.



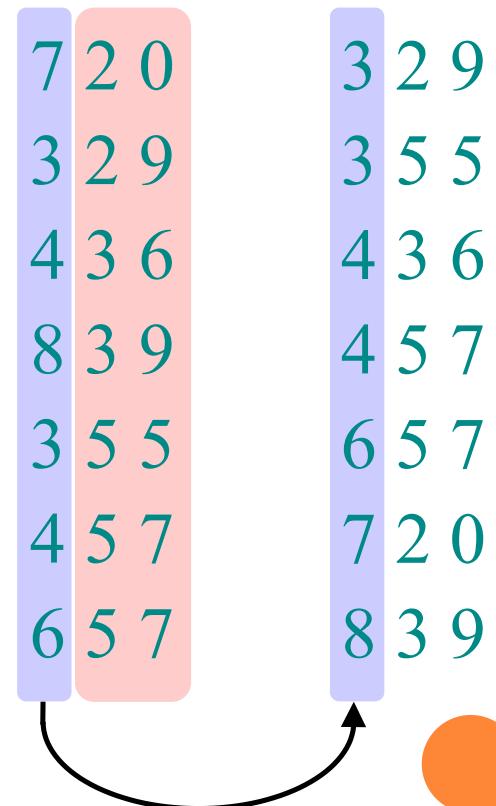
OPERATION OF RADIX SORT



CORRECTNESS OF RADIX SORT

Induction on digit position

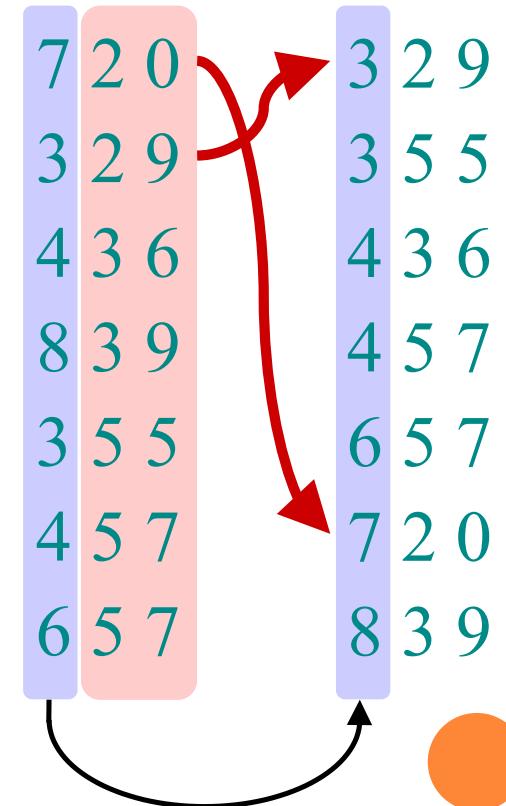
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t



CORRECTNESS OF RADIX SORT

Induction on digit position

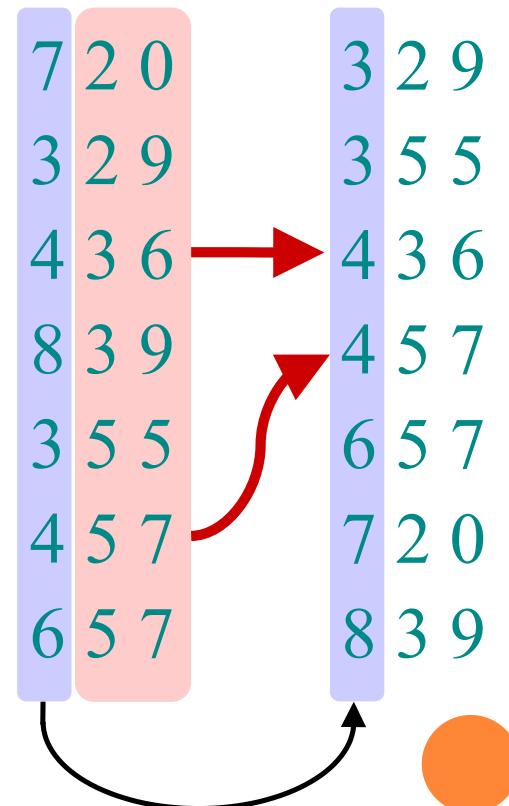
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
 - Two numbers that differ in digit t are correctly sorted.



CORRECTNESS OF RADIX SORT

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
 - Two numbers that differ in digit t are correctly sorted.
 - Two numbers equal in digit t are put in the same order as the input \Rightarrow correct order.

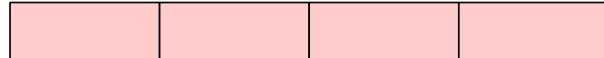


ANALYSIS OF RADIX SORT

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base- 2^r digits.

8 8 8 8

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base- 2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base- 2^{16} digits.

How many passes should we make?



考的考出大题

ANALYSIS (CONTINUED)

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b -bit word is broken into r -bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r}(n + 2^r)\right).$$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.



ANALYSIS (CONTINUED)

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r > n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.



CONCLUSIONS

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quick sort do at least $\lceil \lg 2000 \rceil = 11$ passes.

Downside: Can't sort in place using counting sort. Also, Unlike quick sort, radix sort displays little locality of reference, and thus a well-tuned quick sort fares better sometimes on modern processors, with steep memory hierarchies.



ORIGIN OF RADIX SORT

Hollerith's original 1889 patent alludes to a most-significant-digit-first radix sort:

“The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then resorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards.”

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

BUCKET SORT

- In Bucket sort, the range $[a,b]$ of input numbers is divided into m equal sized intervals, called buckets.
- Each element is placed in its appropriate bucket.
- If the numbers are **uniformly** divided in the range, the buckets can be expected to have roughly identical number of elements.
- Elements in the buckets are locally sorted.
- The expected time of this algorithm is $\Theta(n)$.



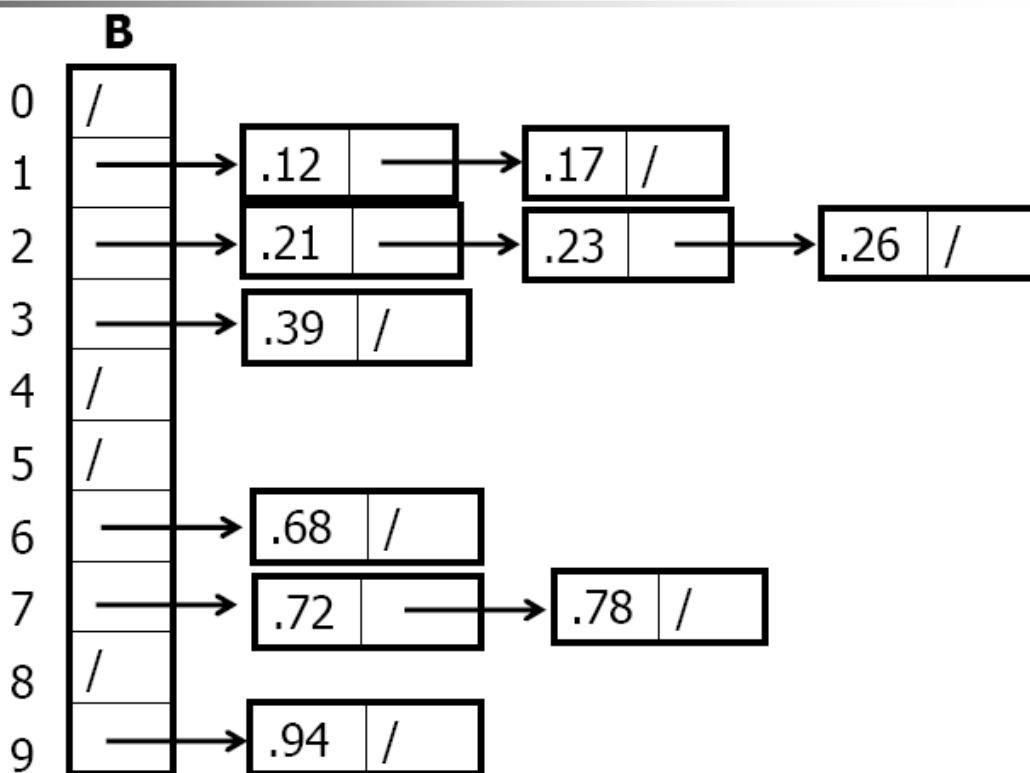
线-段代码，给化输入，考你输出

BUCKET-SORT(A)

- 1 $n \leftarrow \text{length } [A]$
- 2 **for** $i \leftarrow 1$ **to** n
 - 3 **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
 - 4 **for** $i \leftarrow 0$ **to** $n-1$
 - 5 **do** sort $B[i]$ with insertion sort
 - 6 Concatenate the lists $B[0], B[1], \dots, B[n-1]$ together in order

Bucket sort example

A	.78
1	.17
2	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68



ANALYSIS OF BUCKET SORT

- All operations except insertion sort take $O(n)$ time in the worst case.
- Let n_i be the random variable denoting the number of elements placed in bucket $B[i]$.
- Since the running time of insertion sort is $O(n^2)$, the running time of bucket sort is:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$



ANALYSIS OF BUCKET SORT

Taking expectations of both sides and using linearity of expectation, we have:

$$\begin{aligned} E[T(n)] &= E \left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \end{aligned}$$

分析
worst case



ANALYSIS OF BUCKET SORT

Define $X_{ij} = \mathbf{I}\{A[j] \text{ falls in bucket } i\}$ means the event that $A[j]$ falls in bucket i , then we have

$$n_i = \sum_{j=1}^n X_{ij}$$

By expand the square and regroup terms:

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] = E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij}X_{ik}\right] = E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} X_{ij}X_{ik}\right] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij}X_{ik}] \end{aligned}$$



ANALYSIS OF BUCKET SORT

Now we know:

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$
$$E[n_i^2] = \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij}X_{ik}]$$

Indicator random variable X_{ij} is 1 with probability $1/n$ and 0 otherwise, so:

$$E[X_{ij}^2] = 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}$$



ANALYSIS OF BUCKET SORT

Now we know:

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

$$E[n_i^2] = \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij}X_{ik}]$$

When $k \neq j$, the variable X_{ij} and X_{ik} are independent, and hence:

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$



ANALYSIS OF BUCKET SORT

Now we know:

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$
$$E[n_i^2] = \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij}X_{ik}]$$

By substituting these two expected values in equations above, we obtain:

$$E[n_i^2] = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n}$$

$$E[T(n)] = \Theta(n) + n \cdot O\left(2 - \frac{1}{n}\right) = \Theta(n)$$

Thus, the entire bucket sort algorithm runs in linear expected time.

Hash Tables



Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

Outline

- Dynamic Sets
- Hash Tables
 - Direct-access tables
 - Resolving collisions by chaining
 - Choosing hash functions
 - Open addressing

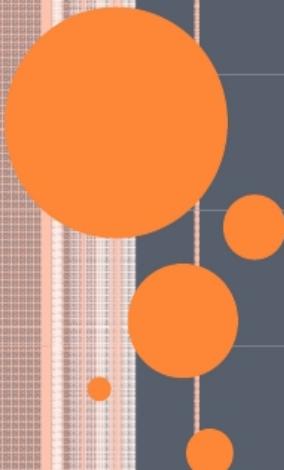
考重點

① 寫出哈希過程

② 設計哈希函數



Dynamic Sets



Dynamic Sets

- Structures for *dynamic sets*
 - Elements have a *key* and *satellite data*
 - Dynamic sets may support *queries* such as:
 - $\text{Search}(S, k)$, $\text{Minimum}(S)$, $\text{Maximum}(S)$, $\text{Successor}(S, x)$, $\text{Predecessor}(S, x)$
 - They may also support *modifying operations* like:
 - $\text{Insert}(S, x)$, $\text{Delete}(S, x)$

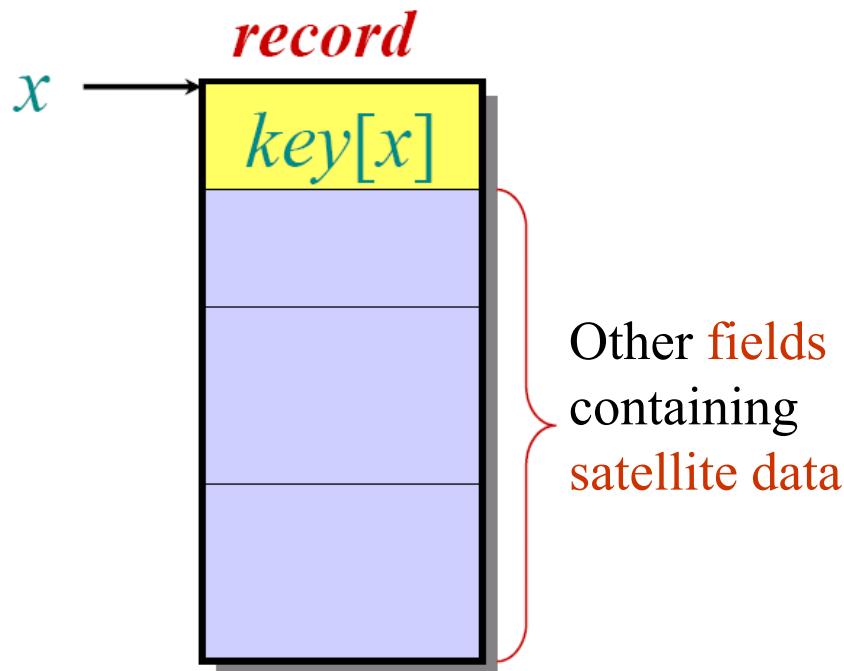


Hash Tables



Symbol-table problem

Symbol table S holding n *records*:



Operations on S :

- INSERT (S, x)
- DELETE (S, x)
- SEARCH (S, k)

How should the data structure S be organized?

Direct-access table

IDEA: Suppose that the keys are drawn from the set $U \subseteq \{0, 1, \dots, m-1\}$, and keys are distinct. Set up an array $T[0 \dots m-1]$:

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } \text{key}[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

Then, operations take $\Theta(1)$ time.



Direct-access table

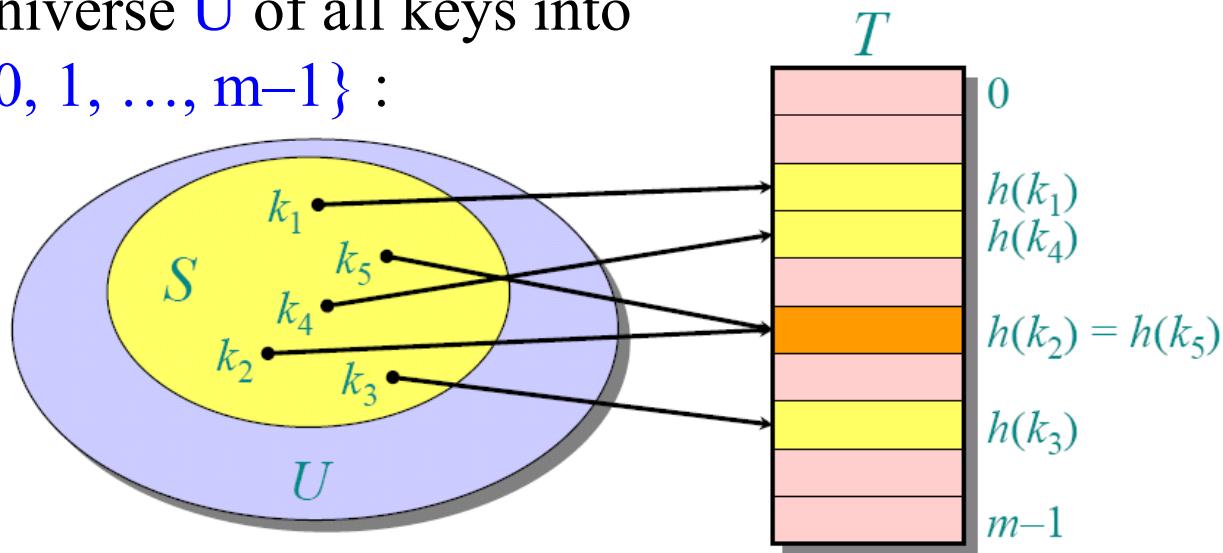
Problem:

1. The range of keys can be large:
 - 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
 - character strings (even larger!).
2. The set K of keys actually stored maybe $\ll U$,
Most of the space allocated for T is wasted.



Hash Tables

Solution: Use a *hash function* h to map the universe U of all keys into $\{0, 1, \dots, m-1\}$:

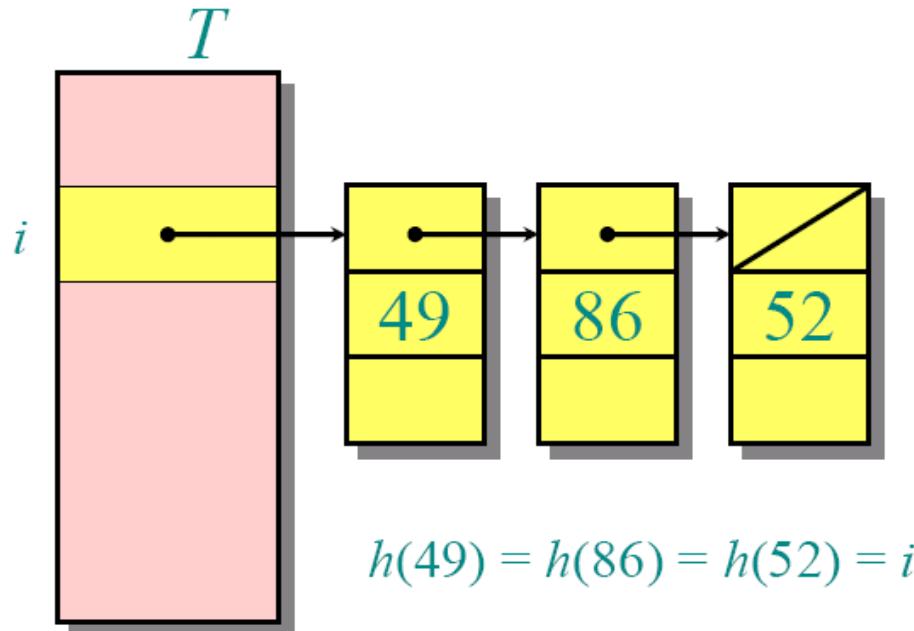


When a record to be inserted maps to an already occupied slot in T , a *collision* occurs.



Resolving collisions by chaining

- Link records in the same slot into a list.



Worst case:

- Every key hashes to the same slot.
- Access time = $\Theta(n)$ if $|S| = n$

Average-case analysis of chaining

We make the assumption of *simple uniform hashing*:

- Each key $k \in S$ is equally likely to be hashed to any slot of table T , independent of where other keys are hashed.

Let n be the number of keys in the table, and let m be the number of slots.

Define the *load factor* of T to be

$$\alpha = n/m$$

=average number of keys per slot.

Search cost

证明.

The expected time for an *unsuccessful* search for a record with a given key is
 $= \Theta(1 + \alpha)$.

成功的 search 也是 $\Theta(1 + \alpha)$

Search cost

The expected time for an *unsuccessful* search for a record with a given key is

$$= \Theta(1 + \alpha).$$

*apply hash function
and access slot*

*search
the list*



Search cost

The expected time for an *unsuccessful* search for a record with a given key is

$$= \Theta(1 + \alpha).$$

*apply hash function
and access slot*

*search
the list*

Expected search time = $\Theta(1)$ if $\alpha = O(1)$,
or equivalently, if $n = O(m)$.



Search cost

The expected time for an *unsuccessful* search for a record with a given key is

$$= \Theta(1 + \alpha).$$

*apply hash function
and access slot*

*search
the list*

Expected search time = $\Theta(1)$ if $\alpha = O(1)$,
or equivalently, if $n = O(m)$.

A *successful* search has same asymptotic bound, but a rigorous argument is a little more complicated. (See textbook.)



Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.



Division method

Assume all keys are integers, and define

$$h(k) = k \bmod m.$$

Deficiency: Don't pick an m that has a small remainder d . A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: If $m = 2^r$, then the hash doesn't even depend on all the bits of k :

- If $k = 1011000111\underbrace{011010}_2$ and $r = 6$, then $h(k) = 011010_2$.



Division method (continued)

$$h(k) = k \bmod m.$$

Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

Annoyance:

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.



Division method (continued)

- Example
 - If $n=2000$, with 8-bit keys. The desired number of searched keys in an unsuccessful search is about 3.



Division method (continued)

- Example
 - If $n=2000$, with 8-bit keys. The desired number of searched keys in an unsuccessful search is about 3.
 - $m=701$
 - Approximately $=2000/3$
 - A prime not close to any 2^r
 - $h(k)=k \bmod 701$



Multiplication method

Assume that all keys are integers, $m = 2^r$, and our computer has w -bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w-r),$$

where **rsh** is the “bitwise right-shift” operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

- Don’t pick A too close to 2^{w-1} or 2^w .
- Multiplication modulo 2^w is fast compared to division.
- The **rsh** operator is fast.

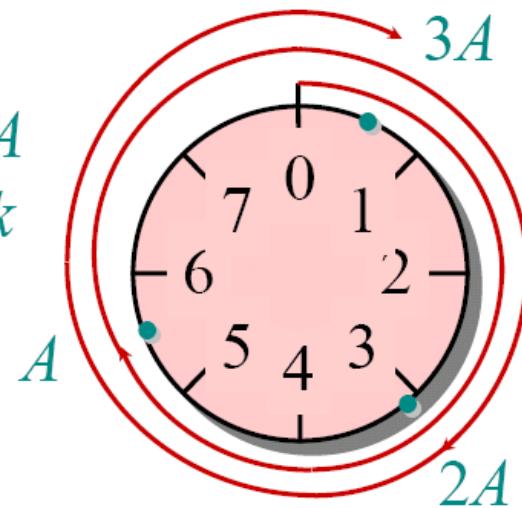


Multiplication method example

(逐分段(可能))

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$ -bit words:

$$\begin{array}{r} 1011001 \\ \times 1101011 \\ \hline 1001010 \underbrace{011}_{h(k)} 0011 \end{array} = A \\ = k$$



Modular wheel

Resolving collisions by open addressing

No storage is used outside of the hash table itself.

- Save pointers for a larger number of slots
- The hash function depends on both the key and probe number:

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

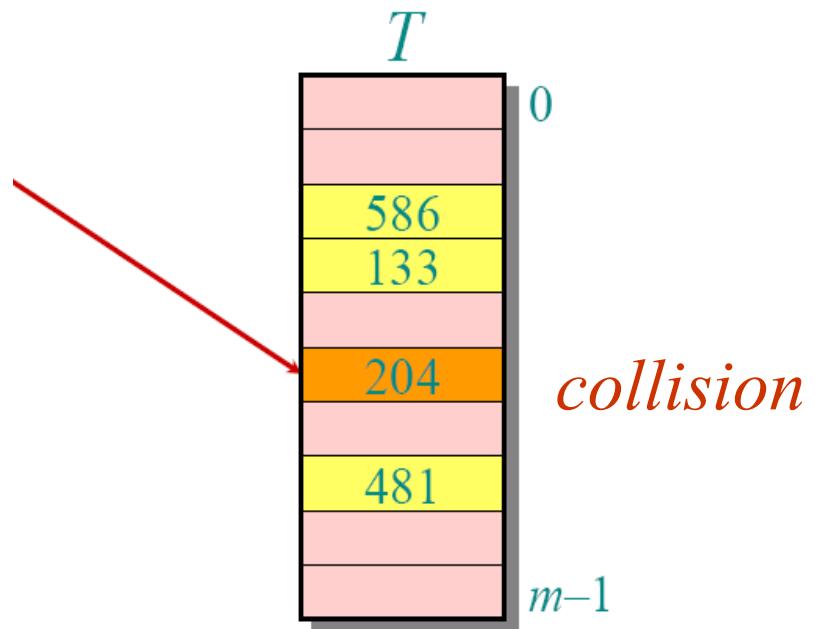
- The probe sequence $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ should be a permutation of $\{0, 1, \dots, m-1\}$.
- Insertion systematically probes the table until an empty slot is found.



Example of open addressing

Insert key $k=496$:

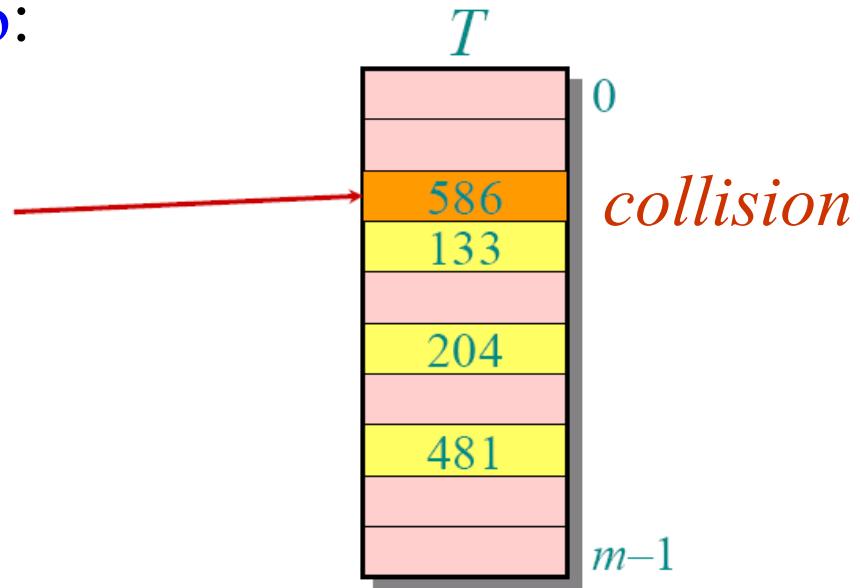
0. Probe $h(496, 0)$



Example of open addressing

Insert key $k=496$:

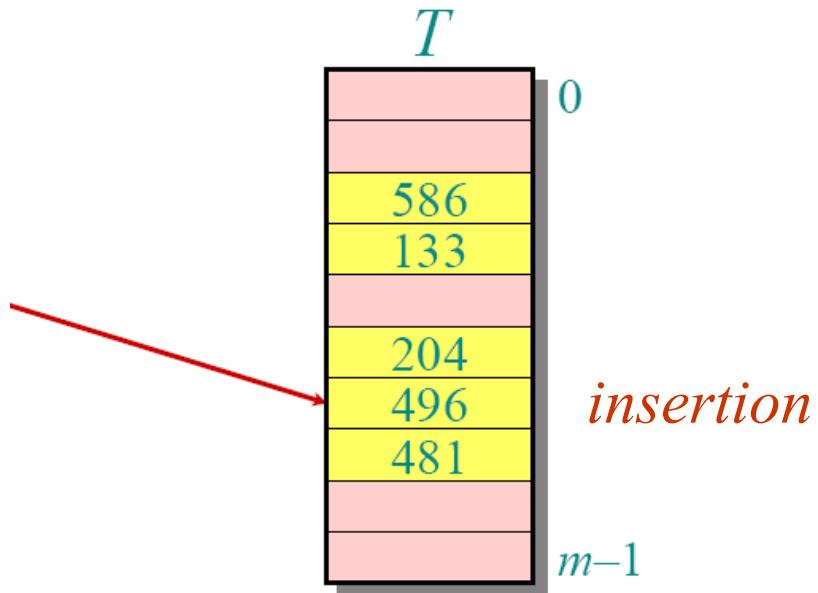
0. Probe $h(496,0)$
1. Probe $h(496,1)$



Example of open addressing

Insert key $k=496$:

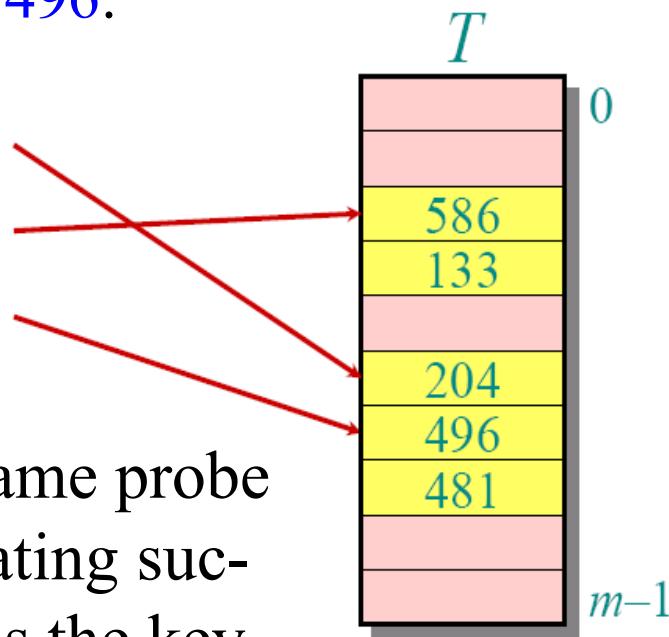
0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Example of open addressing

Search for key $k=496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.

Probing strategies

- The assumption of uniform hashing:
- Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \dots, m-1 \rangle$ as its probe sequence.
- Probing functions:
 - Must guarantee the $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ is a permutation of $\langle 0, 1, \dots, m-1 \rangle$ for each key.
 - Try to fulfill the assumption of uniform hashing



Probing strategies

Linear probing:

Given an ordinary hash function $h'(k)$, linear probing uses the hash function

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

This method, though simple, suffers from *primary clustering*, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.



Probing strategies

Quadratic probing:

Quadratic probing uses the hash function

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \text{ mod } m.$$

Where $h'(k)$ is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants.

To make full use of the hash table, the values of c_1 , c_2 , and m are constrained.

Much better than linear probing, still suffers from a milder form of clustering, called

Secondary clustering.



Probing strategies

Double hashing:

Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

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

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m . One way is to make m a power of 2 and design $h_2(k)$ to produce only odd numbers.



Probing strategies

- Linear probing
 - m different probe sequence
- Quadratic probing
 - m different probe sequence
- Double hash
 - m^2 , close to the “ideal”

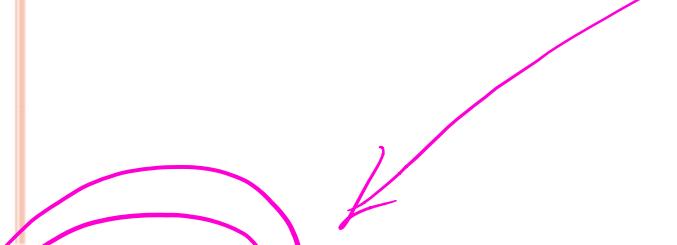


Analysis of open addressing

We make the assumption of *uniform hashing*:

- Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.

Theorem. Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.



Proof of the theorem

Proof.

- At least one probe is always necessary.
- With probability n/m , the first probe hits an occupied slot, and a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, and a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe

Observe that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \dots, n$.



Proof (continued)

Therefore, the expected number of probes is

$$\begin{aligned} & 1 + \frac{n}{m} \left(1 + \frac{n-1}{m-1} \left(1 + \frac{n-2}{m-2} \left(\dots \left(1 + \frac{1}{m-n+1} \right) \dots \right) \right) \right) \\ & \leq 1 + \alpha(1 + \alpha(1 + \alpha(\dots(1 + \alpha)\dots))) \\ & \leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots \\ & = \sum_{i=0}^{\infty} \alpha^i \\ & = \frac{1}{1 - \alpha}. \quad \blacksquare \end{aligned}$$

The textbook has a more rigorous proof and an analysis of successful searches.



Analysis of open addressing

- **Corollary** Inserting an element into an open-address hash table with load factor α requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.



Implications of the theorem

- If α is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is $1/(1-0.5) = 2$.
- If the table is 90% full, then the expected number of probes is $1/(1-0.9) = 10$.



(送分习题)

Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

$h(10,0)=10$ 

$h(22,0)=0$ 

$h(31,0)=9$ 

$h(4,0)=4$ 

$h(15,0)=4$
conflict! 

$h(15,1)=5$ 



Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

$h(28,0)=6$

22				4	15	28			31	10
----	--	--	--	---	----	----	--	--	----	----

$h(17,0)=6$
conflict!

22				4	15	28			31	10
----	--	--	--	---	----	----	--	--	----	----

$h(17,1)=7$

22				4	15	28	17		31	10
----	--	--	--	---	----	----	----	--	----	----

$h(88,0)=0$
conflict!

22				4	15	28	17		31	10
----	--	--	--	---	----	----	----	--	----	----

$h(88,1)=1$

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$h(59,0)=4$
conflict!

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$h(59,1)=5$
conflict!

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

$h(59,2)=6$
conflict!

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$h(59,3)=7$
conflict!

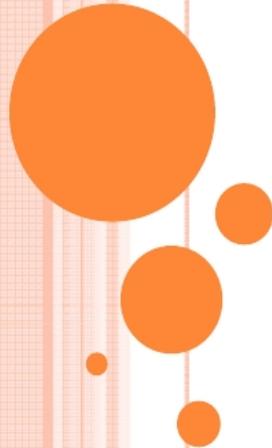
22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$h(59,4)=8$

22	88			4	15	28	17	59	31	10
----	----	--	--	---	----	----	----	----	----	----

数据结构基本操作：变种

Binary Search Trees



Prof. Zhenyu He

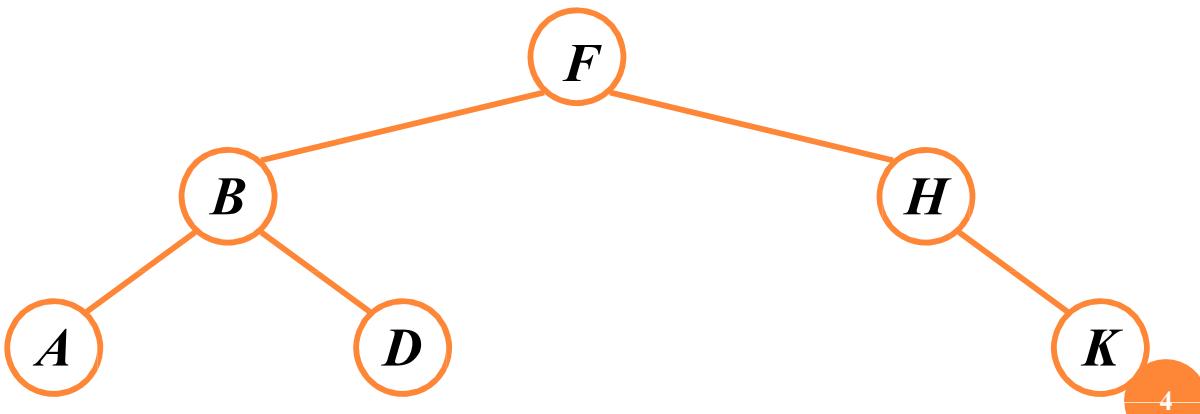
Harbin Institute of Technology, Shenzhen

BINARY SEARCH TREES

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
 - key*: an identifying field inducing a total ordering
 - left*: pointer to a left child (may be NULL)
 - right*: pointer to a right child (may be NULL)
 - p*: pointer to a parent node (NULL for root)

BINARY SEARCH TREES

- BST property:
 $\text{key}[\text{left}(x)] \leq \text{key}[x] \leq \text{key}[\text{right}(x)]$
- Example:



INORDER TREE WALK (TRAVERSAL)

- *What does the following code do?*

```
TreeWalk(x) TreeWalk(left[x]); print(x);  
    TreeWalk(right[x]);
```

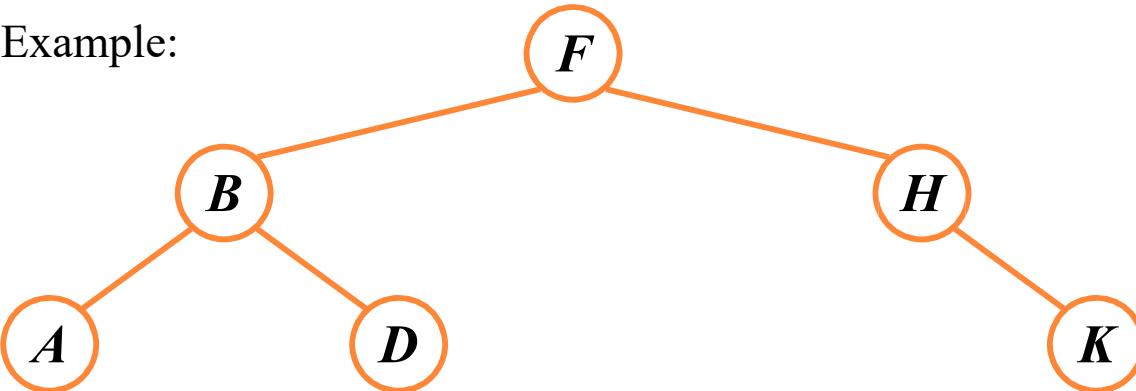
- A: prints elements in sorted (increasing) order
- This is called an *inorder tree walk*

Preorder tree walk: print root, then left, then right

Postorder tree walk: print left, then right, then root

INORDER TREE WALK

- Example:



- Output: A B D F H K
- *How long will a tree walk take?*

Theorem 12.1

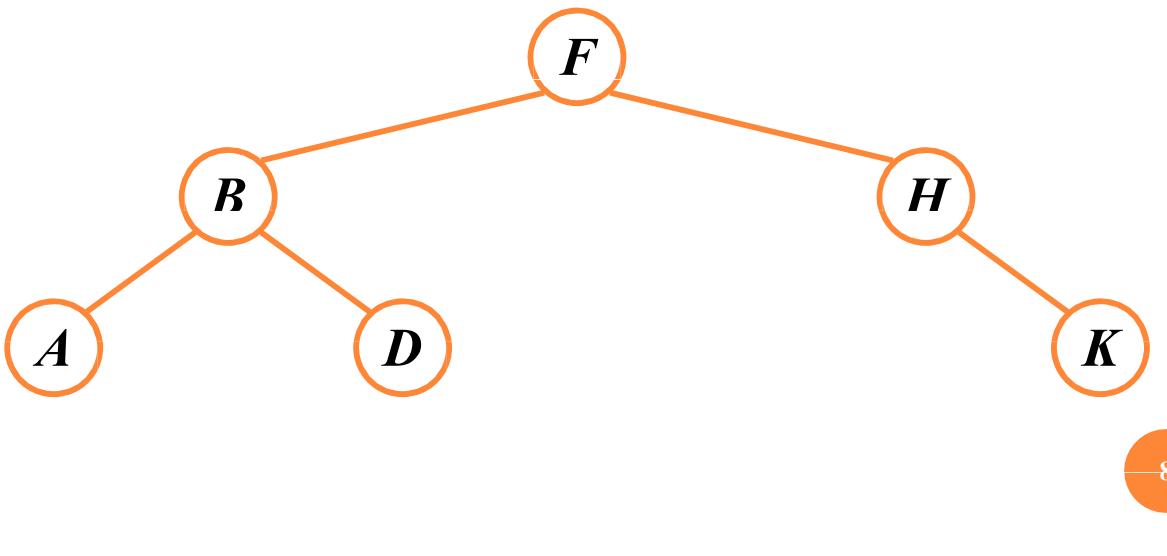
OPERATIONS ON BSTS: SEARCH

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
TreeSearch(x, k)
if (x = NULL or      k = key[x]) return x;
if (k < key[x])
return TreeSearch(left[x], k); else
return TreeSearch(right[x], k);
```

BST SEARCH: EXAMPLE

- Search for D and C :



OPERATIONS ON BSTS: SEARCH

- Here's another function that does the same:

```
TreeSearch(x, k)
while (x != NULL and k != key[x]) if
    (k < key[x])
    x = left[x]; else
    x = right[x]; return x;
```

- *Which of these two functions is more efficient?*



OPERATIONS OF BSTS: INSERT

- Adds an element x to the tree so that the binary search tree property continues to hold

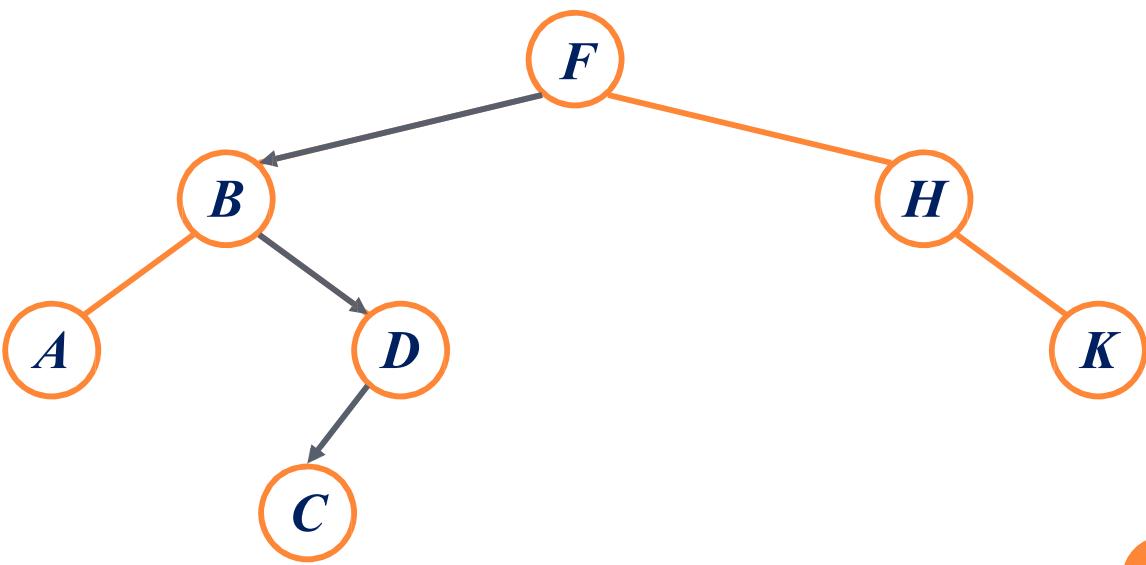
- The basic algorithm

Like the search procedure above

Insert x in place of NULL

BST INSERT: EXAMPLE

- Example: Insert C



11

BST SEARCH/INSERT: RUNNING TIME

- The height of a binary search tree is h
- What is the running time of TreeSearch() or TreeInsert()?

$O(h)$

- What is the height of a binary search tree?

Worst case: $h = O(n)$ when tree
is just a linear string of left or right children

Minimum of BST

- TREE-Minimum(x)
 - 1 while left(x)≠NIL
 - 2 do $x \leftarrow \text{left}[x]$
 - 3 Return x

Maximum of BST

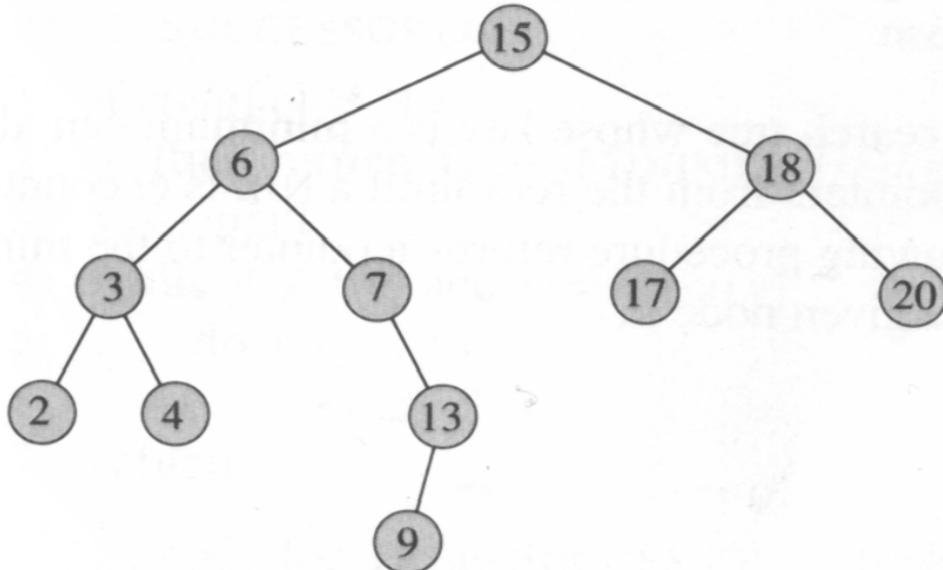
- TREE-Maximum(x)
 - 1 while right(x)≠NIL
 - 2 do $x \leftarrow \text{right}[x]$
 - 3 Return x

BST OPERATIONS: SUCCESSOR

- The successor of the current node is the one in the in-order tree walk.
- Two cases:
 - x has a right subtree: successor is minimum node in right subtree
 - x has no right subtree: successor is lowest ancestor of x whose left child is also one ancestor of x
(every node is its own ancestor)
Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.

BST OPERATIONS: SUCCESSOR

- What is the successor of node 3? 15? 13? 17?



- How about predecessor?

(这个可能)

BST OPERATIONS: DELETE

- Deletion is a bit tricky

- 3 cases:

x has no children:

- Remove x

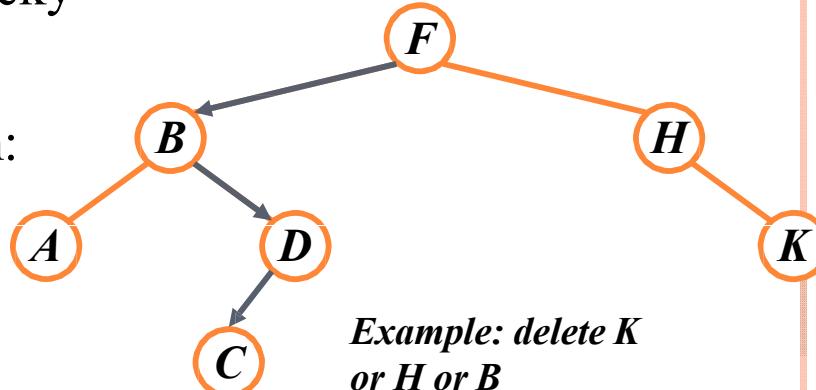
x has one child:

- Splice out x

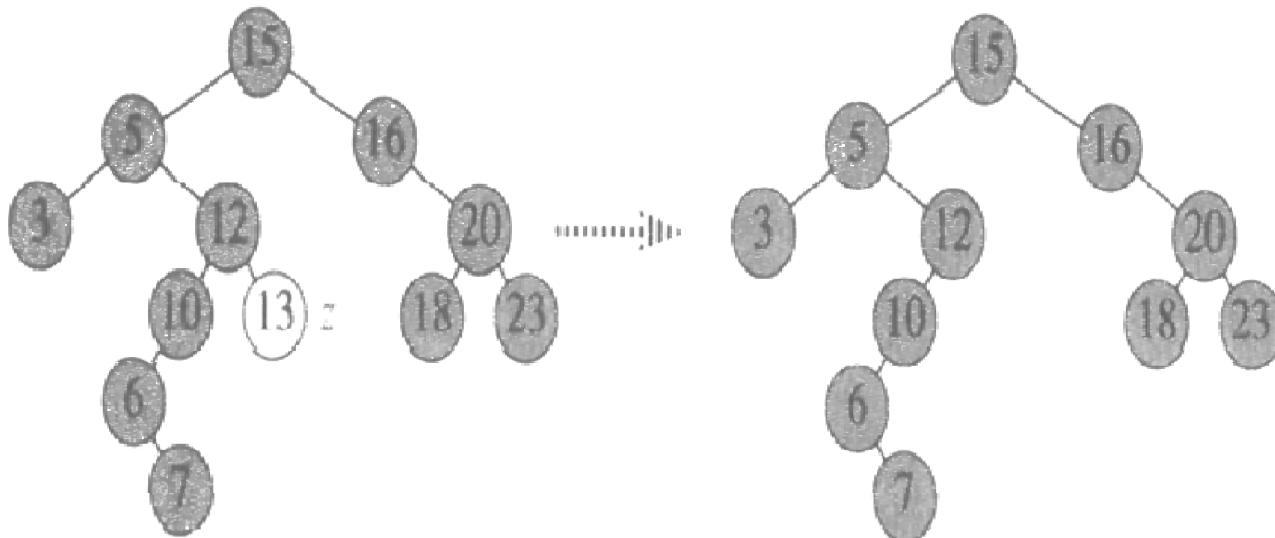
x has two children:

- Swap x with successor

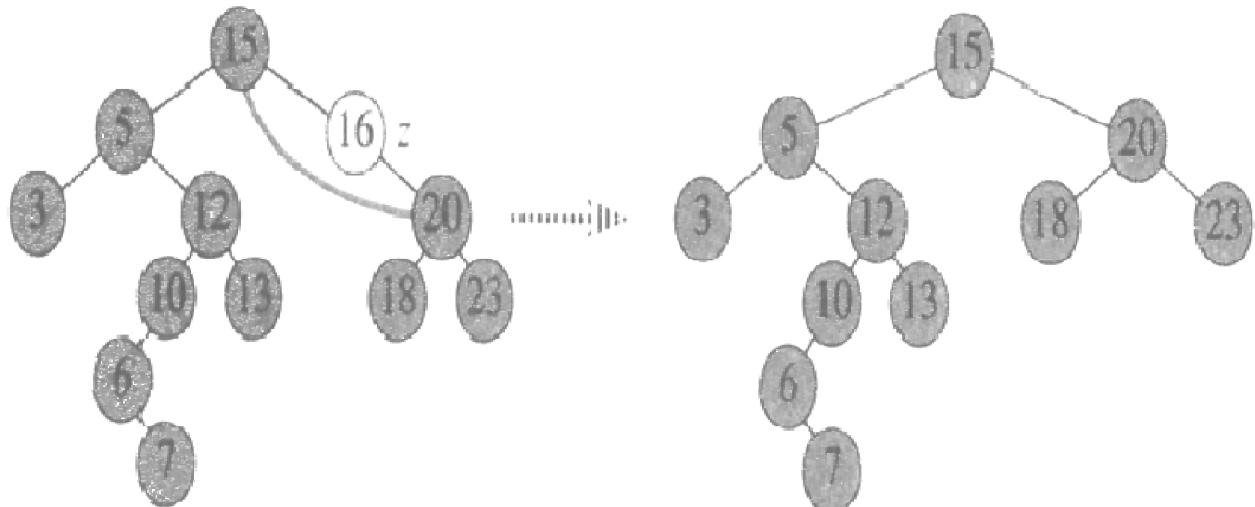
- Perform case 1 or 2 to delete it



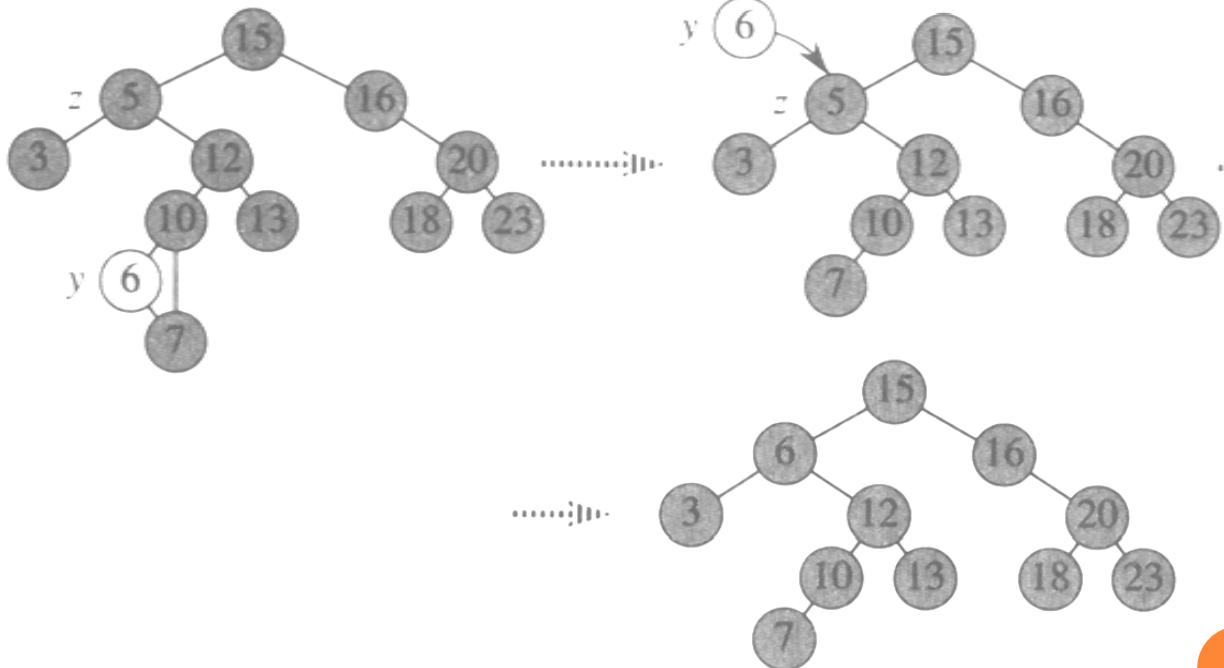
Z HAS NO CHILDREN



Z HAS ONLY ONE CHILD



Z HAS TWO CHILDREN



BST OPERATIONS: DELETE

- *Why will case 2 always go to case 0 or case 1?*

When x has 2 children, its successor is the minimum in its right subtree.

- *Could we swap x with predecessor instead of successor?*

Yes.

SORTING WITH BINARY SEARCH TREES

- Can you come out an algorithm for sorting by BST?

3 1 8 2 6 7 5

SORTING WITH BINARY SEARCH TREES

- Informal code for sorting array A of length n :

BSTSort(A)

for i=1 to n

TreeInsert(A[i]);

与未排序的区隔

InorderTreeWalk(root);

- *What will be the running time in the*

Worst case?

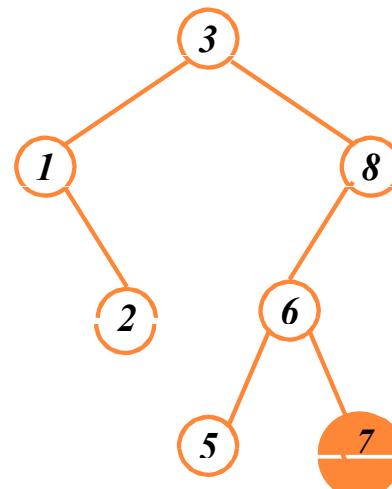
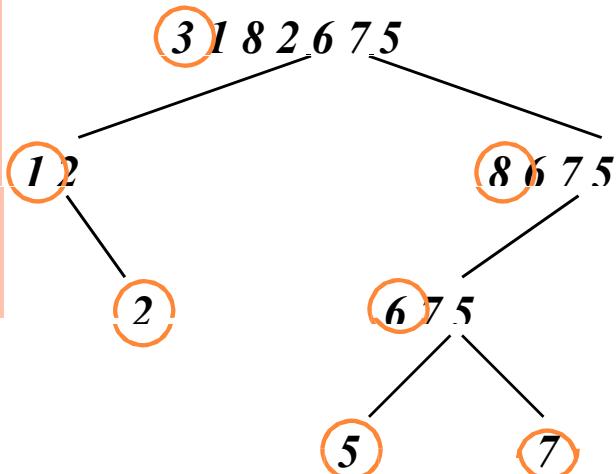
Best case?

Average case?

SORTING WITH BSTS

- Average case analysis
It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
    InorderTreeWalk(root);
```



SORTING WITH BSTS

- Inserted nodes are similar to partition pivot used in quicksort, but in a different order.

BST does not partition immediately after picking the inserted node.

SORTING WITH BSTS

- Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \lg n)$
- Which do you think is better, quicksort or BSTSort? Why?

SORTING WITH BSTS

- Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \lg n)$
- Which do you think is better, quicksort or BSTSort? Why?

Quicksort

Sorts in place

Doesn't need to build data structure

MORE BST OPERATIONS

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*

Insert

Minimum

Randomly Built Binary Search Tree

DEFINITION

- A randomly built binary search tree on n keys is one that arises from inserting the keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely.

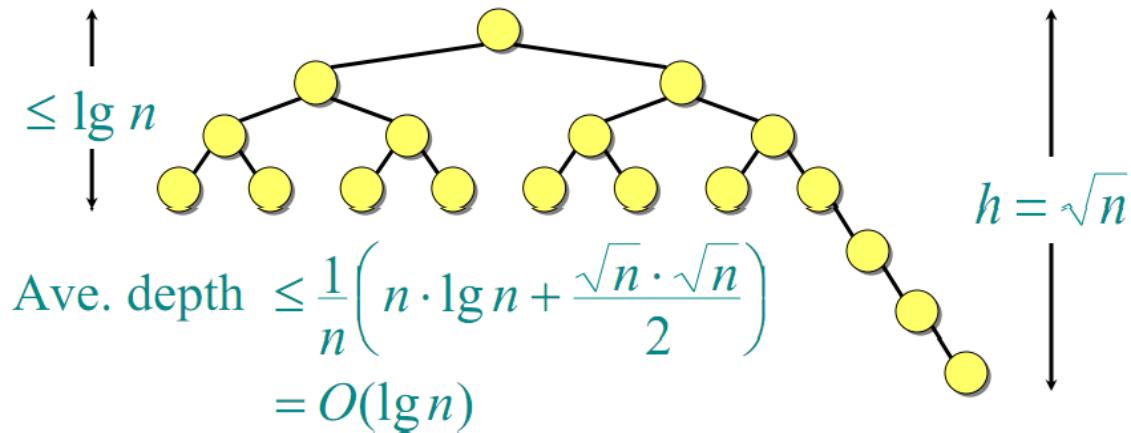
RANDOMLY BUILT BINARY SEARCH TREE

- **Theorem:** The average height of a randomly-built binary search tree of n distinct keys is $O(\lg n)$
- **Corollary:** The dynamic operations Successor, Predecessor, Search, Min, Max, Insert, and Delete all have $O(\lg n)$ average complexity on randomly-built binary search trees.

EXPECTED TREE HEIGHT

- Average node depth of a randomly built BST = $O(\lg n)$ does not necessarily mean that its expected height is also $O(\lg n)$ (although it is).

Example.



HEIGHT OF A RANDOMLY BUILT BINARY SEARCH TREE

Outline of the analysis:

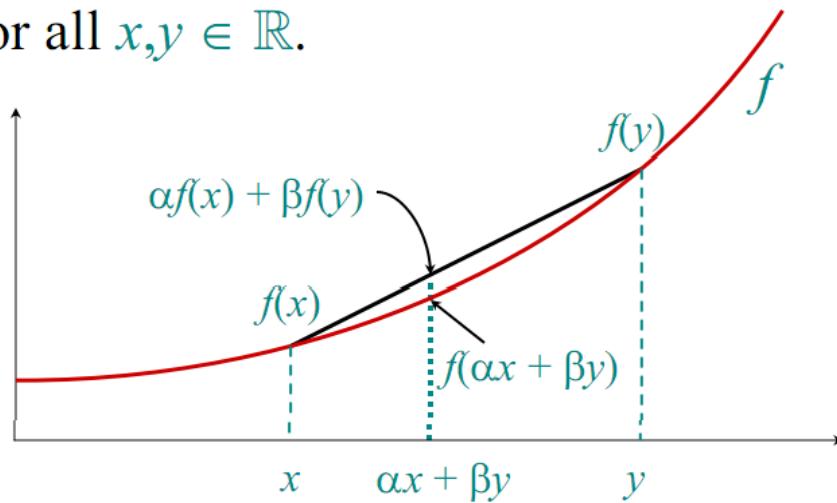
- Prove *Jensen's inequality*, which says that $f(E[X]) \leq E[f(X)]$ for any convex function f and random variable X .
- Analyze the *exponential height* of a randomly built BST on n nodes, which is the random variable $Y_n = 2^{X_n}$, where X_n is the random variable denoting the height of the BST.
- Prove that $2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n] = O(n^3)$, and hence that $E[X_n] = O(\lg n)$.

CONVEX FUNCTIONS

A function $f: \mathbb{R} \rightarrow \mathbb{R}$ is **convex** if for all $\alpha, \beta \geq 0$ such that $\alpha + \beta = 1$, we have

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$

for all $x, y \in \mathbb{R}$.



CONVEXITY LEMMA

Lemma. Let $f: \mathbb{R} \rightarrow \mathbb{R}$ be a convex function, and let $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$ be a set of nonnegative constants such that $\sum_k \alpha_k = 1$. Then, for any set $\{x_1, x_2, \dots, x_n\}$ of real numbers, we have

$$f\left(\sum_{k=1}^n \alpha_k x_k\right) \leq \sum_{k=1}^n \alpha_k f(x_k).$$

Proof. By induction on n . For $n = 1$, we have $\alpha_1 = 1$, and hence $f(\alpha_1 x_1) \leq \alpha_1 f(x_1)$ trivially.

PROOF (CONTINUED)

Inductive step:

$$f\left(\sum_{k=1}^n \alpha_k x_k\right) = f\left(\alpha_n x_n + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right)$$

Algebra.

PROOF (CONTINUED)

Inductive step:

$$\begin{aligned} f\left(\sum_{k=1}^n \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \end{aligned}$$

Convexity.

PROOF (CONTINUED)

Inductive step:

$$\begin{aligned} f\left(\sum_{k=1}^n \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} f(x_k) \end{aligned}$$

Induction.

PROOF (CONTINUED)

Inductive step:

$$\begin{aligned} f\left(\sum_{k=1}^n \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} f(x_k) \\ &= \sum_{k=1}^n \alpha_k f(x_k). \quad \square \qquad \text{Algebra.} \end{aligned}$$

JENSEN'S INEQUALITY

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \leq E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\} \right)$$

Definition of expectation.

JENSEN'S INEQUALITY

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \leq E[f(X)]$.

Proof.

$$\begin{aligned} f(E[X]) &= f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right) \\ &\leq \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\} \end{aligned}$$

Convexity lemma (generalized).

JENSEN'S INEQUALITY

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \leq E[f(X)]$.

Proof.

$$\begin{aligned} f(E[X]) &= f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right) \\ &\leq \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\} \\ &= E[f(X)]. \quad \blacksquare \end{aligned}$$

Tricky step, but true—think about it.

ANALYSIS OF BST HEIGHT

Let X_n be the random variable denoting the height of a randomly built binary search tree on n nodes, and let $Y_n = 2^{X_n}$ be its exponential height.

If the root of the tree has rank k , then

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\} ,$$

since each of the left and right subtrees of the root are randomly built. Hence, we have

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\} .$$

ANALYSIS (CONTINUED)

Define the indicator random variable Z_{nk} as

$$Z_{nk} = \begin{cases} 1 & \text{if the root has rank } k, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $\Pr\{Z_{nk} = 1\} = E[Z_{nk}] = 1/n$, and

$$Y_n = \sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\}) .$$

EXPONENTIAL HEIGHT RECURRENCE

$$E[Y_n] = E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\}) \right]$$

Take expectation of both sides.

EXPONENTIAL HEIGHT RECURRENCE

$$\begin{aligned} E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\ &= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})] \end{aligned}$$

Linearity of expectation.

EXPONENTIAL HEIGHT RECURRENCE

$$\begin{aligned} E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\ &= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})] \\ &= 2 \sum_{k=1}^n E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}] \end{aligned}$$

Independence of the rank of the root
from the ranks of subtree roots.

EXPONENTIAL HEIGHT RECURRENCE

$$\begin{aligned} E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\ &= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})] \\ &= 2 \sum_{k=1}^n E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}] \\ &\leq \frac{2}{n} \sum_{k=1}^n E[Y_{k-1} + Y_{n-k}] \end{aligned}$$

The max of two nonnegative numbers is at most their sum, and $E[Z_{nk}] = 1/n$.

EXPONENTIAL HEIGHT RECURRENCE

$$\begin{aligned} E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\ &= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})] \\ &= 2 \sum_{k=1}^n E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}] \\ &\leq \frac{2}{n} \sum_{k=1}^n E[Y_{k-1} + Y_{n-k}] \\ &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \end{aligned}$$

Each term appears twice, and reindex.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c , which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k]$$

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned} E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\ &\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \end{aligned}$$

Substitution.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned} E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\ &\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \\ &\leq \frac{4c}{n} \int_0^n x^3 dx \end{aligned}$$

Integral method.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned} E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\ &\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \\ &\leq \frac{4c}{n} \int_0^n x^3 dx \\ &= \frac{4c}{n} \left(\frac{n^4}{4} \right) \end{aligned}$$

Solve the integral.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned} E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\ &\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \\ &\leq \frac{4c}{n} \int_0^n x^3 dx \\ &= \frac{4c}{n} \left(\frac{n^4}{4} \right) \\ &= cn^3. \end{aligned}$$

Algebra.

THE GRAND FINALE

Putting it all together, we have

$$2^{E[X_n]} \leq E[2^{X_n}]$$

Jensen's inequality, since
 $f(x) = 2^x$ is convex.

THE GRAND FINALE

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \end{aligned}$$

Definition.

THE GRAND FINALE

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \\ &\leq cn^3. \end{aligned}$$

What we just showed.

THE GRAND FINALE

Putting it all together, we have

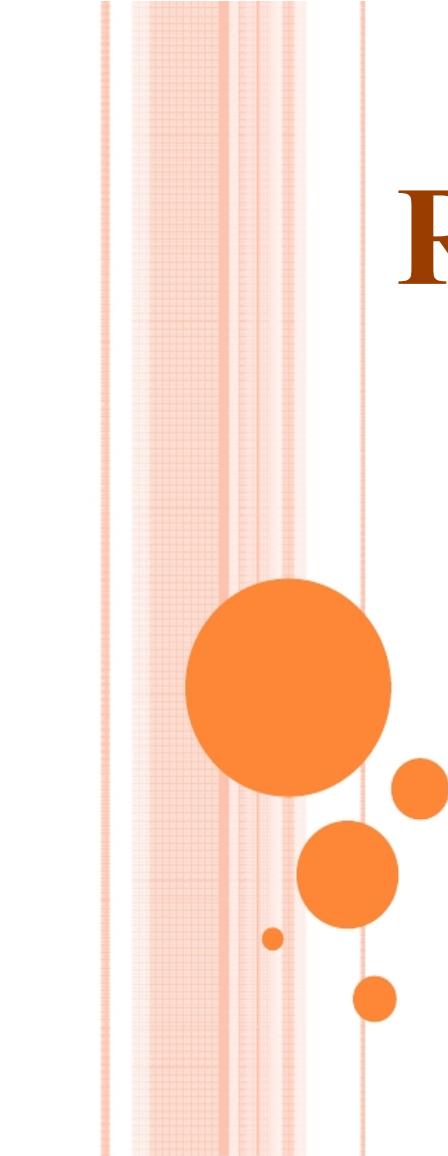
$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \\ &\leq cn^3. \end{aligned}$$

Taking the \lg of both sides yields

$$E[X_n] \leq 3 \lg n + O(1).$$

不会考太多 BST

Red Black Trees



Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

RED-BLACK TREES

- *Red-black trees*:

Binary search trees augmented with node color

Operations designed to guarantee that the height
 $h = O(\lg n)$

- First: describe the properties of red-black trees
- Then: prove that these guarantee $h = O(\lg n)$
- Finally: describe operations on red-black trees

记忆

RED-BLACK PROPERTIES

- The *red-black properties*:

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
 - Note: this means every “real” node has 2 children
3. If a node is red, both children are black
 - Note: can’t have 2 consecutive reds on a path
4. Every path from node to descendent leaf contains the same number of black nodes (黑高)
5. The root is always black

RED-BLACK TREES

- Put example on board and verify properties:
 1. Every node is either red or black
 2. Every leaf (NULL pointer) is black
 3. If a node is red, both children are black
 4. Every path from node to descendent leaf contains the same number of black nodes
 5. The root is always black
- *black-height*: # black nodes on path to leaf
 - Label example with h and bh values

HEIGHT OF RED-BLACK TREES

- *What is the minimum black-height of a node with height h ?*
- A: a height- h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$
- *How do you suppose we'll prove this?*

RB TREES: PROVING HEIGHT BOUND

- o Prove: n -node RB tree has height $h \leq 2 \lg(n+1)$
- o Claim: A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes

Proof by induction on height h

Base step: x has height 0 (i.e., NULL leaf node)

- o *What is $bh(x)$?*

RB TREES: PROVING HEIGHT BOUND

- Prove: n -node RB tree has height $h \leq 2 \lg(n+1)$
- Claim: A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes

Proof by induction on height h

Base step: x has height 0 (i.e., NULL leaf node)

- *What is $bh(x)$?*

- A: 0

- So...subtree contains $2^{bh(x)} - 1$
 $= 2^0 - 1$
 $= 0$ internal nodes (TRUE)

RB TREES: PROVING HEIGHT BOUND

- Inductive proof that subtree at node x contains at least $2^{\text{bh}(x)} - 1$ internal nodes

Inductive step: x has positive height and 2 children

- Each child has black-height of $\text{bh}(x)$ or $\text{bh}(x)-1$ (*Why?*)
- The height of a child = (height of x) - 1
- So the subtrees rooted at each child contain at least $2^{\text{bh}(x) - 1} - 1$ internal nodes
- Thus subtree at x contains
$$(2^{\text{bh}(x) - 1} - 1) + (2^{\text{bh}(x) - 1} - 1) + 1 \\ = 2 \cdot 2^{\text{bh}(x)-1} - 1 = 2^{\text{bh}(x)} - 1$$
 nodes

RB TREES: PROVING HEIGHT BOUND

- Thus at the root of the red-black tree:

$$n \geq 2^{\text{bh}(\text{root})} - 1 \quad (\text{Why?})$$

$$n \geq 2^{h/2} - 1 \quad (\text{Why?})$$

$$\lg(n+1) \geq h/2 \quad (\text{Why?})$$

$$h \leq 2 \lg(n + 1) \quad (\text{Why?})$$

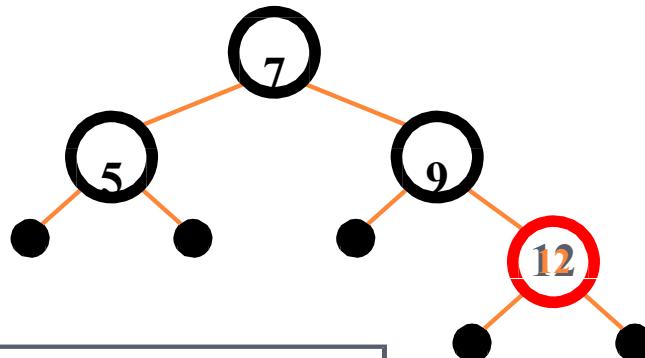
Thus $h = O(\lg n)$

RB TREES: WORST-CASE TIME

- So we've proved that a red-black tree has $O(\lg n)$ height
- Corollary: These operations take $O(\lg n)$ time:
 - Minimum(), Maximum()
 - Successor(), Predecessor()
 - Search()
- Insert() and Delete():
 - Will also take $O(\lg n)$ time
 - But will need special care since they modify tree

RED-BLACK TREES: AN EXAMPLE

○ *Color this tree:*



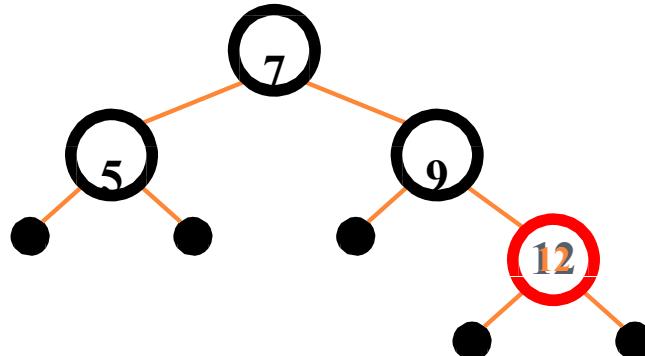
Red-black properties:

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

- *Insert 8:*

- *Where does it go?*



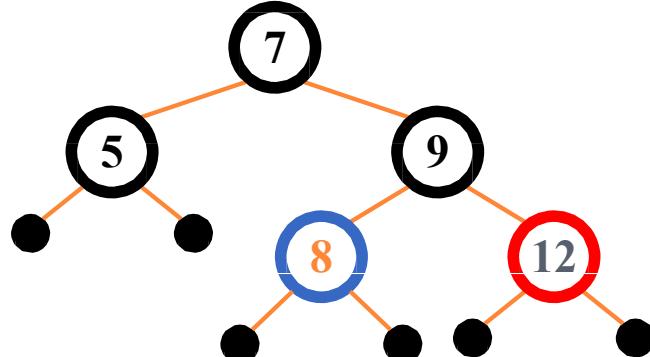
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

- Insert 8

Where does it go?

*What color
should it be?*



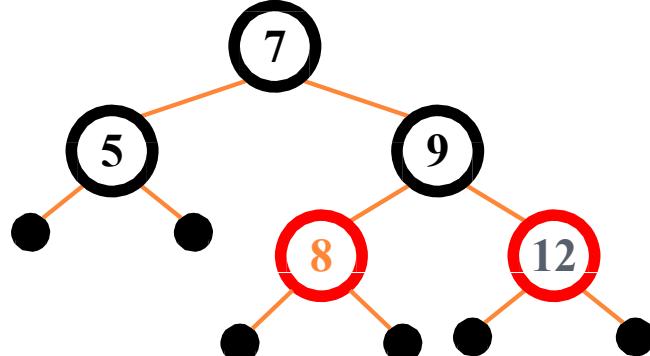
- Every node is either red or black
- Every leaf (NULL pointer) is black
- If a node is red, both children are black
- Every path from node to descendent leaf contains the same number of black nodes
- The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

- Insert 8

Where does it go?

*What color
should it be?*

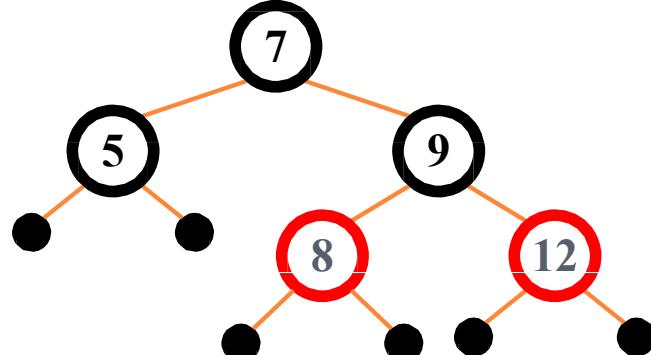


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

- Insert 11

Where does it go?



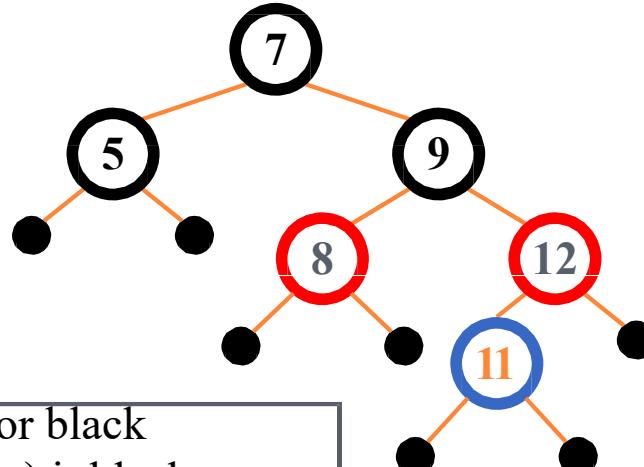
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

- Insert 11

Where does it go?

What color?



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

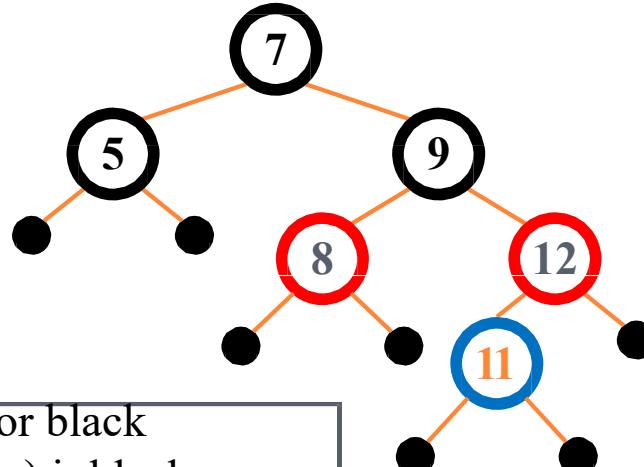
RED-BLACK TREES: THE PROBLEM WITH INSERTION

- Insert 11

Where does it go?

What color?

- Can't be red! (#3)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendant leaf contains the same number of black nodes
5. The root is always black

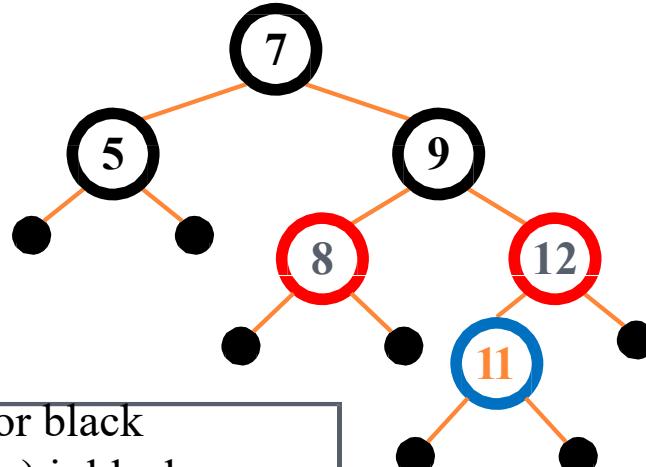
RED-BLACK TREES: THE PROBLEM WITH INSERTION

○ Insert 11

Where does it go?

What color?

- Can't be red! (#3)
- Can't be black! (#4)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendant leaf contains the same number of black nodes
5. The root is always black

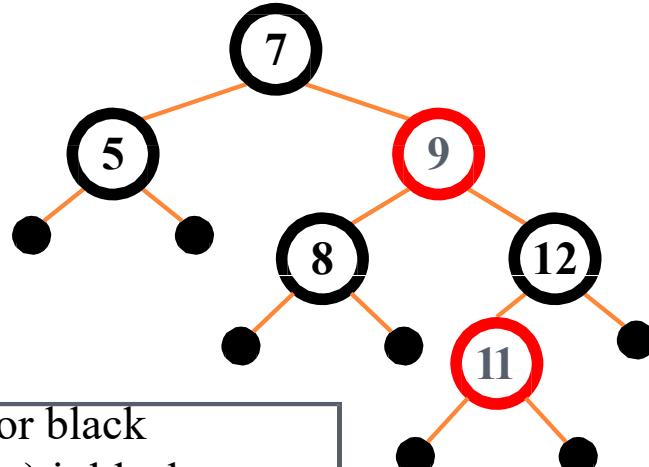
RED-BLACK TREES: THE PROBLEM WITH INSERTION

Insert 11

Where does it go?

What color?

- Solution:
recolor the tree

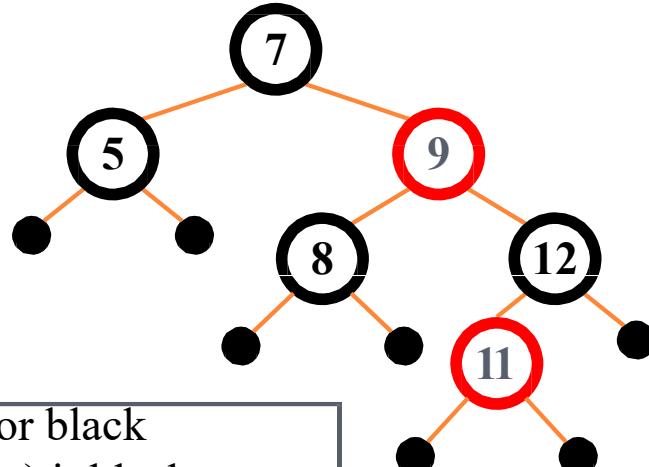


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendant leaf contains the same number of black nodes
5. The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

- Insert 10

Where does it go?



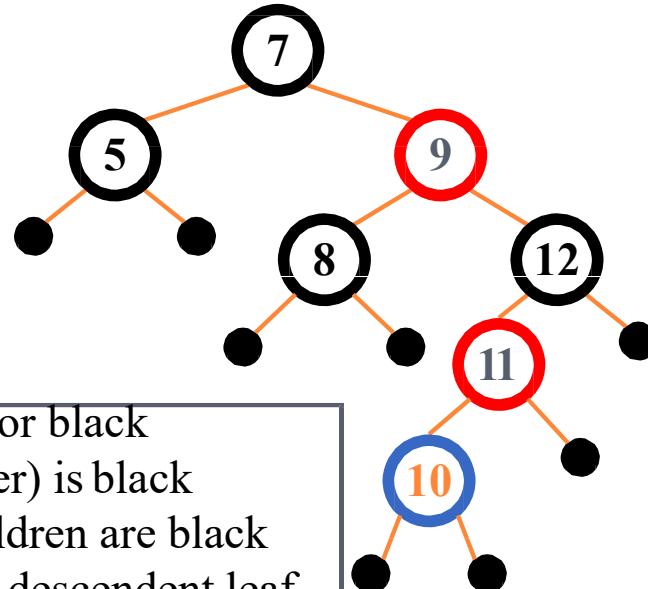
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

- Insert 10

Where does it go?

What color?



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendant leaf contains the same number of black nodes
5. The root is always black

RED-BLACK TREES: THE PROBLEM WITH INSERTION

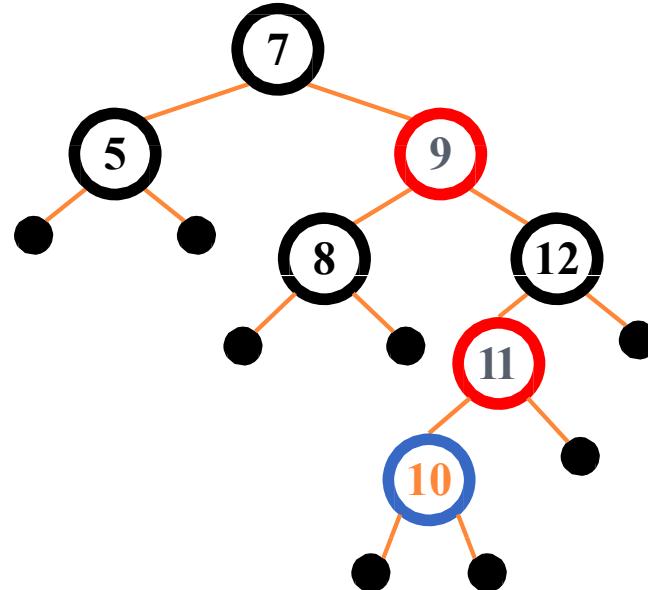
- Insert 10

Where does it go?

What color?

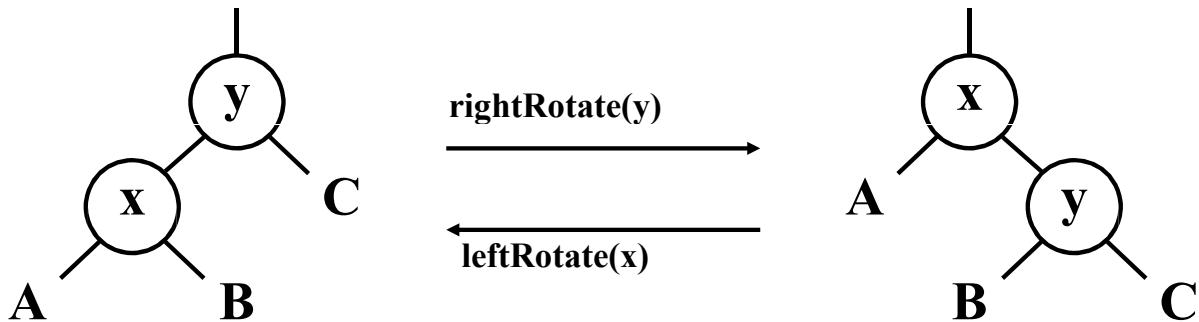
- A: no color! Tree is too imbalanced
- Must change tree structure to allow recoloring

Goal: restructure tree in $O(\lg n)$ time



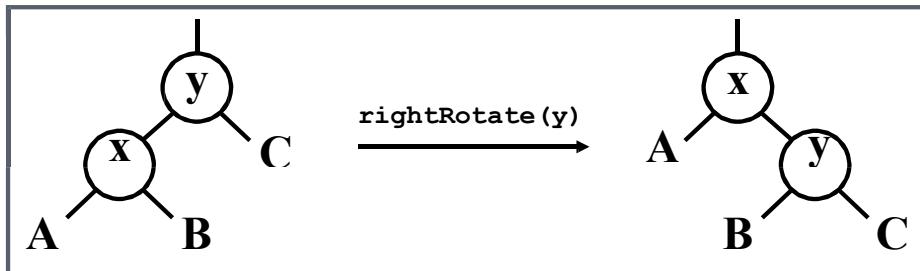
RB TREES: ROTATION

- Our basic operation for changing tree structure is called *rotation*:



- Does rotation preserve inorder key ordering?
- What would the code for `rightRotate()` actually do?

RB TREES: ROTATION



- Answer: A lot of pointer manipulation

- x keeps its left child

- y keeps its right child

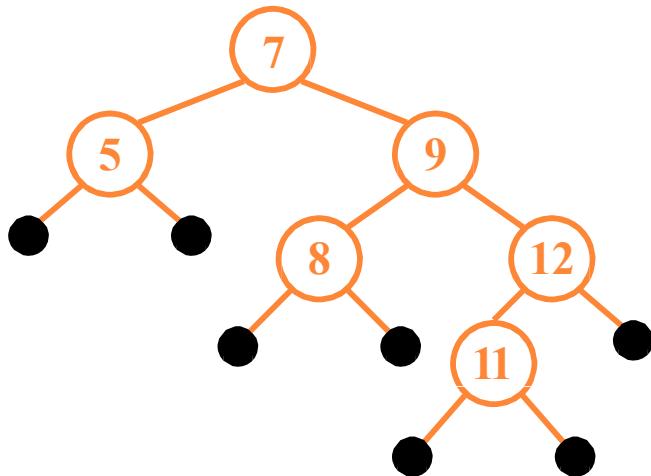
- x 's right child becomes y 's left child

- x 's and y 's parents change

- *What is the running time?*

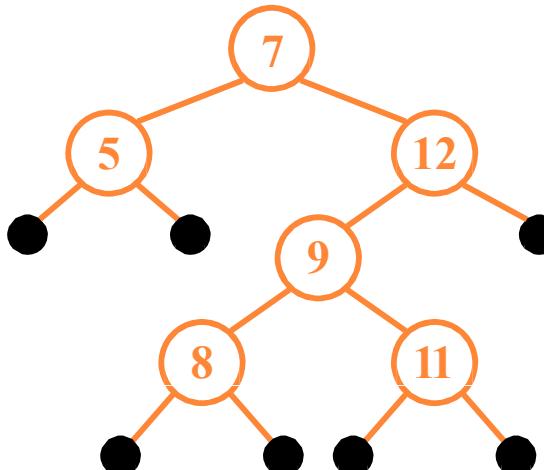
ROTATION EXAMPLE

- Rotate left about 9:



ROTATION EXAMPLE

- Rotate left about 9:



26

RED-BLACK TREES: INSERTION

- Insertion: the basic idea

- Insert x into tree, color x red

- Only r-b property 3 might be violated (if $p[x]$ red)

- If so, move violation up tree until a place is found where it can be fixed

- Total time will be $O(\lg n)$

```

rbInsert(x)
    treeInsert(x);
    x->color = RED;
    // Move violation of #3 up tree, maintaining #4 as invariant:
    while (x!=root && x->p->color == RED)
        if (x->p == x->p->p->left)
            y = x->p->p->right;
            if (y->color == RED)
                x->p->color = BLACK;
                y->color = BLACK;
                x->p->p->color = RED;
                x = x->p->p;
            else // y->color == BLACK
                if (x == x->p->right)
                    x = x->p;
                    leftRotate(x);
                    x->p->color = BLACK;
                    x->p->p->color = RED;
                    rightRotate(x->p->p);
        else // x->p == x->p->p->right
            (same as above, but with
             "right" & "left" exchanged)

```



```

rbInsert(x)

    treeInsert(x);
    x->color = RED;

    // Move violation of #3 up tree, maintaining #4 as invariant:
    while (x!=root && x->p->color == RED)
        if (x->p == x->p->p->left)
            y = x->p->p->right;
            if (y->color == RED)
                x->p->color = BLACK;
                y->color = BLACK;
                x->p->p->color = RED;
                x = x->p->p;
            else // y->color == BLACK
                if (x == x->p->right)
                    x = x->p;
                    leftRotate(x);
                    x->p->color = BLACK;
                    x->p->p->color = RED;
                    rightRotate(x->p->p);
                else // x->p == x->p->p->right
                    (same as above, but with
                     "right" & "left" exchanged)

```

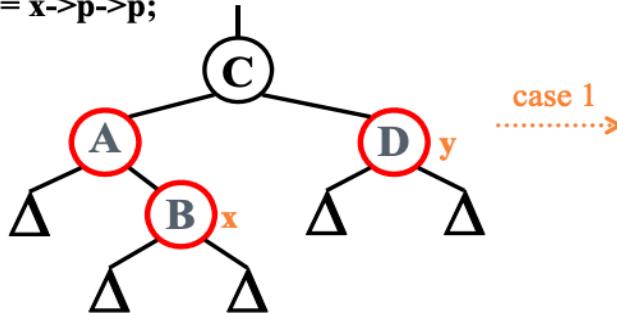
} Case 1: uncle is RED

} Case 2

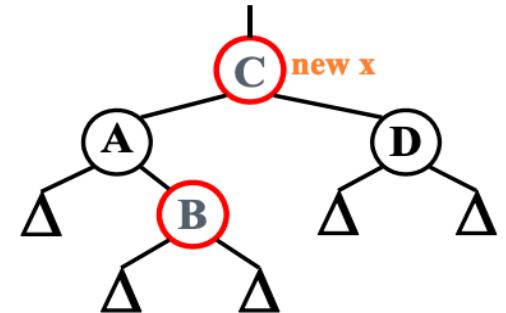
} Case 3

RB INSERT: CASE 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```



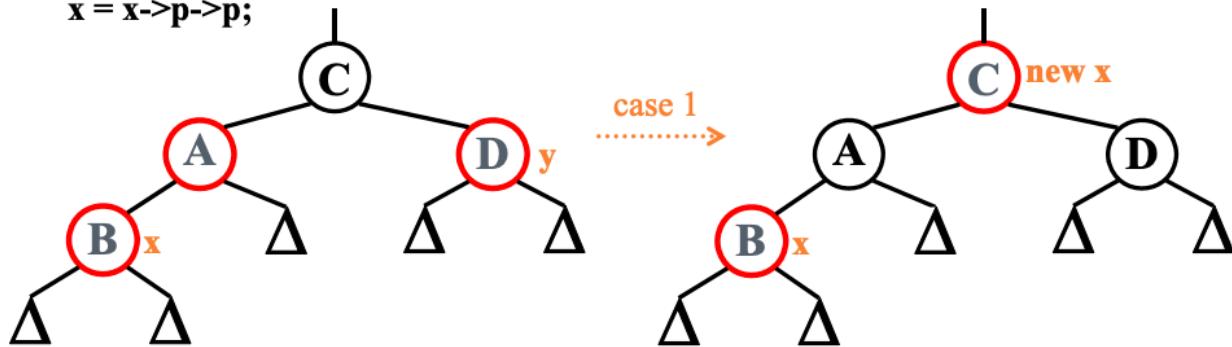
- Case 1: “uncle” is red
- In figures below, all Δ ’s are equal-black-height subtrees



Change colors of some nodes, preserving #4: all downward paths have equal b.h.
The while loop now continues with x’s grandparent as the new x

RB INSERT: CASE 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

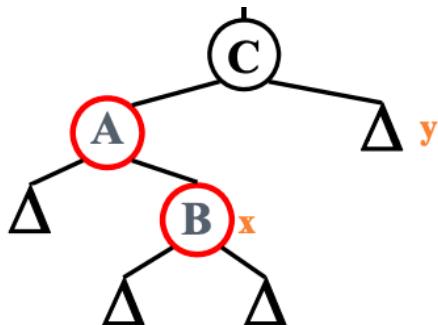


- Case 1: “uncle” is red
- In figures below, all Δ ’s are equal-black-height subtrees

Same action whether x is a left or a right child

RB INSERT: CASE 2

```
if (x == x->p->right)
    x = x->p;
    leftRotate(x);
// continue with case 3 code
```

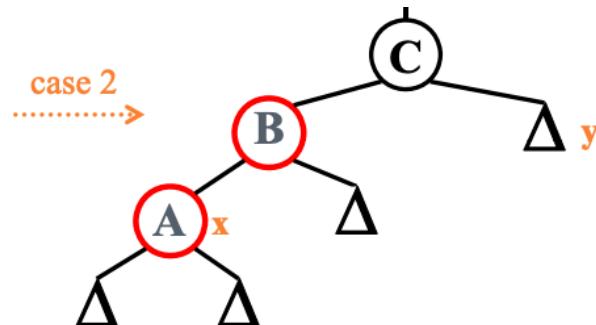


- Case 2:

“Uncle” is black

Node x is a right child

- Transform to case 3 via a left-rotation

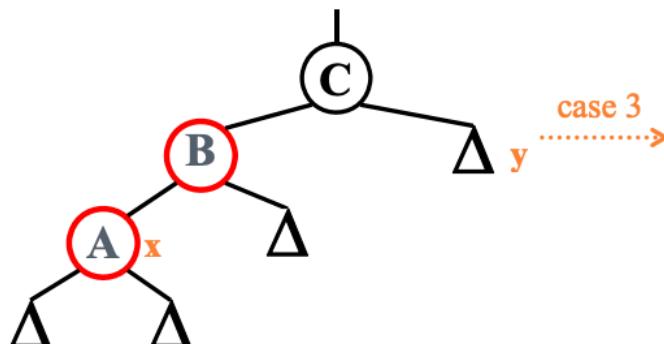


case 2
.....>

Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

RB INSERT: CASE 3

```
x->p->color = BLACK;  
x->p->p->color = RED;  
rightRotate(x->p->p);
```

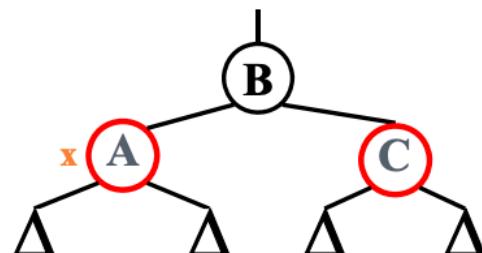


- Case 3:

- “Uncle” is black

- Node x is a left child

- Change colors; rotate right



Perform some color changes and do a right rotation

Again, preserves property 4: all downward paths contain same number of black nodes

Argumenting Data Structures

Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

INTRODUCTION

In most cases a standard data structure is sufficient
(possibly provided by a software library)

But sometimes one needs additional operations that aren't supported by any standard data structure

→ need to design new data structure?

Not always: often **augmenting** an existing structure is sufficient

“One good thief is worth ten good scholars”



DYNAMIC ORDER STATISTICS

We've seen algorithms for finding the i th element of an unordered set in $O(n)$ time

OS-Tree(order statistic tree) T: a structure to support finding the i th element of a dynamic set in $O(\lg n)$ time

Support standard dynamic set operations (Insert(), Delete(), Min(), Max(), Succ(), Pred())

Also support these order statistic operations:

```
void OS-Select(root, i);  
int OS-Rank(x);
```

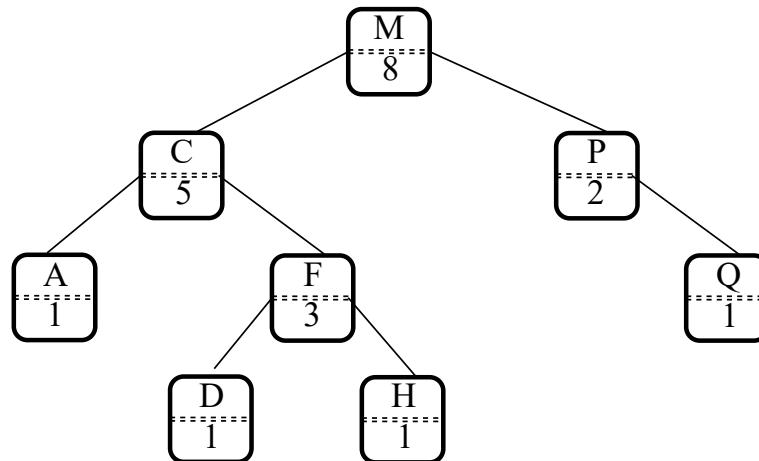
REVIEW: ORDER STATIC TREES

OS-Trees augment red-black trees:

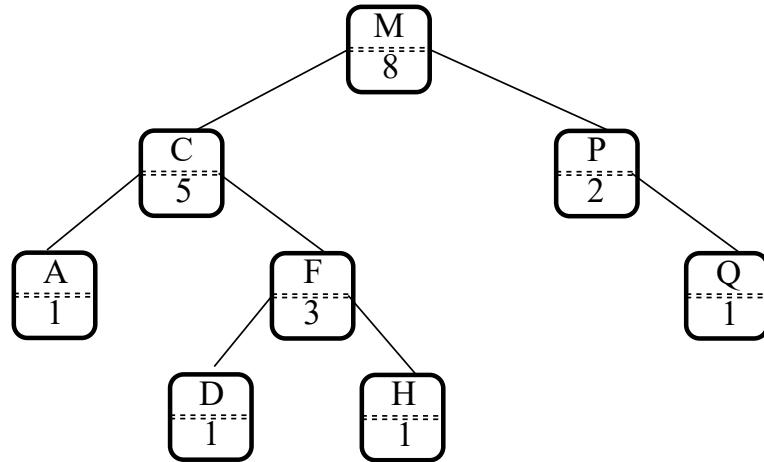
Associate a size field with each node in the tree

$x \rightarrow \text{size}$ records the size of subtree rooted at x ,
including x itself:

$$\text{size}[\text{nil}[T]] = 0$$



SELECTION ON OS-T REES



$$\text{size}[x] = \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1$$

How can we use this property
to select the *i*th element of the set?



OS-SELECT

```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```

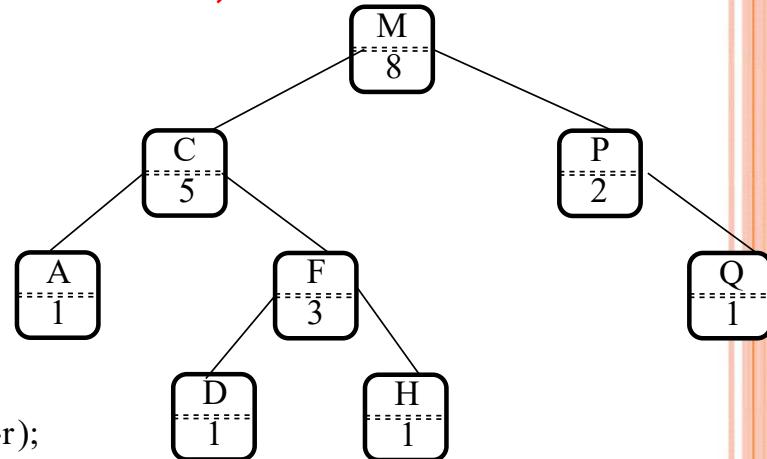


OS-SELECT EXAMPLE

Example: show OS-Select(root, 5):

```
OS-Select(x, i)
```

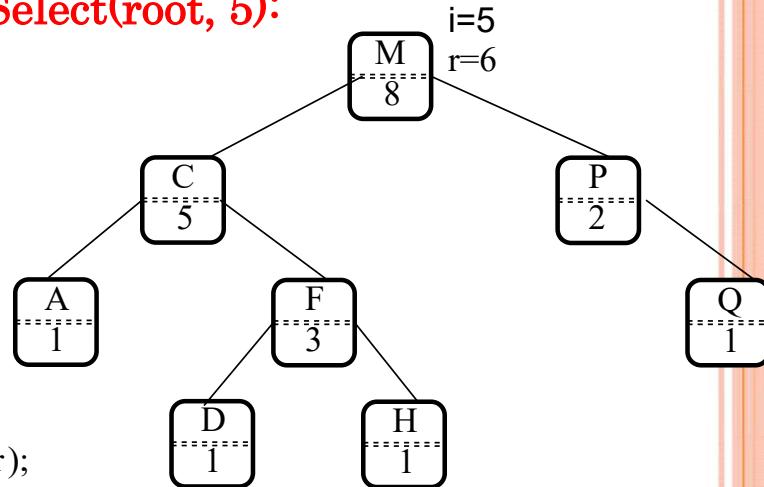
```
{  
    r = x->left->size + 1;  
    if (i == r)  
        return x;  
    else if (i < r)  
        return OS-Select(x->left, i);  
    else  
        return OS-Select(x->right, i-r);  
}
```



OS-SELECT EXAMPLE

Example: show OS-Select(root, 5):

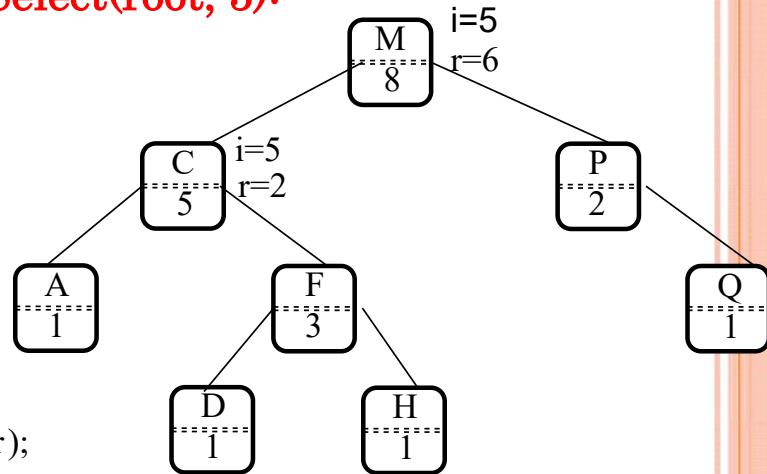
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```



OS-SELECT EXAMPLE

Example: show OS-Select(root, 5):

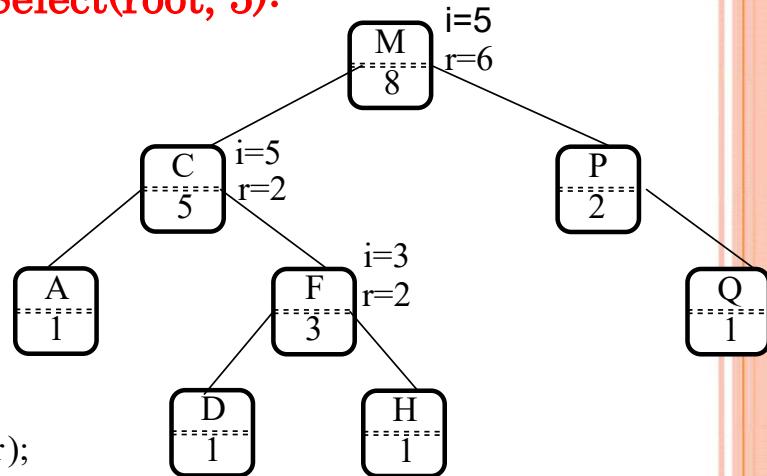
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```



OS-SELECT EXAMPLE

Example: show OS-Select(root, 5):

```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```

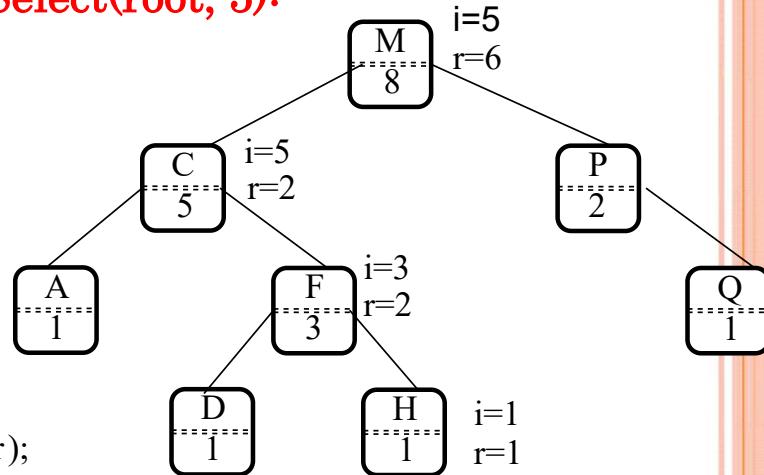


OS-SELECT EXAMPLE

Example: show OS-Select(root, 5):

```
OS-Select(x, i)
```

```
{  
    r = x->left->size + 1;  
    if (i == r)  
        return x;  
    else if (i < r)  
        return OS-Select(x->left, i);  
    else  
        return OS-Select(x->right, i-r);  
}
```



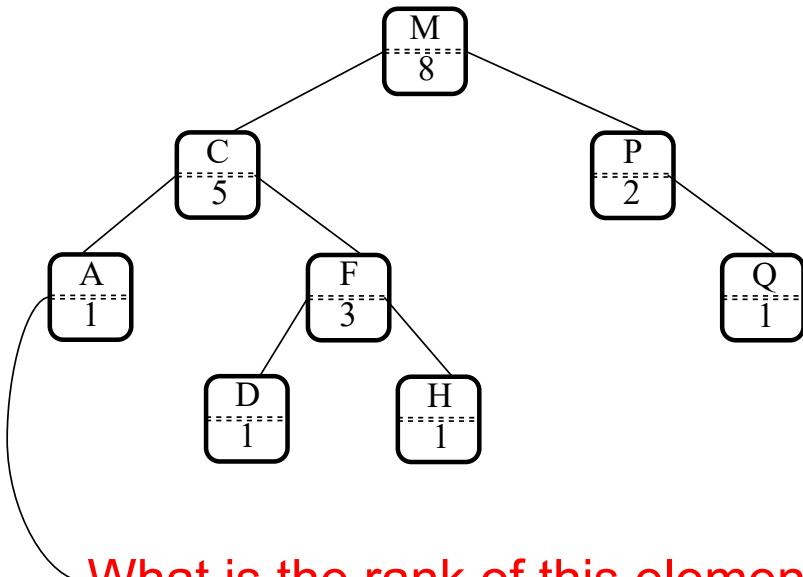
OS-SELECT: A SUBTLETY

```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```

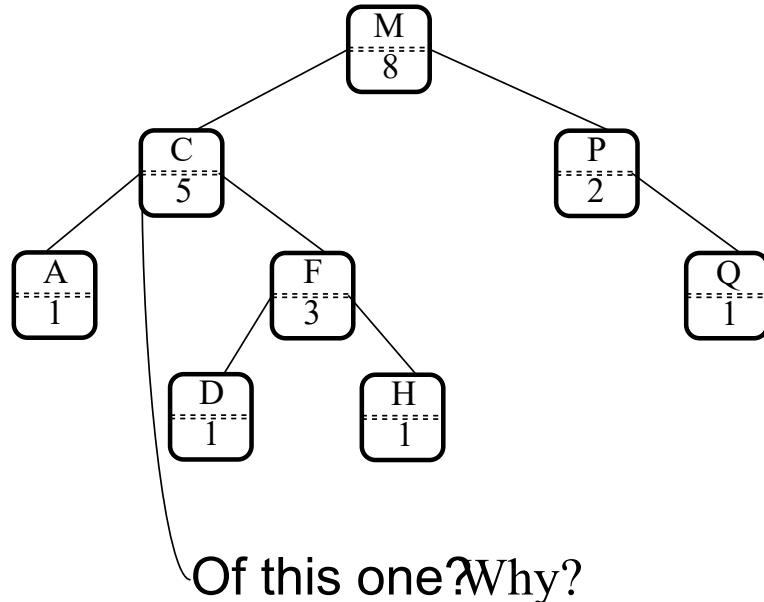
- z What happens at the leaves?
- z How can we deal elegantly with this?
- z What will be the running time?



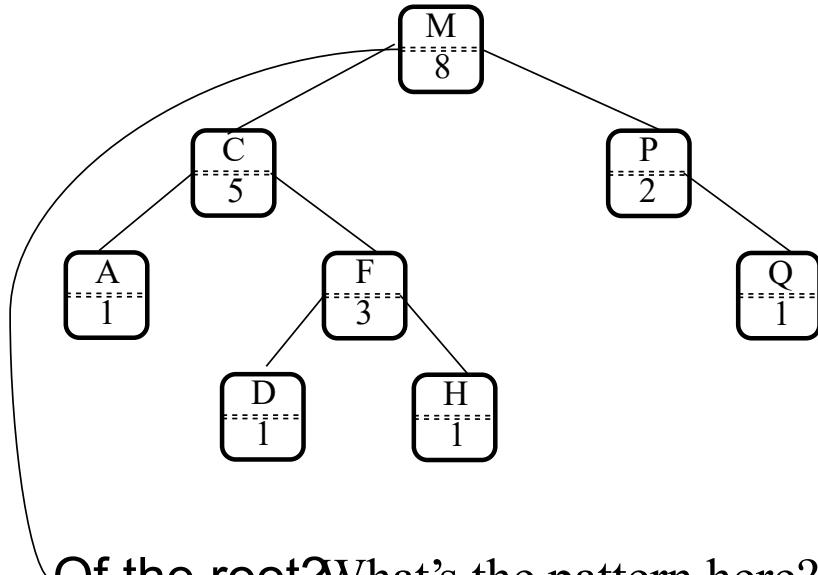
DETERMINING THE RANK OF AN ELEMENT



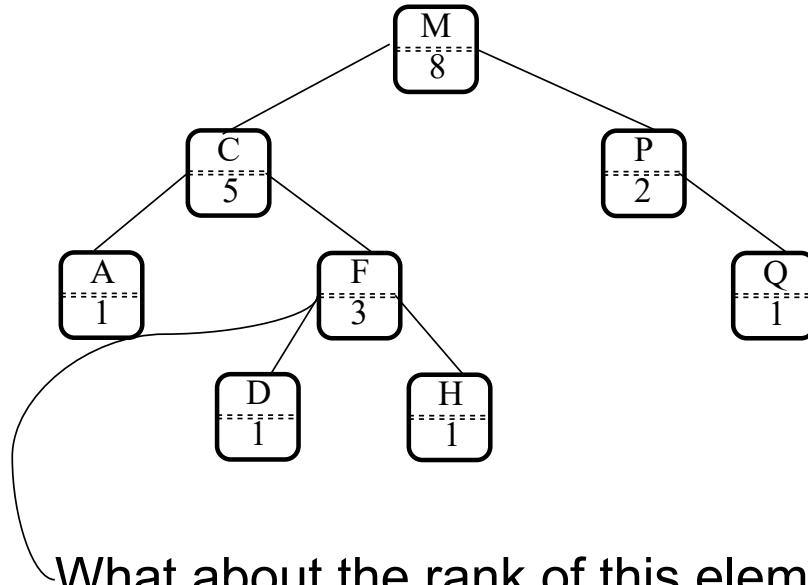
DETERMINING THE RANK OF AN ELEMENT



DETERMINING THE RANK OF AN ELEMENT

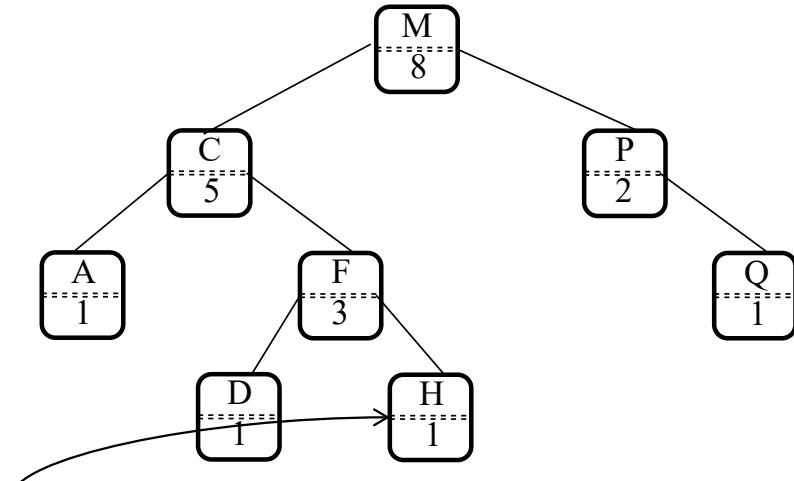


DETERMINING THE RANK OF AN ELEMENT



What about the rank of this element?

DETERMINING THE RANK OF AN ELEMENT



This one? What's the pattern here?



OS-RANK

```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size +
    1;
    y = y->p;
    return r;
}
```

z What will be the running time?

OS-TREES : MAINTAINING SIZES

So we've shown that with subtree sizes, order statistic operations can be done in $O(\lg n)$ time

Next step: maintain sizes during `Insert()` and `Delete()` operations

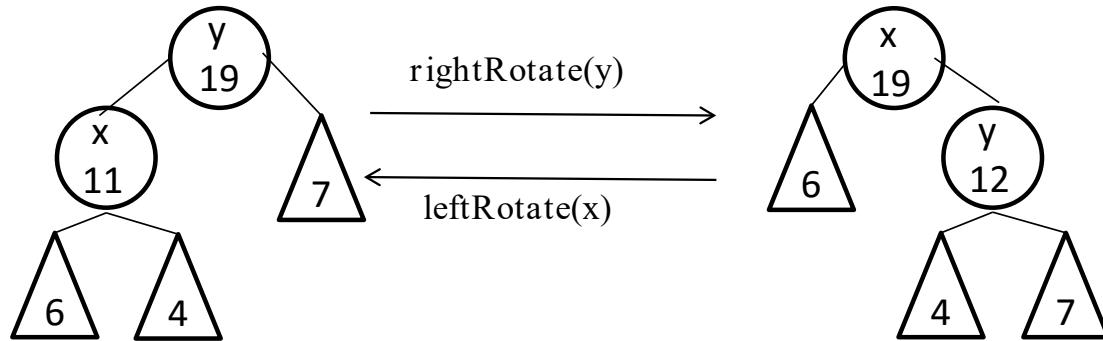
How would we adjust the size fields during insertion on a plain binary search tree?

A: increment sizes of nodes traversed during search

Why won't this work on red-black trees?



MAINTAINING SIZE THROUGH ROTATION



- z Salient point: rotation invalidates only x and y
- z Can recalculate their sizes in constant time

z Why?

- z $12 \text{ size}[y] \leftarrow \text{size}[x]$
- z $13 \text{ size}[x] \leftarrow \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1$

AUGMENTING DATA STRUCTURES: METHODOLOGY

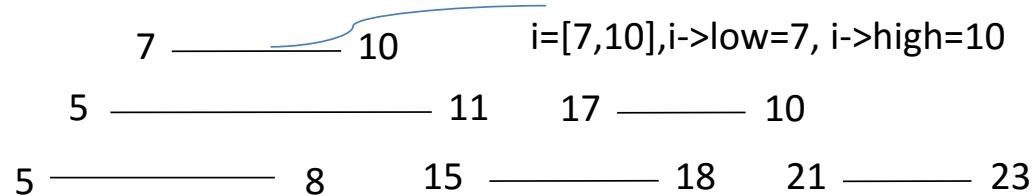
- z Choose underlying data structure
 - E.g., red-black trees
- z Determine additional information to maintain
 - E.g., subtree sizes
- z Verify that information can be maintained for operations that modify the structure
 - E.g., Insert(), Delete() (don't forget rotations!)
- z Develop new operations
 - E.g., OS-Rank(), OS-Select()



INTERVAL TREES

- The problem : maintain a set of intervals

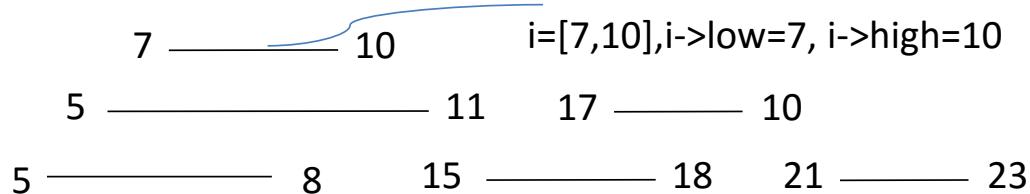
- E.g. , time intervals for a scheduling program :



INTERVAL TREES

- The problem : maintain a set of intervals

- E.g. , time intervals for a scheduling program :



- We can represent an interval $[t1,t2]$ as an object i , with field $\text{slow}[i]=t1$ (**the low endpoint**) and $\text{high}[i]=t2$ (**the high endpoint**)

We say that intervals i and i' overlap if $i \cap i' \neq \emptyset$, that is, if $\text{low}[i] \leq \text{high}[i']$ and $\text{low}[i'] \leq \text{high}[i]$. Any two intervals i and i' satisfy the **interval trichotomy**; that exactly one of the following three properties holds:

- a. i and i' overlap.,
- b. i is to the left of i' (i.e., $\text{high}[i] < \text{low}[i']$),.
- c. i is to the right of i' (i.e., $\text{high}[i'] < \text{low}[i]$),.

INTERVAL TREES

- z Interval trees support the following operations.
 - INTERVAL-INSERT(T, x)
 - INTERVAL-DELETE(T, x)
 - INTERVAL-SEARCH(T, i)

INTERVAL TREES

- z Following the methodology:
 - Pick underlying data structure
 - Decide what additional information to store
 - Figure out how to maintain the information
 - Develop the desired new operations



INTERVAL TREES

- z Following the methodology:
 - Pick underlying data structure
 - Red-black trees will store intervals, keyed on $i \rightarrow \text{low}$
 - Decide what additional information to store
 - Figure out how to maintain the information
 - Develop the desired new operations

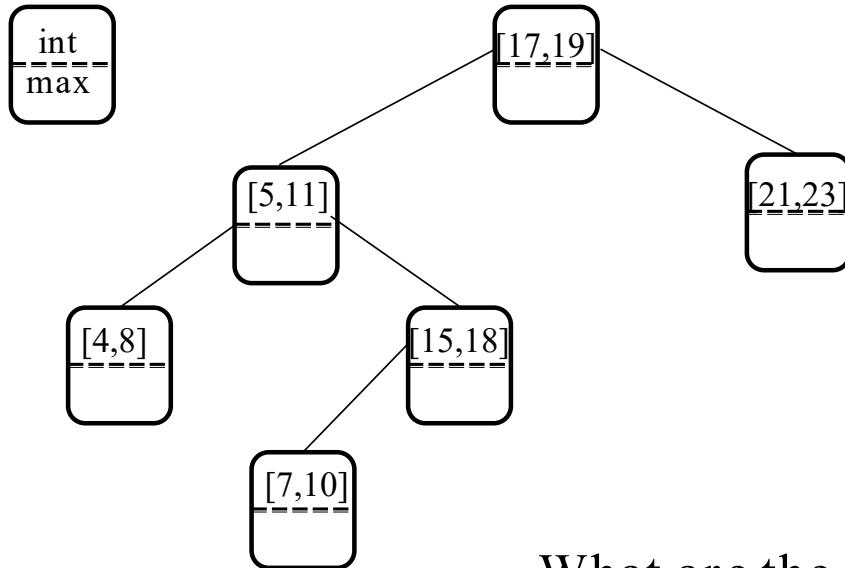


INTERVAL TREES

- z Following the methodology:
 - Pick underlying data structure
 - Red-black trees will store intervals, keyed on $i \rightarrow \text{low}$
 - Decide what additional information to store
 - We will store max , the maximum endpoint in the subtree rooted at i .
 - Figure out how to maintain the information
 - Develop the desired new operations



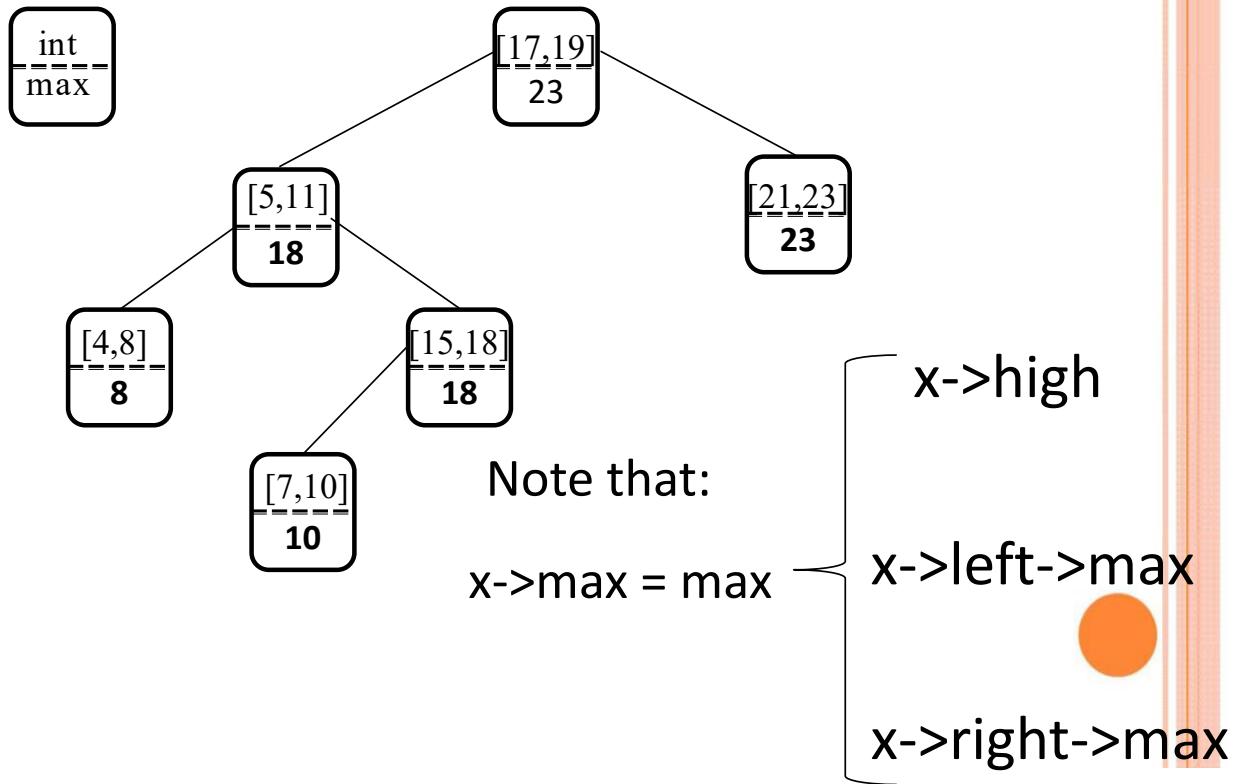
INTERVAL T REES



What are the max fields?



INTERVAL TREES

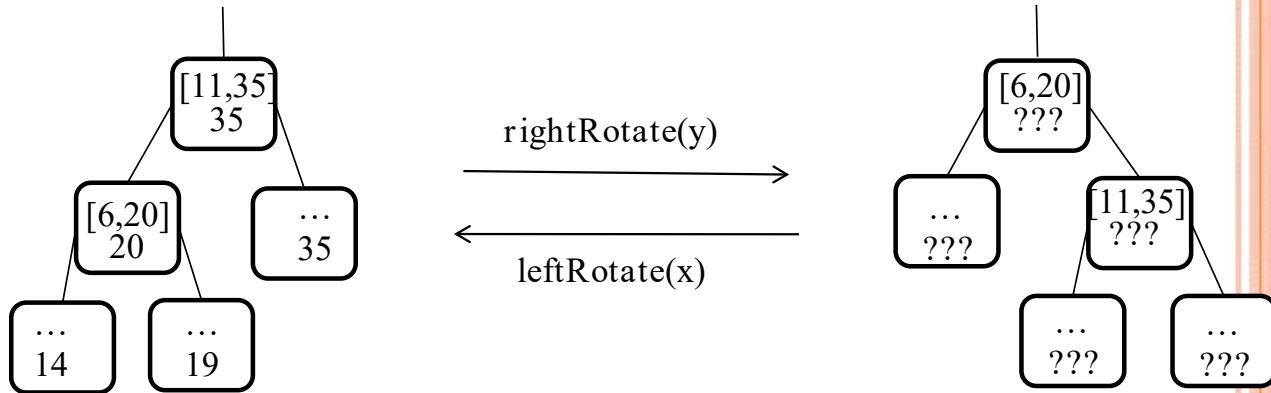


INTERVAL TREES

- z Following the methodology:
 - Pick underlying data structure
 - Red-black trees will store intervals, keyed on $i \rightarrow \text{low}$
 - Decide what additional information to store
 - Store the maximum endpoint in the subtree rooted at i
 - Figure out how to maintain the information
 - How would we maintain max field for a BST?
 - What's different?
 - Develop the desired new operations

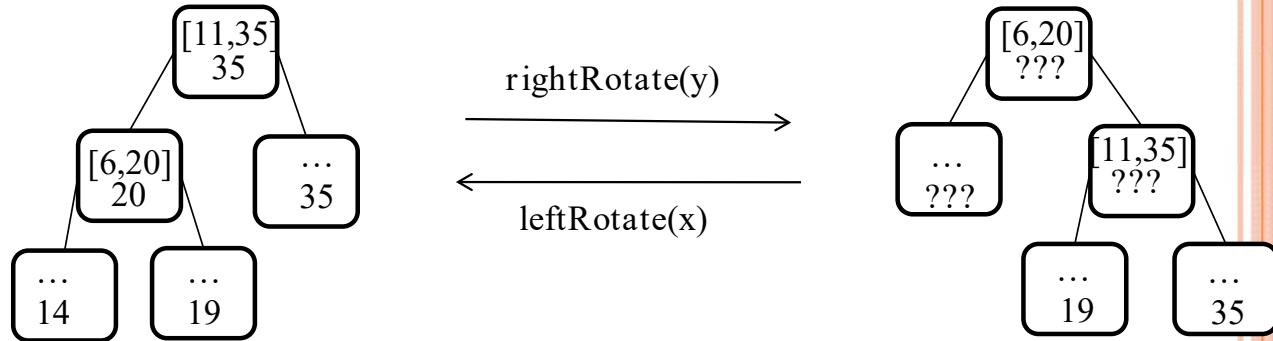


INTERVAL TREES



What are the new max values for the subtrees?

INTERVAL TREES



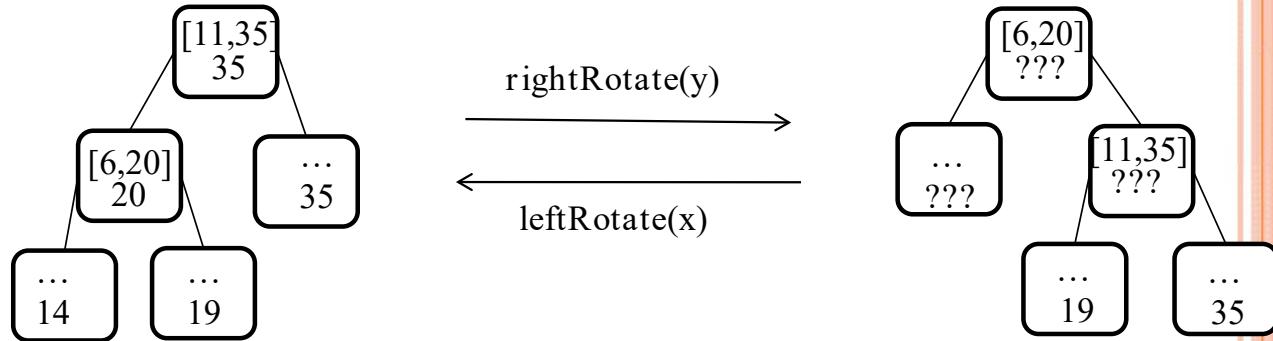
What are the new max values for the subtrees?

A: Unchanged

What are the new max values for x and y?



INTERVAL TREES



What are the new max values for the subtrees?

A: Unchanged

What are the new max values for x and y?

A: root value unchanged, recompute other



INTERVAL TREES

- z Following the methodology:
 - Pick underlying data structure
 - Red-black trees will store intervals, keyed on $i \rightarrow \text{low}$
 - Decide what additional information to store
 - Store the maximum endpoint in the subtree rooted at i
 - Figure out how to maintain the information
 - Insert: update max on way down, during rotations
 - Delete: similar
 - Develop the desired new operations



SEARCHING INTERVAL TREES

```
IntervalSearch(T, i)
{
    x = T->root;
    while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max ⊑ i->low)
            x = x->left;
        else
            x = x->right;
    return x
}
```

z What will be the running time?

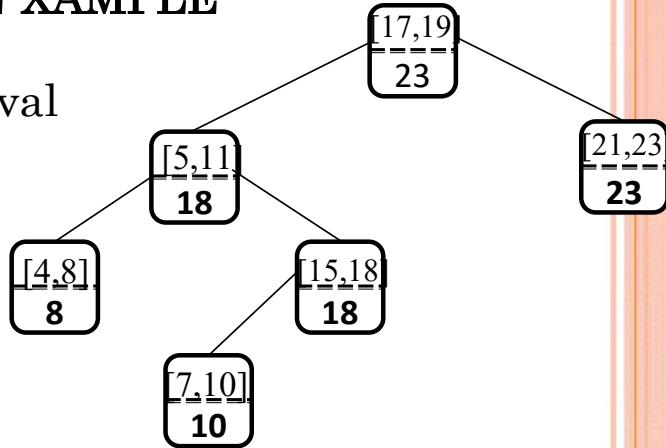


INTERVAL SEARCH () EXAMPLE

Example: search for interval overlapping [14,16]

8
7.10
10

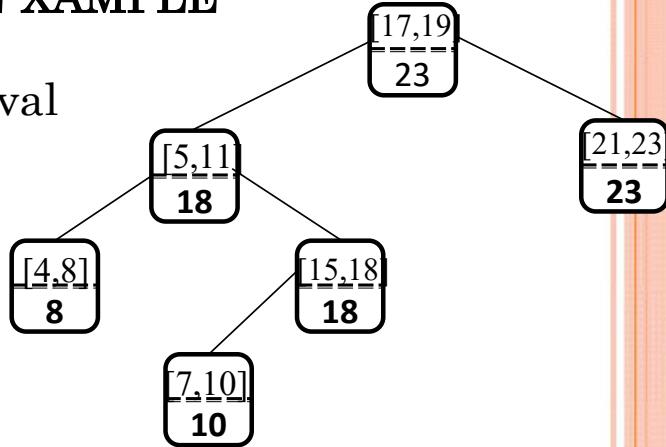
```
IntervalSearch(T, i)
{
    x = T->root;
    while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max < i->low)
            x = x->left;
        else
            x = x->right;
    return x
}
```



INTERVALS SEARCH () EXAMPLE

Example: search for interval overlapping [12,14]

```
IntervalSearch(T, i)
{
    x = T->root;
    while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max < i->low)
            x = x->left;
        else
            x = x->right;
    return x
}
```



C ORRECTNESS OF I NTERVALS EARCH ()

- z Key idea: need to check only 1 of node's 2 children
 - Case 1: search goes right
 - Show that overlap in right subtree, or no overlap at all
 - Case 2: search goes left
 - Show that overlap in left subtree, or no overlap at all

C ORRECTNESS OF I NTERVALS EARCH ()

- z Case 1: if search goes right, overlap in the right subtree or no overlap in either subtree
 - If overlap in right subtree, we're done
 - Otherwise:
 - $x \rightarrow \text{left} = \text{NULL}$, or $x \rightarrow \text{left} \rightarrow \text{max} < x \rightarrow \text{low}$ (Why?)
 - Thus, no overlap in left subtree!

```
while (x != NULL && !overlap(i, x->interval))  
    if (x->left != NULL && x->left->max < i->low)  
        x = x->left;  
    else  
        x = x->right;  
return x;
```



CORRECTNESS OF INTERVALSEARCH()

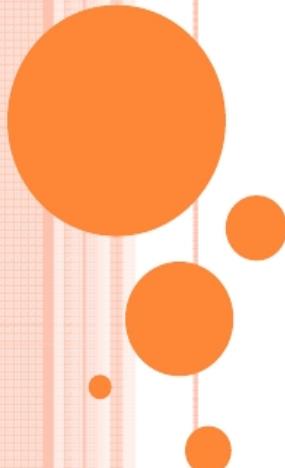
- z Case 2: if search goes left, overlap in the left subtree or no overlap in either subtree
 - If overlap in left subtree, we're done
 - Otherwise:
 - $i \rightarrow low \leq x \rightarrow left \rightarrow max$ by branch condition
 - $x \rightarrow left \rightarrow max = y \rightarrow high$ for some y in left subtree
 - Since i and y don't overlap and $i \rightarrow low \leq y \rightarrow high, i \rightarrow high < y \rightarrow low$
 - Since tree is sorted by low's, $i \rightarrow high <$ any low in right subtree
 - Thus, no overlap in right subtree

```
while (x != NULL && !overlap(i, x->interval))
    if (x->left != NULL && x->left->max < i->low)
        x = x->left;
    else
        x = x->right;
return x;
```

B Trees

Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen



OUTLINE

- *Introduction*
- Definition
- Basic operations
- Applications



INTRODUCTION

- Motivation
- Overview
- Original Publication
- B-tree ...



MOTIVATION

- When data is too large to fit in main memory, then the number of disk accesses becomes important.
- Disk access is unbelievably expensive compared to a typical computer instruction (mechanical limitations).
- One disk access is worth about 200,000 instructions.
- The number of disk accesses will dominate the running time.

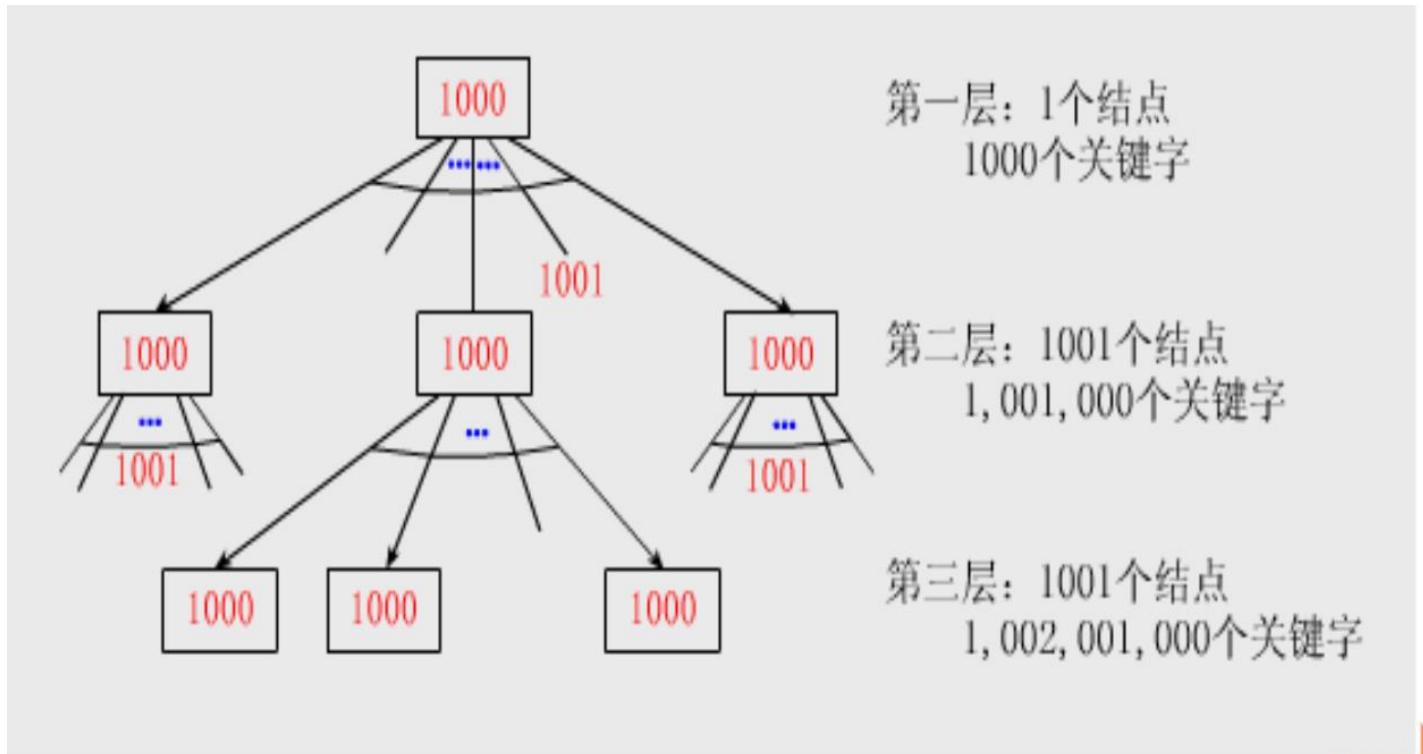
MOTIVATION (CONT.)

- Secondary memory (disk) is divided into equal-sized blocks (typical sizes are 512, 2048, 4096 or 8192 bytes)
- Basic I/O operation transfers the contents of one disk block to/from main memory.
- Goal is to devise a multiway search tree that will *minimize file accesses* (by exploiting disk block read).

OVERVIEW

- Have many children, from a handful to thousands
- The "branching factor" of a B-tree can be quite large, although it is usually determined by characteristics of the disk unit used.
- B-trees are similar to red-black trees in that every n -node B-tree has height $O(\lg n)$, although the height of a B-tree can be considerably less than that of a red-black tree because its branching factor can be much larger.

EXAMPLE



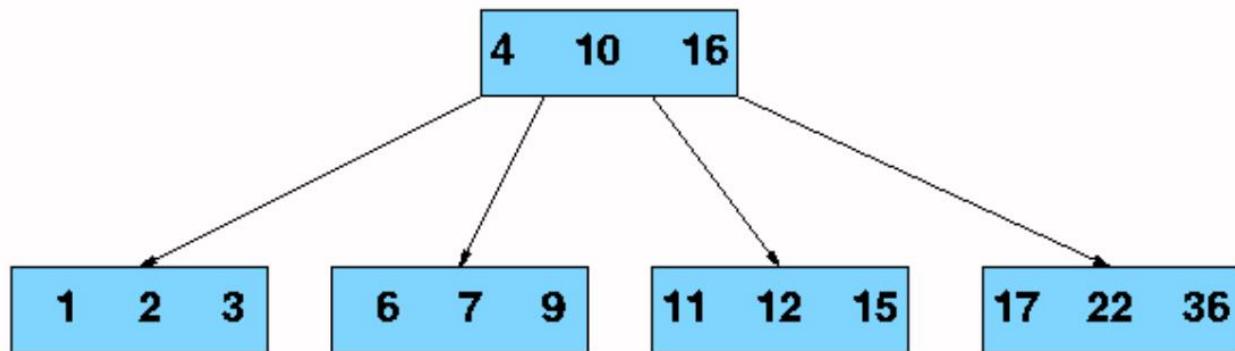
EXAMPLE (CONT.)

- For a large B-tree stored on a disk, branching factors between 50 and 2000 are often used, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key.
- This B-tree with a *branching factor of 1001* and height 2 that can store over one billion keys; nevertheless, since the root node can be kept permanently in main memory, *only two disk accesses* at most are required to find any key in this tree!

OVERVIEW-STRUCTURES

- If an internal B-tree node x contains $n[x]$ keys, then x has $n[x] + 1$ children.
- The keys in node x are used as dividing points separating the range of keys handled by x into $n[x] + 1$ subranges, each handled by one child of x . When searching for a key in a B-tree, we make an $(n[x] + 1)$ -way decision based on comparisons with the $n[x]$ keys stored at node x .
- The structure of leaf nodes differs from that of internal nodes; we will examine these differences later

B-TREE EXAMPLE



HOW TO WORK

- In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once.
- The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed.
- B-tree algorithms are designed so that only a constant number of pages are in main memory at any time; thus, the size of main memory does not limit the size of B-trees that can be handled.

THE ORIGINAL PUBLICATION

- Rudolf Bayer, Edward M. McCreight
- Organization and Maintenance of Large Ordered Indices
- 1972

B-TREE...

- **Also known as** balanced multiway tree
- **Generalization** (I am a kind of ...)
balanced tree, search tree.
- **Specialization** (... is a kind of me.)
2-3-4 tree, B-tree, 2-3 tree..*

B-TREE (CONT.)

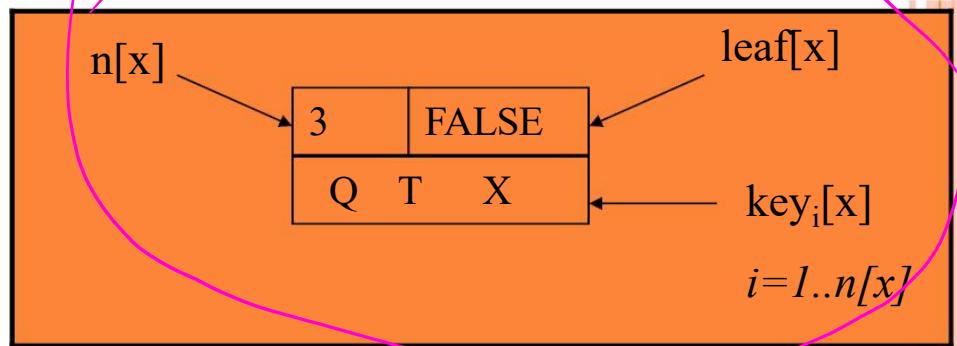
- The B-tree's creators, Rudolf Bayer and Ed McCreight, have not explained what, if anything, the B stands for. The most common belief is that **B stands for *balanced***, as all the leaf nodes are at the same level in the tree. B may also stand for ***Bayer***, or for ***Boeing***, because they were working for *Boeing Scientific Research Labs* at the time.

OUTLINE

- Introduction
- *Definition*
- Basic operations
- Applications



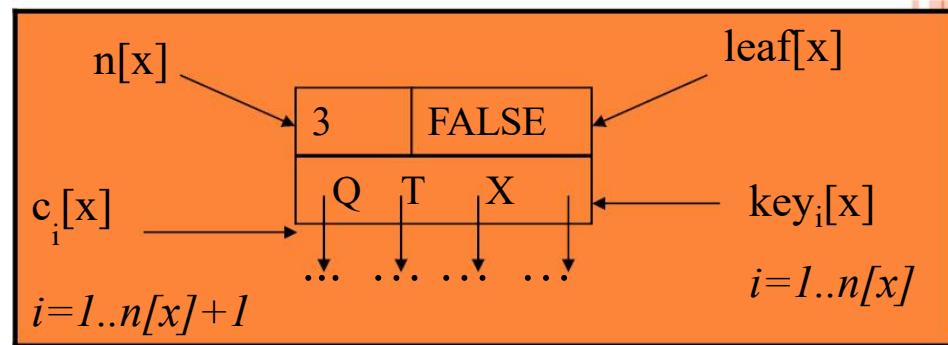
DEFINITION



A **B-tree** T is a rooted tree having the following properties:

- 1 Every node x has the following fields
 - $n[x]$, the number of keys currently stored in node x
 - The $n[x]$ keys themselves stored in nondecreasing order, so that $key_1[x] \leq key_2[x] \leq \dots \leq key_{n[x]}[x]$
 - Leaf[x], a boolean value that is *TRUE* if x is a leaf and *FALSE* if x is an internal node.

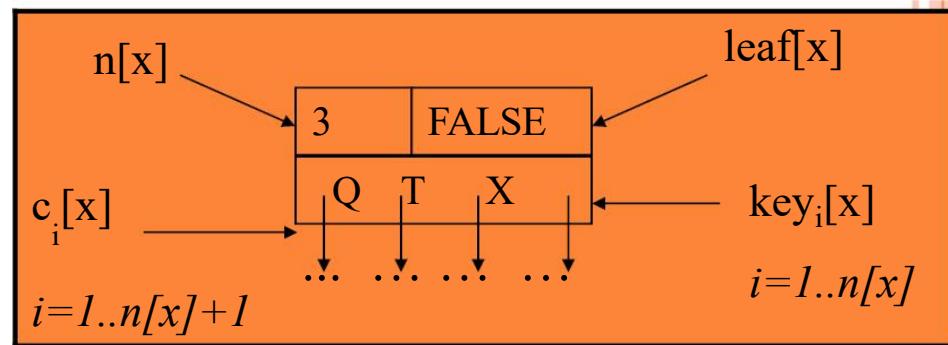
DEFINITION (CONT.)



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

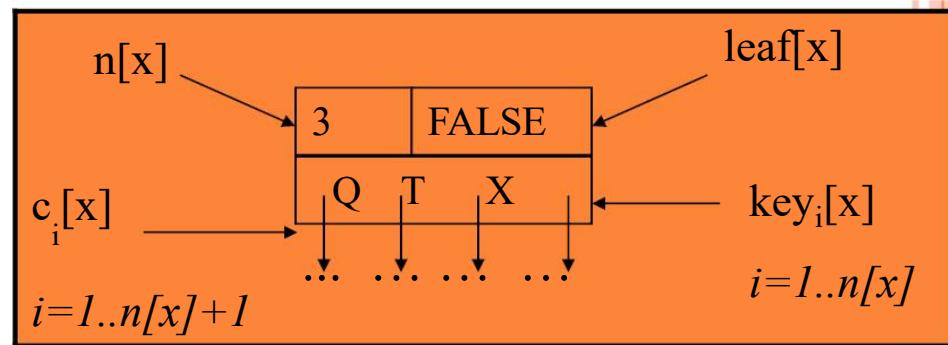
$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \dots \leq key_{n[x]}[x] \leq k_{n[x]+1}$$

DEFINITION (CONT.)



- 4 All leaves have the same depth, which is the tree's height h .
- 5 There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called a ***minimum degree*** of the B-tree:

DEFINITION (CONT.)



- Every node other than the root must have at least $t-1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key
- Every node can contain at most $2t-1$ keys, therefore, an internal node can have at most $2t$ children. We say that a node is **full** if it contains exactly $2t-1$ keys

B-TREE HEIGHT

- $h \leq \log_t((n+1)/2)$ (page 439-440)
- The worst case height is $O(\log n)$. Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, *the base of the logarithm* tends to be large
- Therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

OUTLINE

- Introduction
- Definition
- *Basic operations*
- Applications



BASIC OPERATIONS

- *Searching a B-tree*
- Create an empty B-tree
- Splitting a node
- Inserting a key
- Deleting a key



SEARCH-OVERVIEW

- The search operation on a B-tree is *analogous to a search on a binary tree*.
- Instead of choosing between a left and a right child as in a binary tree, a B-tree search must make an *($n[x] + 1$)-way* choice. The correct child is chosen by performing a linear search of the values in the node.

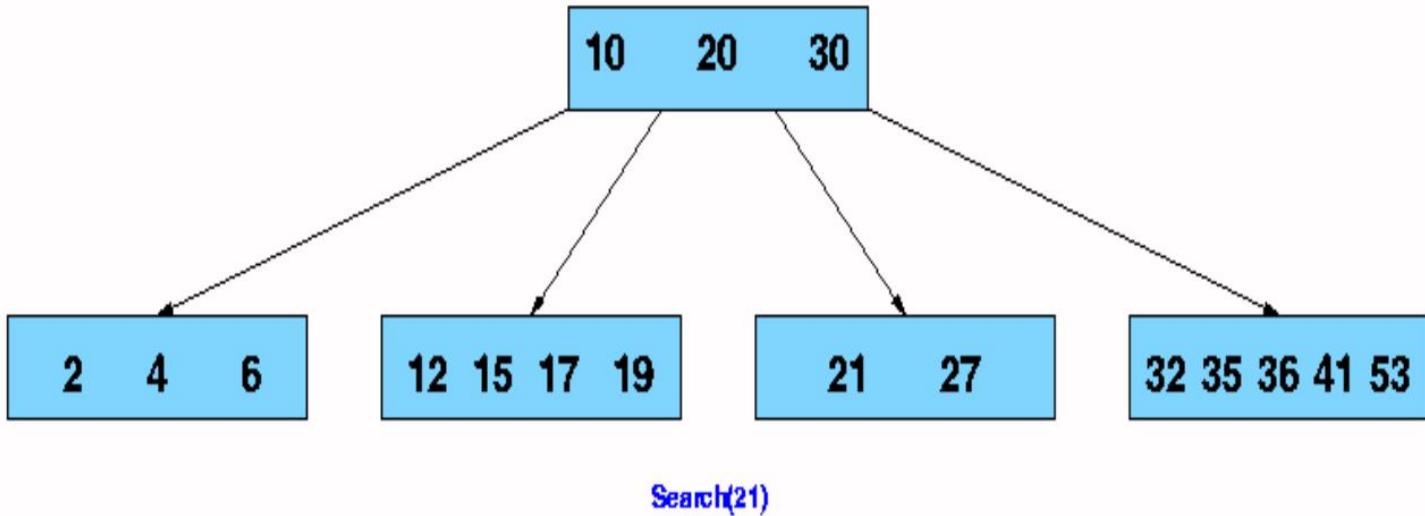
SEARCH-PSEUDOCODE

B-TREE-SEARCH(x, k)

- 1 $i \leftarrow 1$
- 2 **while** $i \leq n[x]$ and $k > key_i[x]$
- 3 **do** $i \leftarrow i + 1$
- 4 **if** $i \leq n[x]$ and $k = key_i[x]$
- 5 **then return** (x, i)
- 6 **if** leaf[x]
- 7 **then return** NIL
- 8 **else** Disk-Read($c_i[x]$)
- 9 **return** B-Tree-Search($c_i[x], k$)

SEARCH-EXAMPLE

B-Tree: Minimization Factor t=3, Minimum Degree = 2, Maximum Degree = 5



SEARCH-ANALYSIS

- After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed.
- Of course, the search can be terminated as soon as the desired node is found.
- $\Theta(\log_t n)$ disk operation
- $O(t \log_t n)$ CPU time

BASIC OPERATIONS

- Searching a B-tree
- *Create an empty B-tree*
- Splitting a node
- Inserting a key
- Deleting a key



CREATE-PSEUDOCODE

B-Tree-Create(T)

```
1 x ← Allocate-Node()
2 leaf[x] ← TRUE
3 n[x] ← 0
4 Disk-Write(x)
5 root[T] ← x
```

CREATE-ANALYSIS

- The *B-Tree-Create* operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node.
- Only the root node is permitted to have these properties; all other nodes must meet the criteria outlined previously.
- $O(1)$ disk operation
- $O(1)$ CPU time

BASIC OPERATIONS

- Searching a B-tree
- Create an empty B-tree
- *Splitting a node*
- Inserting a key
- Deleting a key

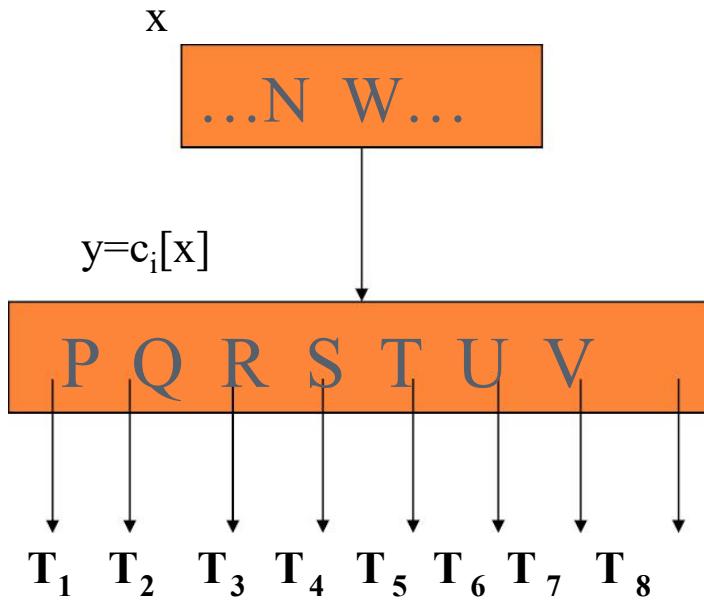


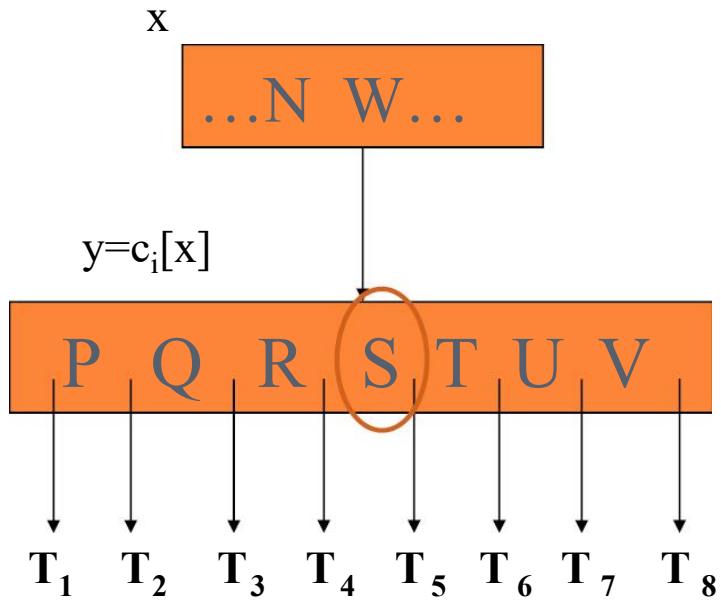
WHY NEED SPLIT

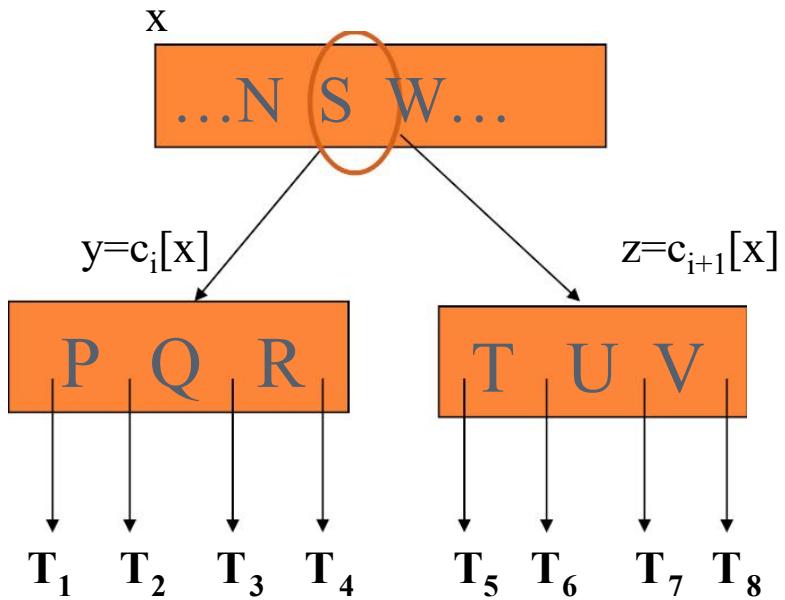
- When inserting a key into a B-tree ,we can't simply create a new leaf node and insert it, as the result tree would fail to be a valid B-tree
- If a node becomes "too full", (having $2t-1$ keys) it is necessary to perform a split operation

SPLIT-OVERVIEW

- It splits a full node y (*having $2t-1$ keys*) around its *median key* $\text{key}_t[y]$ into two nodes having $t-1$ keys each. The median key moves up into y 's parent to identify the dividing point between the two new trees







SPLIT-PSEUDOCODE

B-Tree-Split-Child(x, i, y)

```
1 z←Allocate-Node()
2 leaf[z] ←leaf[y]
3 n[z] ←t - 1
4 for j ←1 to t - 1
5   do keyj[z] ←keyj+t[y]
6 if not leaf[y]
7   then for j ←1 to t
8     do cj[z] ←cj+t[y]
9 n[y] ←t - 1
```

SPLIT-PSEUDOCODE (CONT.)

```
10 for j ← n[x]+ 1 downto i + 1
11   do cj+1[x] ← c [x]
12   ci+1[x] ← z
13 for j ← n[x] downto i
14   do keyj+1[x] ← key [x]
15   keyi[x] ← key [y]
16   n[x] ← n[x] + 1
17 Disk-Write(y)
18 Disk-Write(z)
19 Disk-Write(x)
```

SPLIT-ANALYSIS

- The split operation moves the median key of node y into its parent x where y is the i th child of x . A new node, z , is allocated, and all keys in y right of the median key are moved to z . The keys left of the median key remain in the original node y .
- The new node, z , becomes the child immediately to the right of the median key that was moved to the parent x , and the original node, y , becomes the child immediately to the left of the median key that was moved into the parent x .

SPLIT-ANALYSIS (CONT.)

- The split operation transforms a full node with $2t - 1$ keys into two nodes with $t - 1$ keys each. Note that one key is moved into the parent node.
- $O(1)$ disk operations
- $\theta(t)$ CPU times

BASIC OPERATIONS

- Searching a B-tree
- Create an empty B-tree
- Splitting a node
- *Inserting a key*
- Deleting a key



INSERT-OVERVIEW

- To perform an insertion on a B-tree, the appropriate node for the key must be located using an algorithm similar to *B-Tree-Search*.
- Next, the key must be inserted into the node.
 - If the node is not full prior to the insertion, no special action is required;
 - However, if the node is full, the node must be split to make room for the new key.
- Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node.

INSERT-OVERVIEW (CONT.)

- This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes.
- We don't wait to find out whether we will need to split a full node. Instead, as we travel down the tree searching for the new key belongs ,we split each full node we come to along the way. Thus whenever we want to split a full node y ,we are assumed that its parents is not full.

INSERT-PSEUDOCODE (1)

B-Tree-Insert(T, k)

```
1 r ← root[T]
2 if n[r] = 2t - 1
3   then s ← Allocate-Node()
4     root[T] ← s
5     leaf[s] ← FALSE
6     n[s] ← 0
7     c1 ← r
8     B-Tree-Split-Child(s, 1, r)
9     B-Tree-Insert-Nonfull(s, k)
10  else B-Tree-Insert-Nonfull(r, k)
```

INSERT-PSEUDOCODE (2)

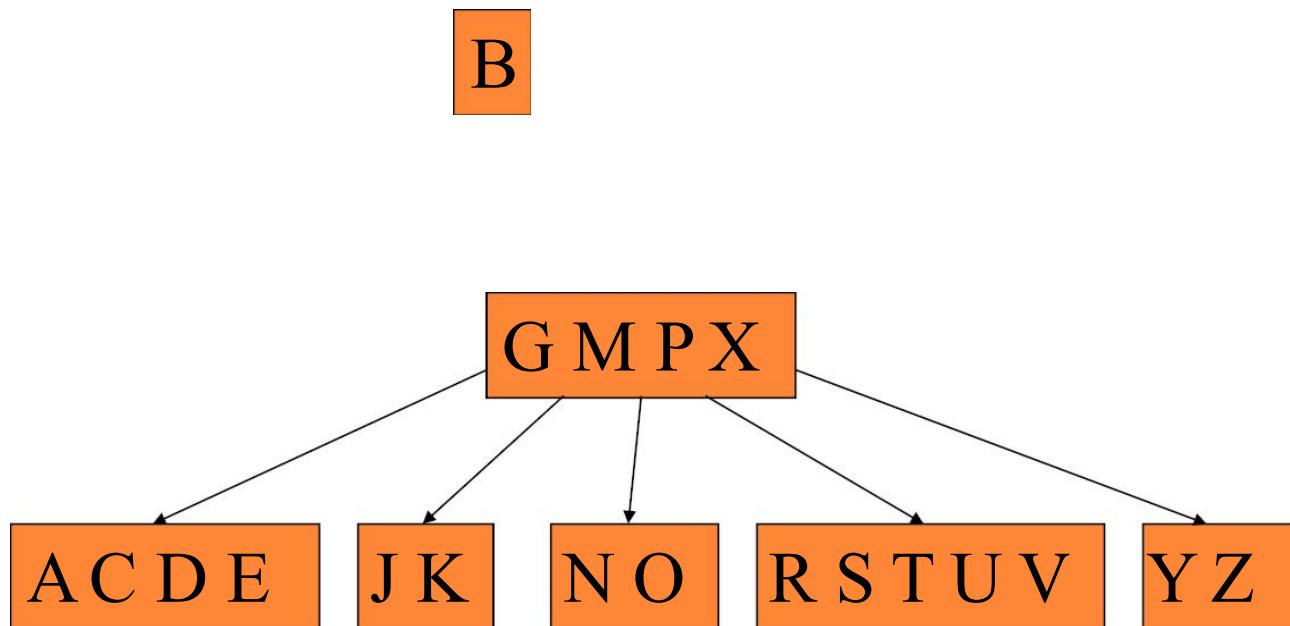
B-Tree-Insert-Nonfull(x, k)

- 1 $i \leftarrow n[x]$
- 2 **if** $\text{leaf}[x]$
- 3 **then while** $i \geq 1$ and $k < \text{key}_i[x]$
- 4 **do** $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$
- 5 $i \leftarrow i - 1$
- 6 $\text{key}_{i+1}[x] \leftarrow k$
- 7 $n[x] \leftarrow n[x] + 1$
- 8 Disk-Write(x)

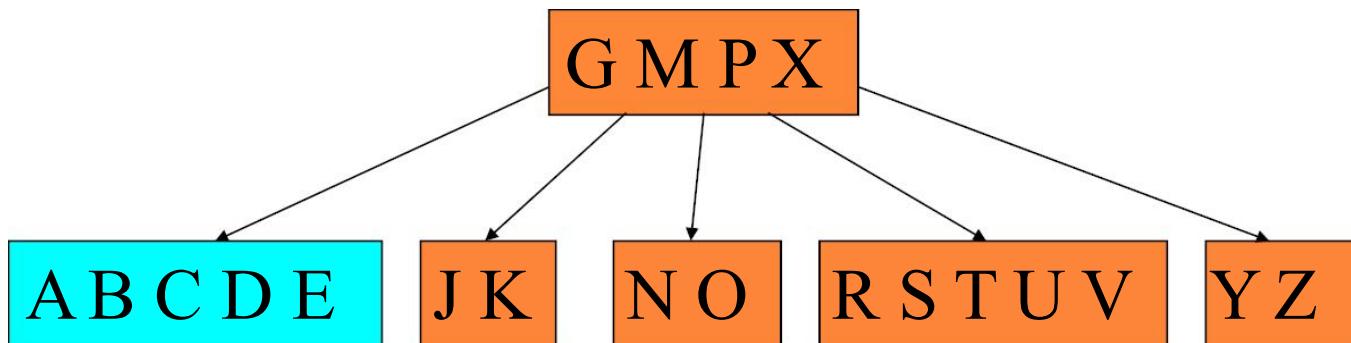
INSERT-PSEUDOCODE (2 CONT.)

```
9   else while i ≥ 1 and k < keyi[x]
10      do i ← i - 1
11      i ← i + 1
12      Disk-Read(ci[x])
13      if n[ci[x]] = 2t - 1
14         then B-Tree-Split-Child(x, i, ci[x])
15         if k > keyi[x]
16            then i ← i + 1
17            B-Tree-Insert-Nonfull(ci[x], k)
```

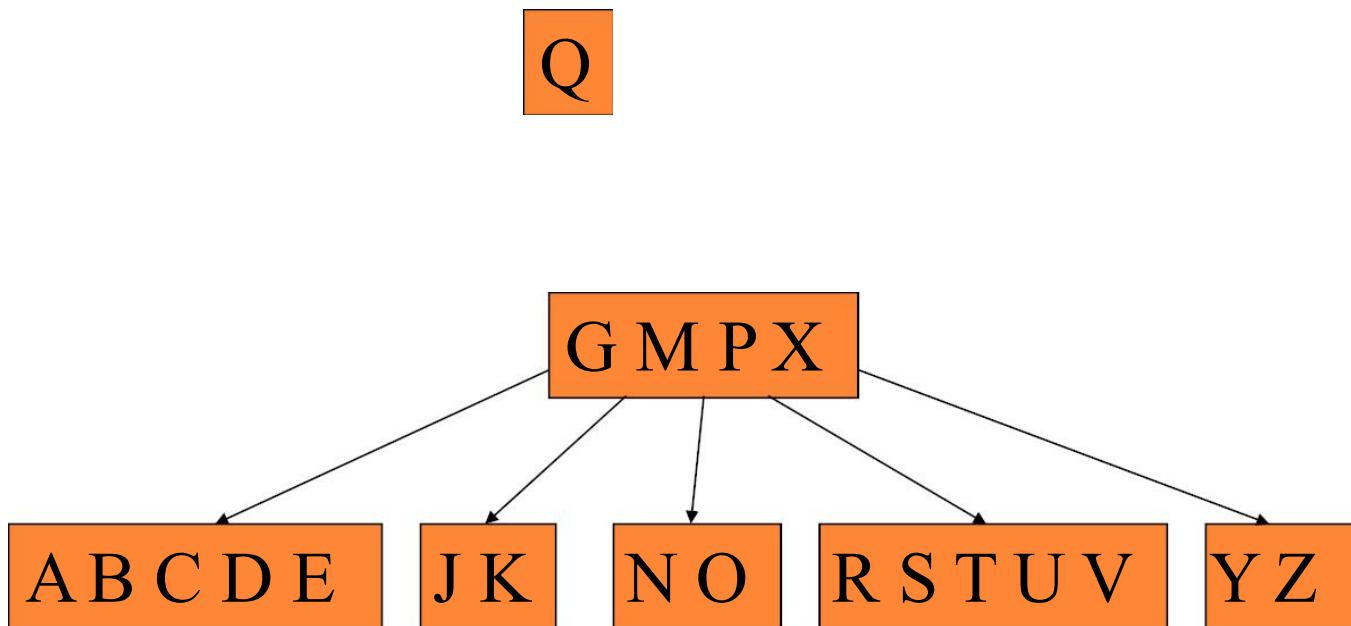
Insert B



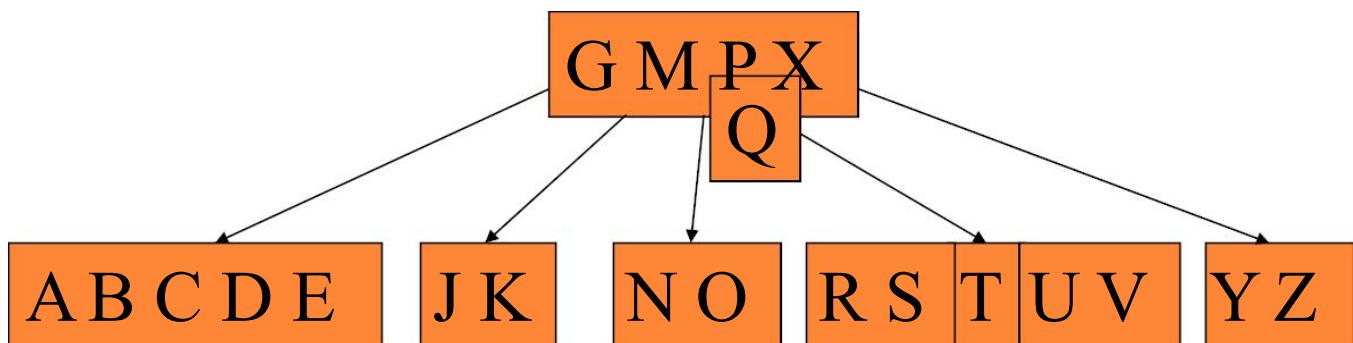
Insert B



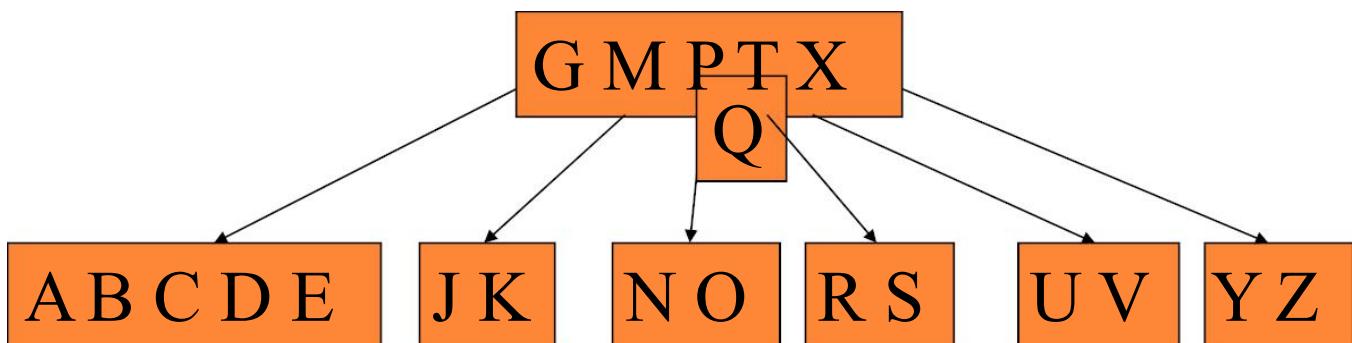
Insert Q



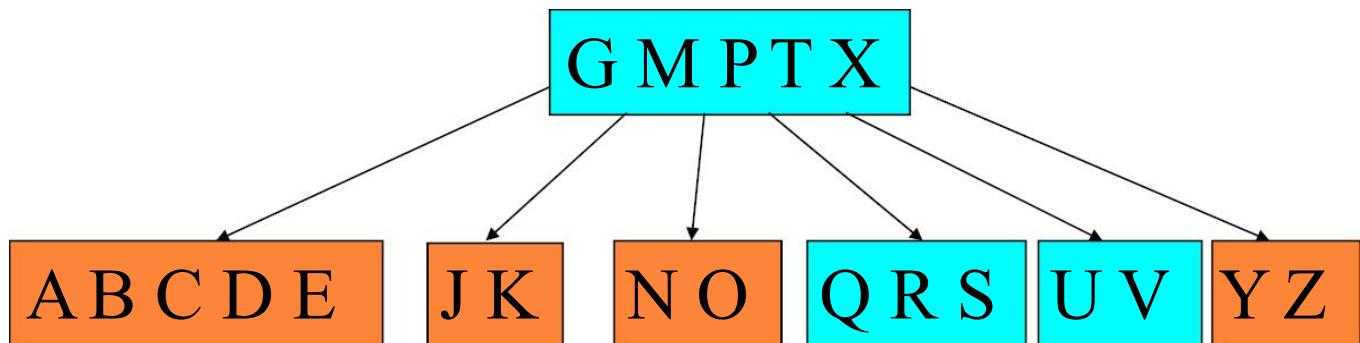
Insert Q



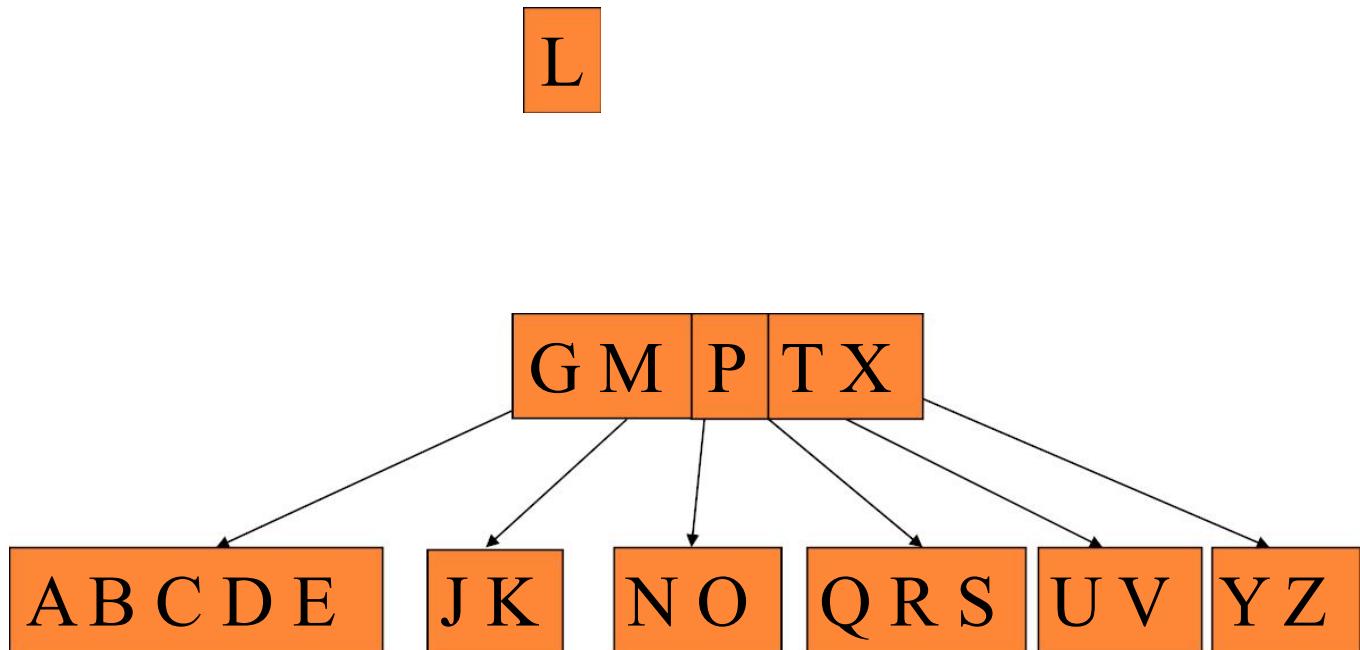
Insert Q



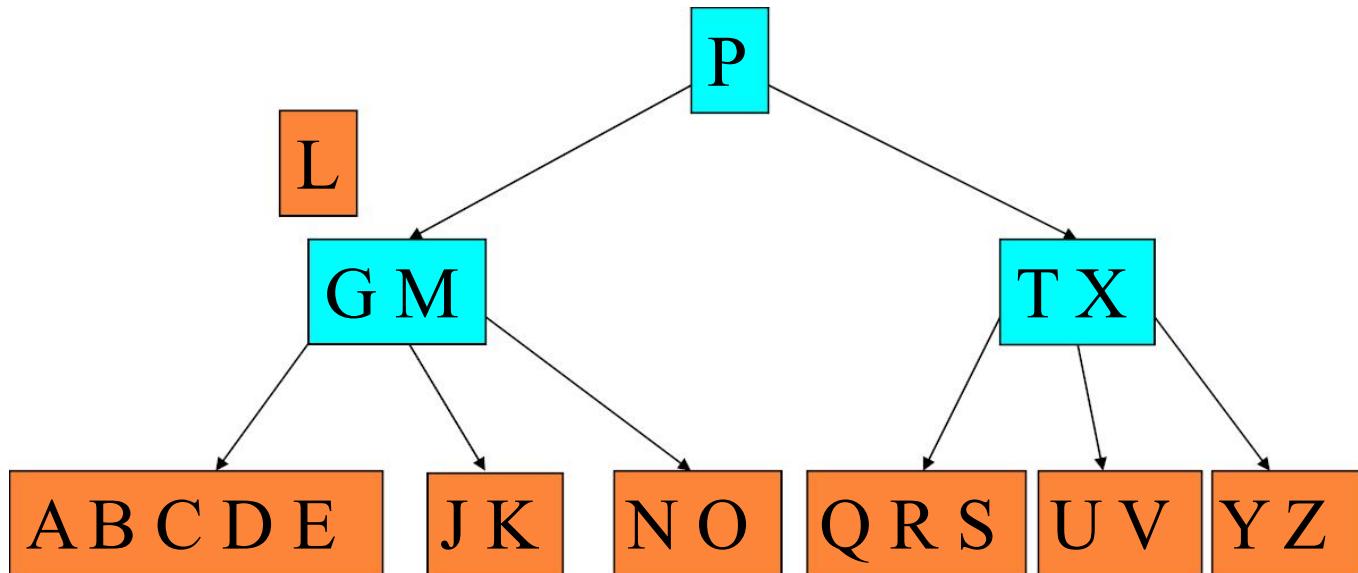
Insert Q



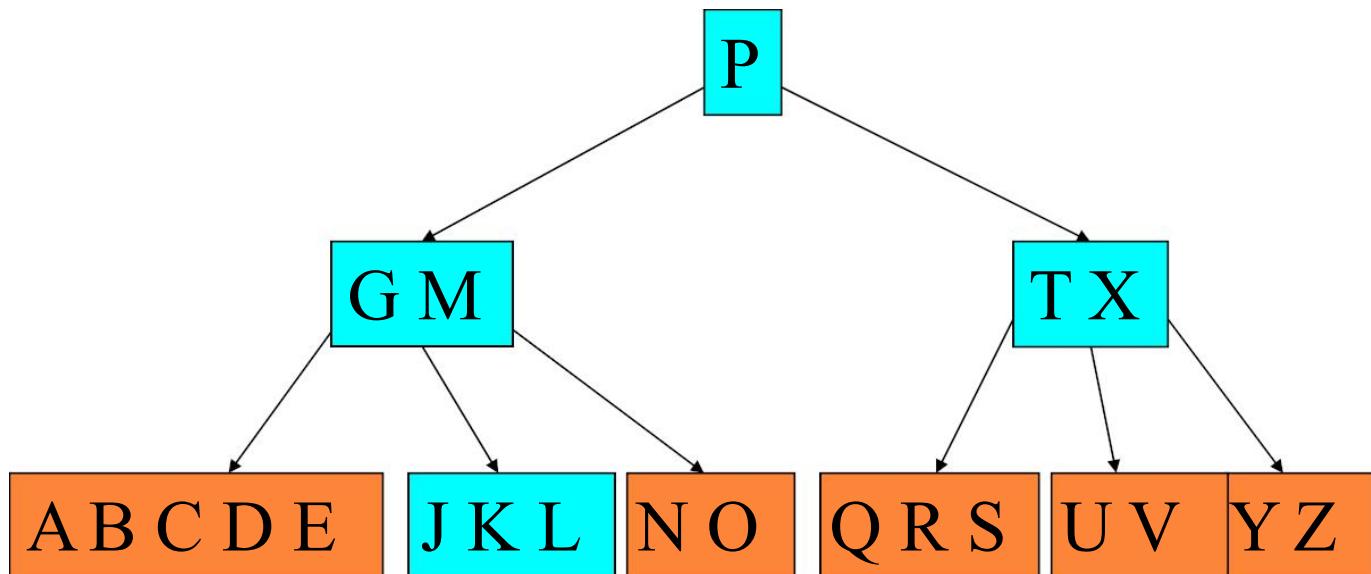
Insert L



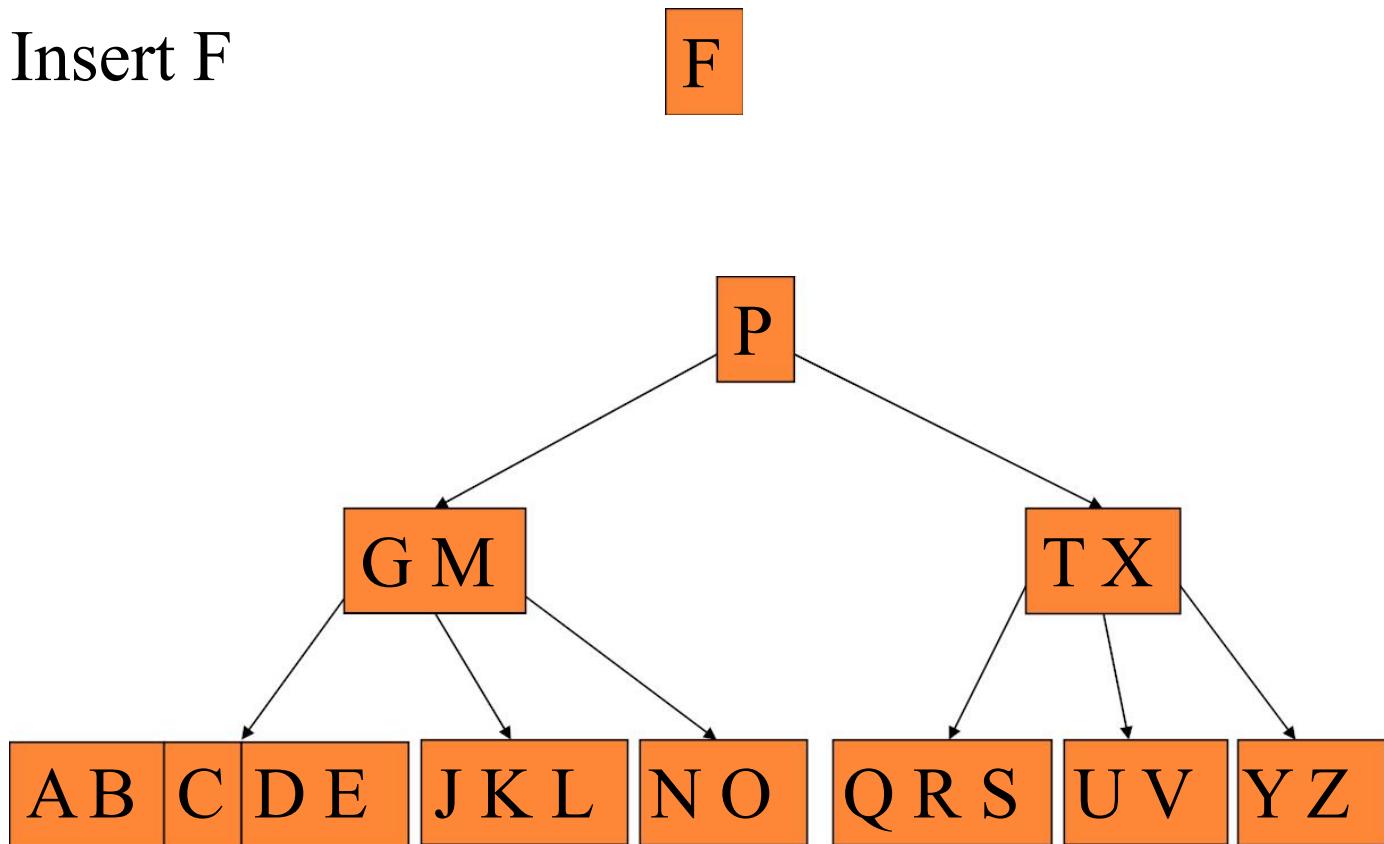
Insert L



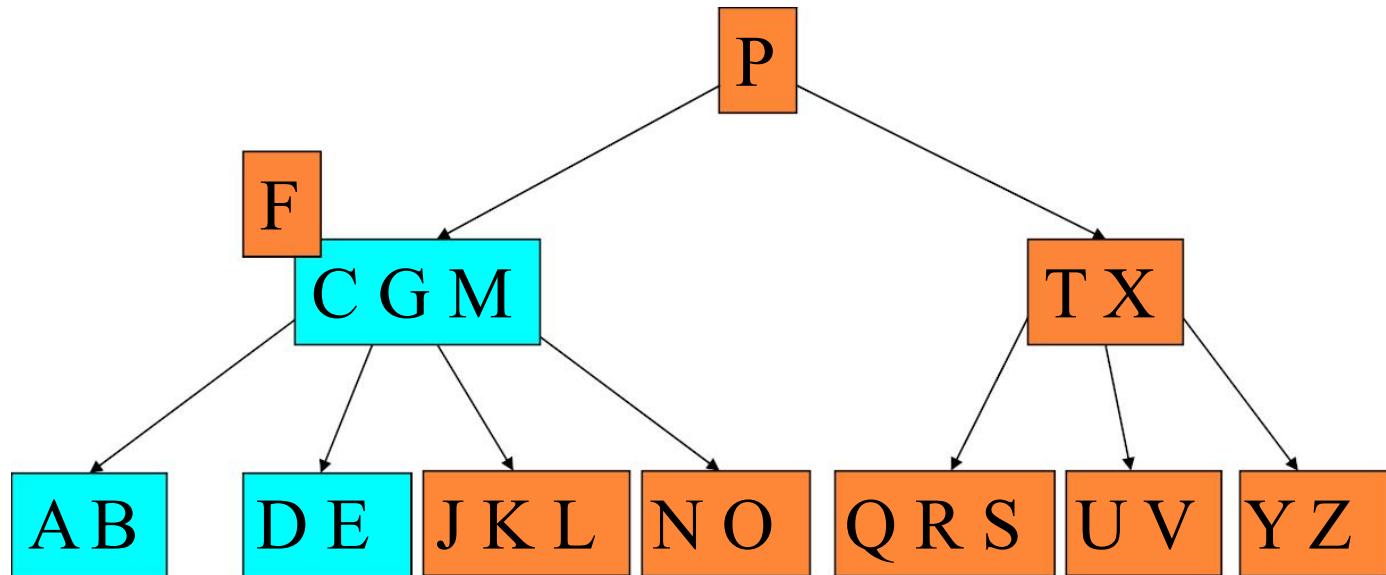
Insert L



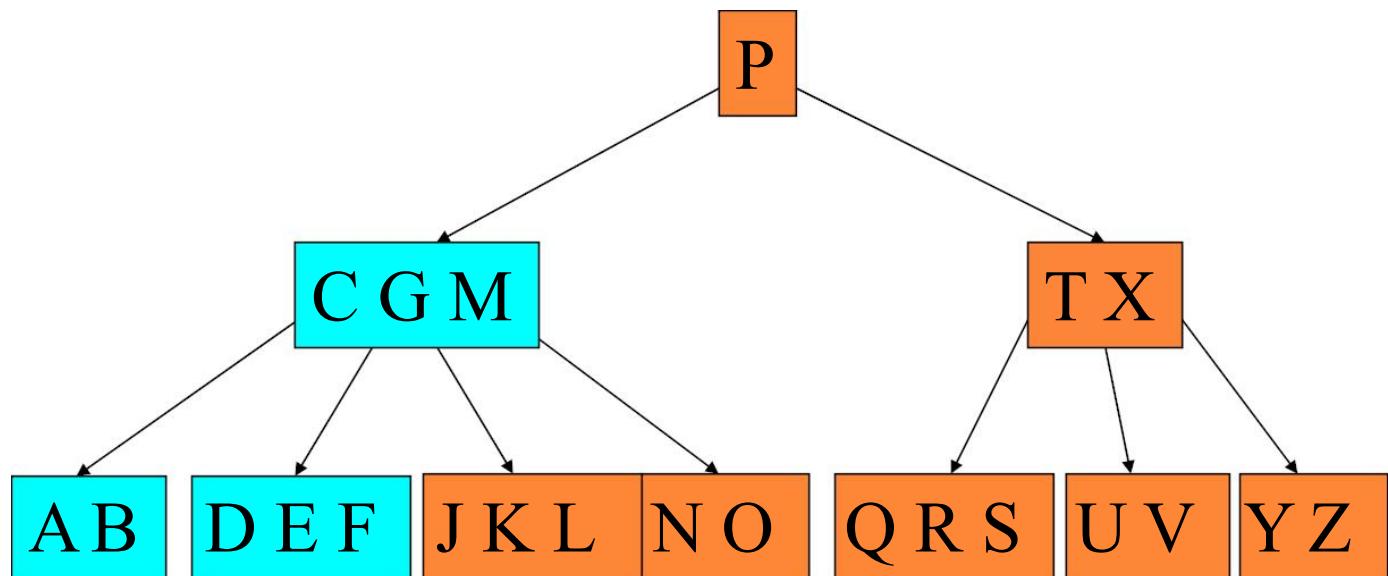
Insert F



Insert F



Insert F



INSERT-ANALYSIS

- Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never full. To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree.
- Although this approach may result in unnecessary split operations, it guarantees that the parent never needs to be split and eliminates the need for a second pass up the tree. Since a split runs in linear time, it has little effect on the $O(t \log_t n)$ running time of *B-Tree-Insert*.

BASIC OPERATIONS

- Searching a B-tree
- Create an empty B-tree
- Splitting a node
- Inserting a key
- *Deleting a key*



DELETE-OVERVIEW

- There are two popular strategies for deletion from a B-Tree.
 - locate and delete the item, then restructure the tree to regain its invariants
 - do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

DELETE-CASES

- 1. If the key k is in node x and x is a leaf, delete the key k from x . [example](#)
- 2. If the key k is in node x and x is an internal node, do the following.
 - a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y . Recursively delete k' , and replace k by k' in x . (Finding k' and deleting it can be performed in a single downward pass.) [example](#)

DELETE-CASES (CONT.)

- b. Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z . Recursively delete k , and replace k by k' in x . (Finding k' and deleting it can be performed in a single downward pass.)
- c. Otherwise, if both y and z have only $t - 1$ keys, merge k and all of z into y , so that x loses both k and the pointer to z , and y now contains $2t - 1$ keys. Then, free z and recursively delete k from y . example

DELETE-CASES (CONT.)

- 3. If the key k is not present in internal node x , determine the root $ci[x]$ of the appropriate subtree that must contain k , if k is in the tree at all. If $ci[x]$ has only $t - 1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x .
 - a. If $ci[x]$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $ci[x]$ an extra key by moving a key from x down into $ci[x]$, moving a key from $ci[x]$'s immediate left or right sibling up into x , and moving the appropriate child pointer from the sibling into $ci[x]$. [example](#)

DELETE-CASES (CONT.)

- b. If $ci[x]$ and both of $ci[x]$'s immediate siblings have $t - 1$ keys, merge $ci[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node. [example](#)

VARIETIES

- **2-3-4 tree** is a B-tree of order 4
- **2-3 tree** is a B-tree that can contain only 2-nodes (a node with 1 field and 2 children) and 3-nodes (a node with 2 fields and 3 children)
- **B+ tree** (also known as a Quarternary Tree) is a variety of B-tree, in contrast to a B-tree, all records are stored at the lowest level of the tree; only keys are stored in interior blocks.
- **B*-tree** is a tree data structure, a variety of B-tree used in the HFS and Reiser4 file systems, which requires nonroot nodes to be at least $2/3$ full instead of $1/2$

OUTLINE

- Introduction
- Definition
- Basic operations
- *Applications*

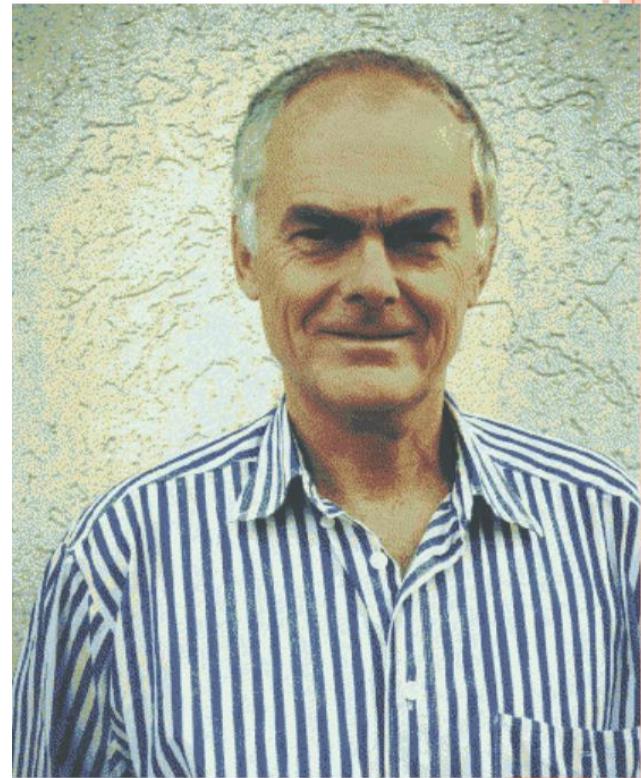


APPLICATIONS

- As databases cannot typically be maintained entirely in memory, b-trees are often used to index the data and to provide fast access. For example, searching an unindexed and unsorted database containing n key values will have a worst case running time of $O(n)$
- *Search Engine Index*
-

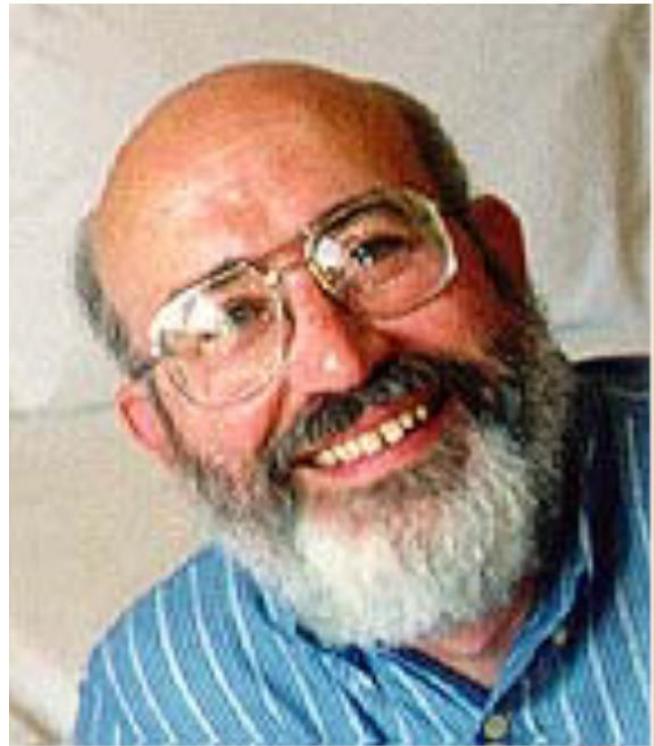
RUDOLF BAYER

- Professor (emeritus) of Informatics at the Technical University of Munich since 1972
- Famous for inventing two data sorting structures: the **B-tree** with Edward M. McCreight, and later the **UB-tree** with Volker Markl
- a recipient of 2001 ACM SIGMOD Edgar F. Codd Innovations Award



EDWARD M. MCCREIGHT

- Attended the College of Wooster ('66) and Carnegie-Mellon University
- Have worked at Boeing Aircraft and Xerox PARC
- Guest professor at the University of Washington, Stanford University, the Technical University of Munich, and the Swiss Federal Institute of Technology in Zürich



SKIP LISTS

Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

OUTLINE

Skip Lists

- Data structure
- Randomized insertion
- With- high- probability bound
- Analysis
- Coin flipping



Skip lists

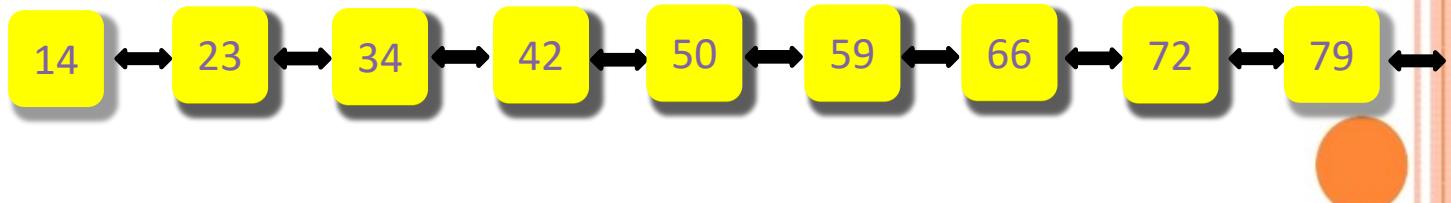
- Simple randomized dynamic search structure
 - Invented by William Pugh in 1989
 - Easy to implement
- Maintains a dynamic set of n elements in $\underline{O(\lg n)}$ time per operation in expectation and with high probability
 - Strong guarantee on tail of distribution of $T(n)$
 - $O(\lg n)$ “almost always”



One linked list

Start from simplest data structure:
(sorted) linked list

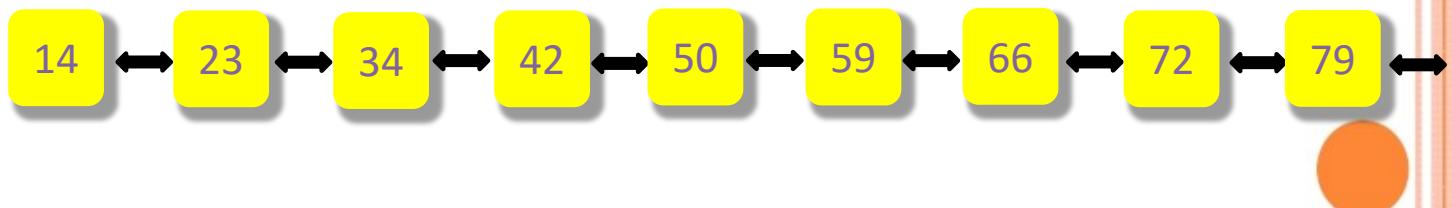
- Searches take $\Theta(n)$ time in worst case
- How can we speed up searches?



Two linked lists

Suppose we had two sorted linked lists
(on subsets of the elements)

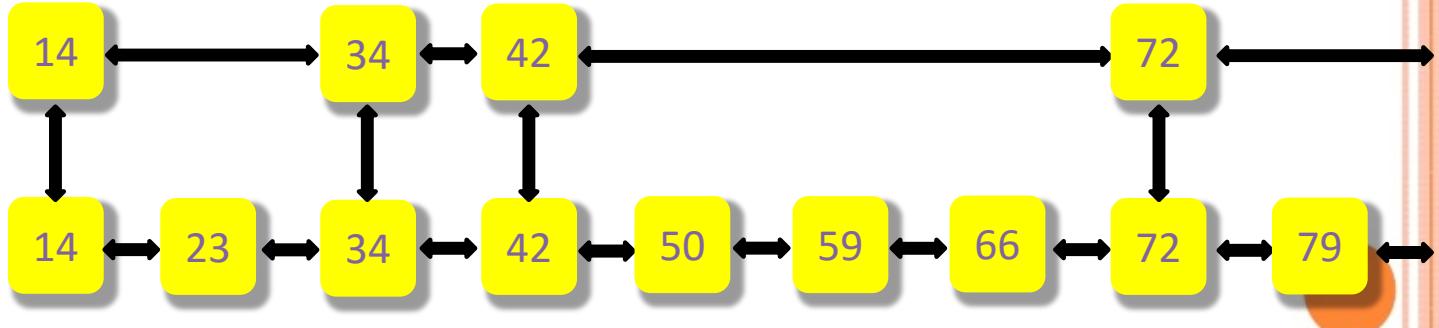
- Each element can appear in one or both lists
- How can we speed up searches?



Two linked lists as a subway

IDEA: Express and local subway lines
(à la New York City 7th Avenue Line)

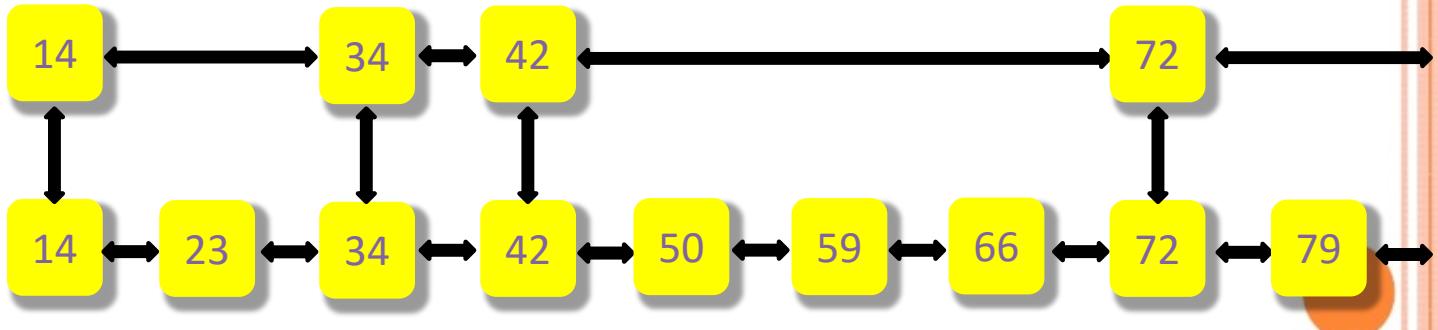
- Express line connects a few of the stations
- Local line connects all stations
- Links between lines at common stations



Searching in two linked lists

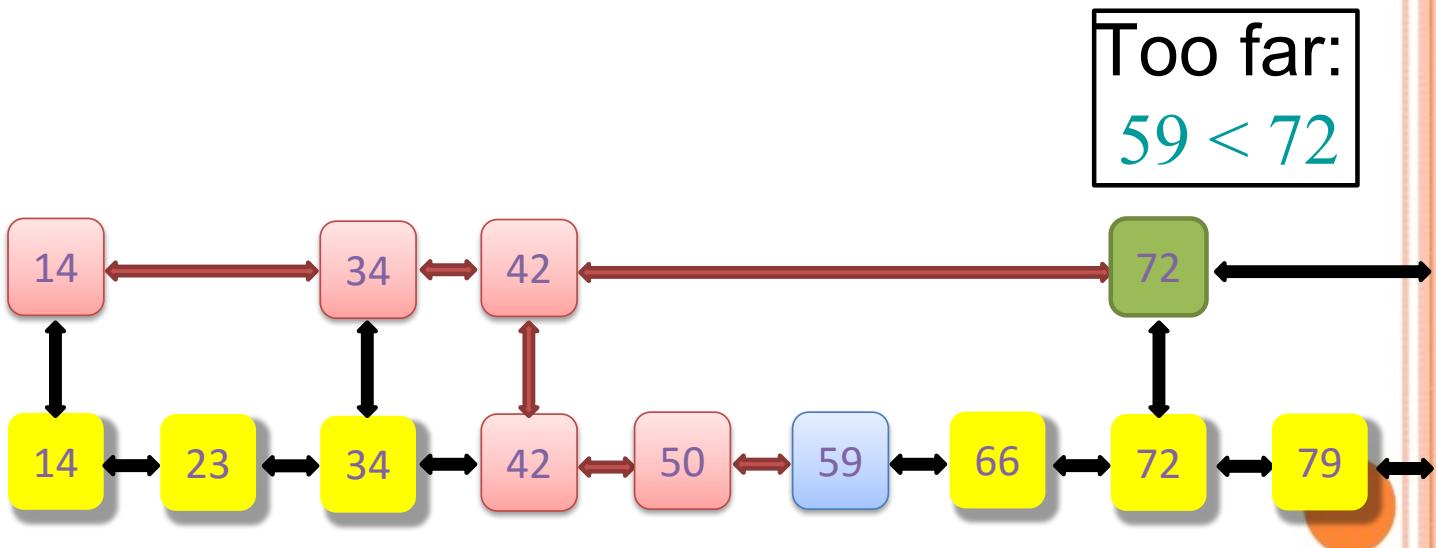
SEARCH(x):

- Walk right in top linked list (L_1) until going right would go too far
- Walk down to bottom linked list (L_2)
- Walk right in L_2 until element found (or not)



Searching in two linked lists

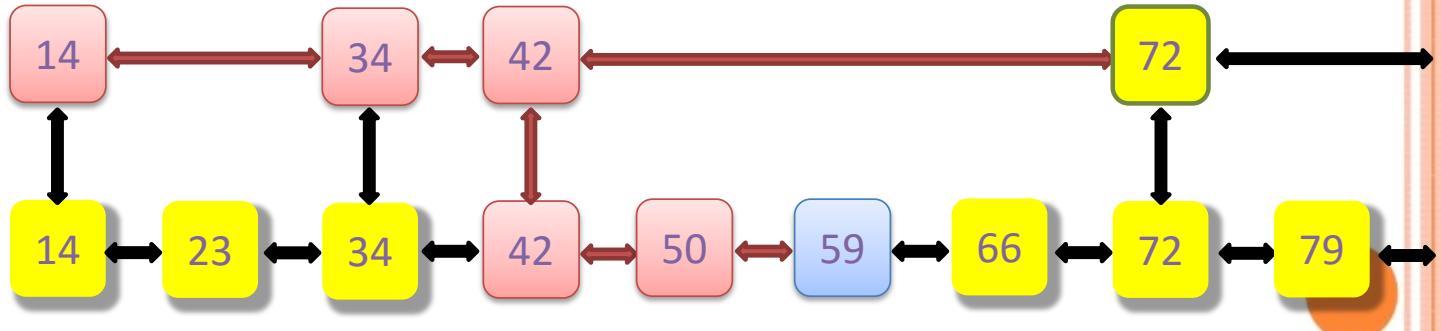
EXAMPLE: SEARCH(59)



Design of two linked lists

QUESTION: Which nodes should be in L_1 ?

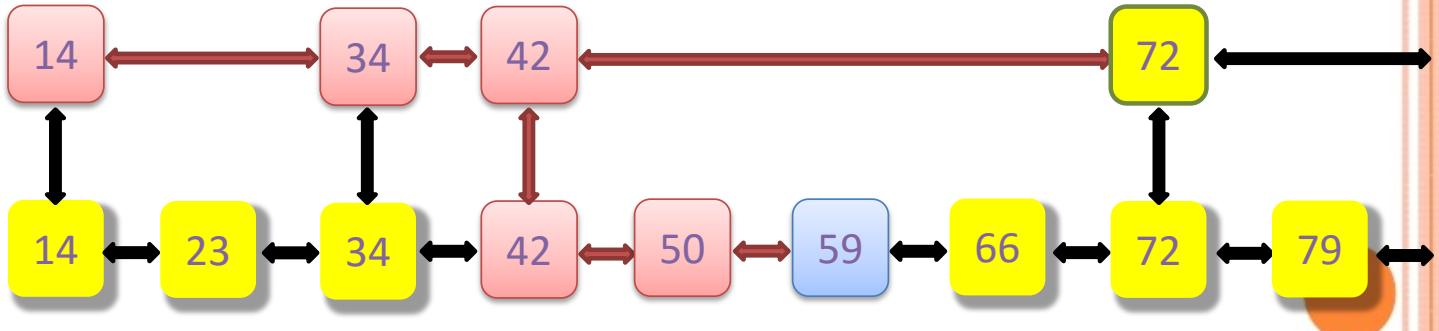
- In a subway, the “popular stations”
- Here we care about worst-case performance
- **Best approach:** Evenly space the nodes in L_1
- But how many nodes should be in L_1 ?



Analysis of two linked lists

ANALYSIS:

- Search cost is roughly $|L_1| + \frac{|L_2|}{|L_1|}$
- Minimized (up to constant factors) when terms are equal
- $|L_1|^2 = |L_2| = n \Rightarrow |L_1| = \sqrt{n}$

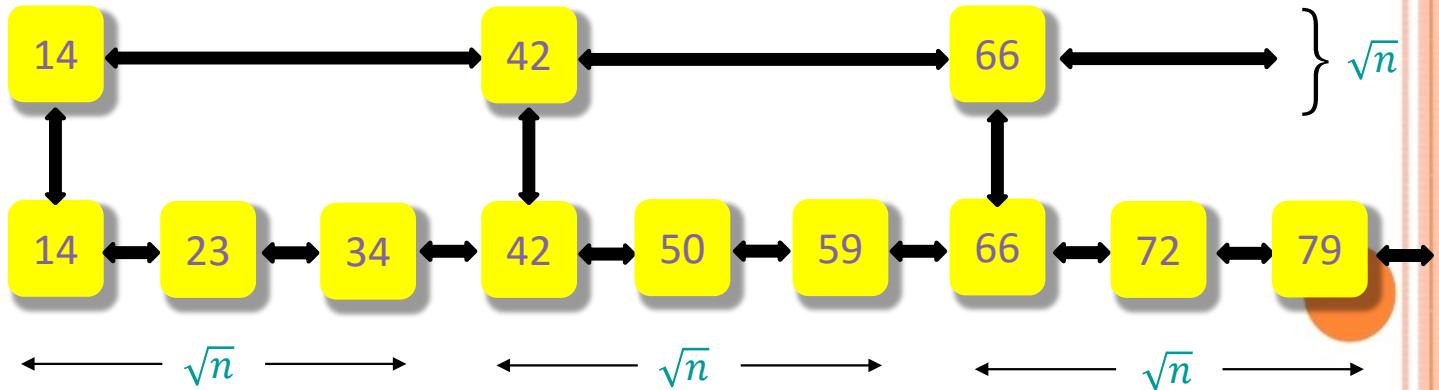


Analysis of two linked lists

ANALYSIS:

- $|L_1| = \sqrt{n}$, $|L_2| = n$
- Search cost is roughly

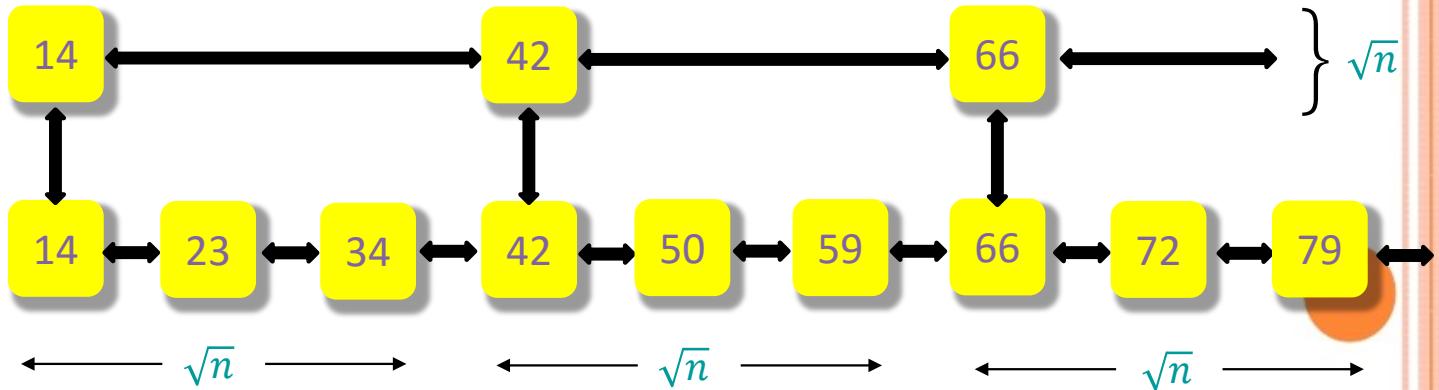
$$|L_1| + \frac{|L_2|}{|L_1|} = \sqrt{n} + \frac{n}{\sqrt{n}} = 2\sqrt{n}$$



More linked lists

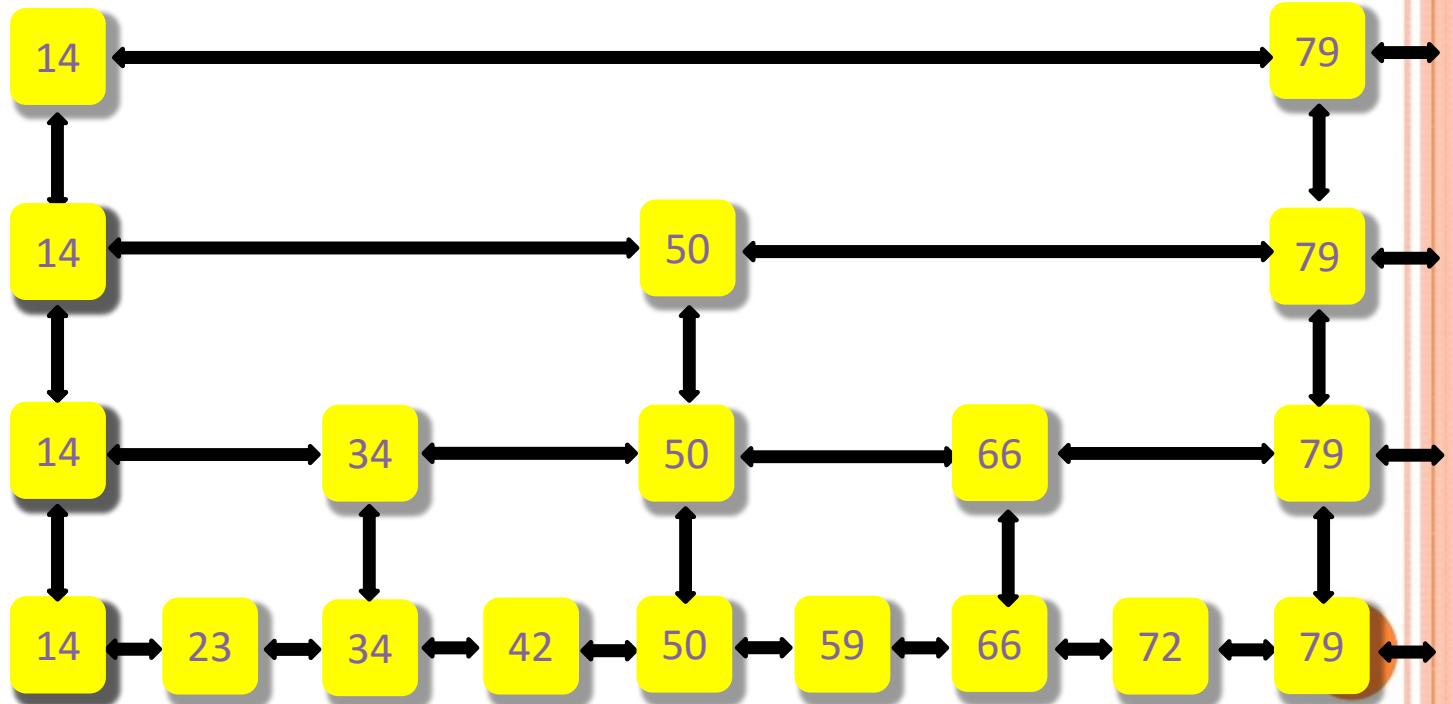
What if we had more sorted linked lists?

- 2 sorted lists $\Rightarrow 2 \cdot \sqrt{n}$
- 3 sorted lists $\Rightarrow 3 \cdot \sqrt[3]{n}$
- k sorted lists $\Rightarrow k \cdot \sqrt[k]{k}$
- $\lg n$ sorted lists $\Rightarrow \lg n \cdot \sqrt[\lg n]{n} = 2 \lg n$



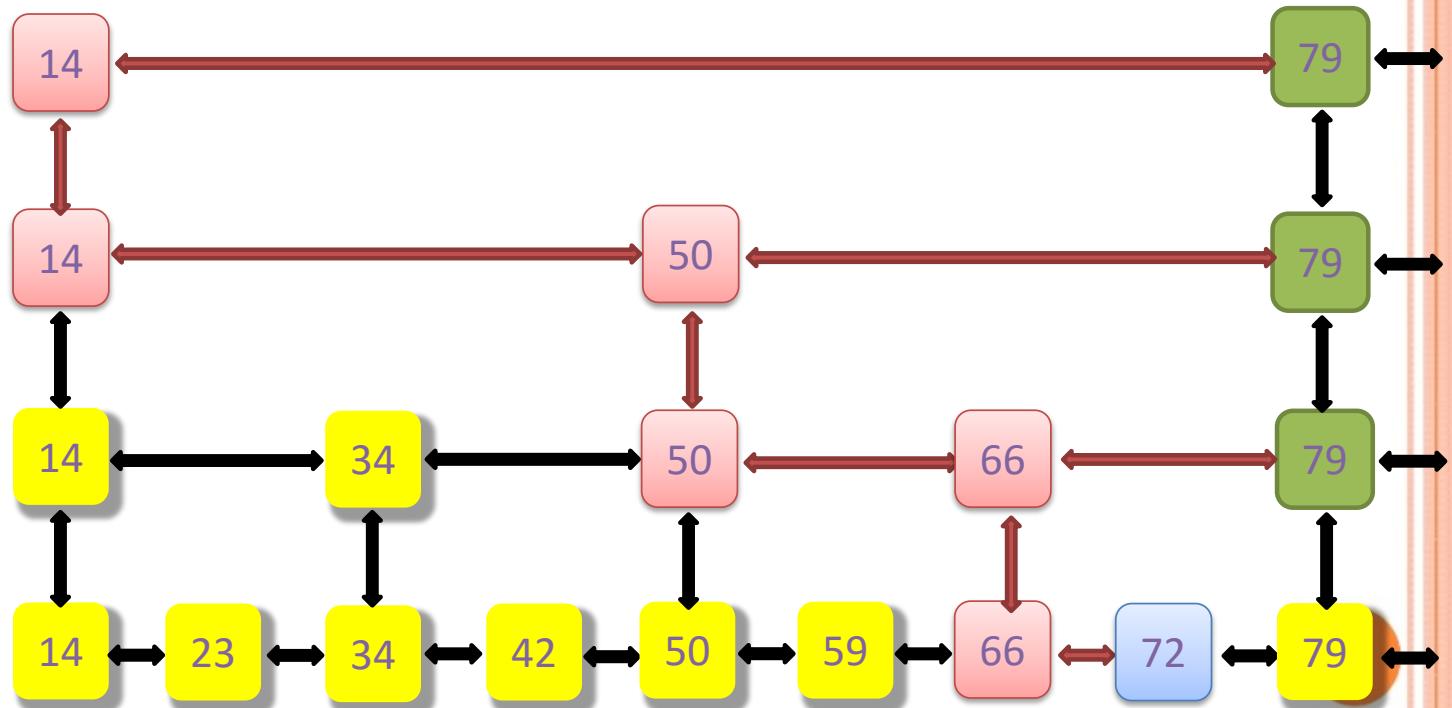
$\lg n$ linked lists

$\lg n$ sorted linked lists are like a binary tree
(in fact, level-linked B^+ -tree; see Problem Set 5)



Searching in $\lg n$ linked lists

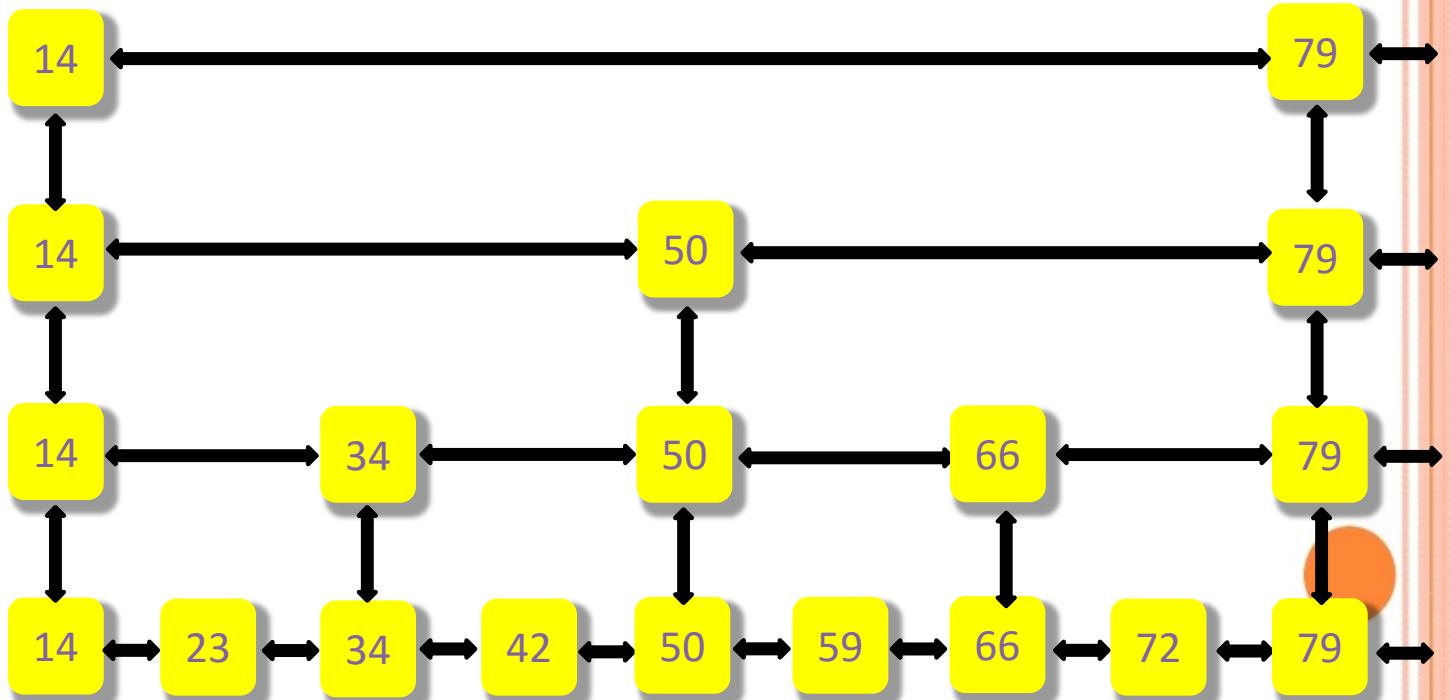
EXAMPLE: SEARCH(72)



Skip lists

Ideal skip list is this $\lg n$ linked list structure

Skip list data structure maintains roughly this structure subject to updates (insert/delete)



INSERT(x)

To insert an element x into a skip list:

- SEARCH(x) to see where x fits in bottom list
- Always insert into bottom list

INVARIANT: Bottom list contains all elements

- Insert into some of the lists above...

QUESTION: To which other lists should we add x ?

INSERT(x)

QUESTION: To which other lists should we add x ?

IDEA: Flip a (fair) coin; if HEADS,
promote x to next level up and flip again

- Probability of promotion to next level = $1/2$
- On average:
 - $1/2$ of the elements promoted 0 levels
 - $1/4$ of the elements promoted 1 level
 - $1/8$ of the elements promoted 2 levels
 - etc.

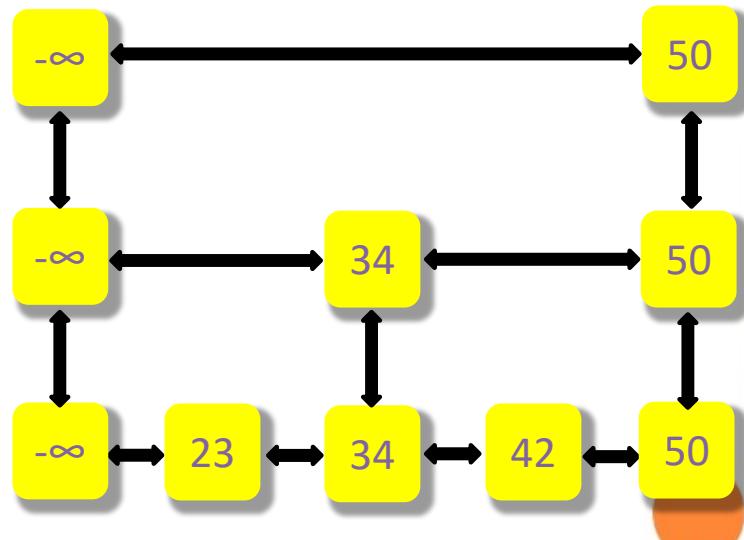


Example of skip list

EXERCISE: Try building a skip list from scratch by repeated insertion using a real coin

Small change:

- Add special $-\infty$ value to every list
⇒ can search with the same algorithm



Skip lists

A **skip list** is the result of insertions (and deletions) from an initially empty structure (containing just $-\infty$)

- $\text{INSERT}(x)$ uses random coin flips to decide promotion level
- $\text{DELETE}(x)$ removes x from all lists containing it



Skip lists

A **skip list** is the result of insertions (and deletions) from an initially empty structure (containing just $-\infty$)

- INSERT(x) uses random coin flips to decide promotion level
- DELETE(x) removes x from all lists containing it

How good are skip lists? (speed/balance)

- **INTUITIVELY:** Pretty good on average
- **CLAIM:** Really, really good, almost always

With-high-probability theorem

THEOREM: With high probability, every search in an n -element skip list costs $O(\lg n)$



With-high-probability theorem

THEOREM: With high probability, every search in a skip list costs $O(\lg n)$

- **INFORMALLY:** Event E occurs with high probability (w.h.p.) if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which E occurs with probability at least $1 - O(1/n^\alpha)$
 - In fact, constant in $O(\lg n)$ depends on α
- **FORMALLY:** Parameterized event E_α occurs with high probability if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which E_α occurs with probability at least $1 - c_\alpha/n^\alpha$

With-high-probability theorem

THEOREM: With high probability, every search in a skip list costs $O(\lg n)$

- **INFORMALLY:** Event E occurs with high probability (w.h.p.) if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which E occurs with probability at least $1 - O(1/n^\alpha)$
- **IDEA:** Can make error probability $O(1/n^\alpha)$
- ~~Very small by setting α large e.g. 100~~ Almost certainly, bound remains true for entire execution of polynomial-time algorithm

Boole's inequality / union bound

Recall:

BOOLE'S INEQUALITY / UNION BOUND:

For any random events E_1, E_2, \dots, E_k ,

$$\Pr\{E_1 \cup E_2 \cup \dots \cup E_k\} \leq \Pr\{E_1\} + \Pr\{E_2\} + \dots + \Pr\{E_k\}$$

Application to with-high-probability events:

If $k = n^{O(1)}$, and each E_i occurs with high probability, then so does $E_1 \cap E_2 \cap \dots \cap E_k$

Analysis Warmup

LEMMA: With high probability,
 n -element skip list has $O(\lg n)$ levels

PROOF:

- Error probability for having at most $c \lg n$ levels
= $\Pr\{\text{more than } c \lg n \text{ levels}\}$
 $\leq n \cdot \Pr\{\text{element } x \text{ promoted at least } c \lg n \text{ times}\}$
(by Boole's Inequality)
= $n \cdot (1/2^{c \lg n})$
= $n \cdot (1/n^c)$
= $1/n^{c-1}$

Analysis Warmup

LEMMA: With high probability,
 n -element skip list has $O(\lg n)$ levels

PROOF:

- Error probability for having at most $c \lg n$ levels
 $\leq 1/n^{c-1}$
- This probability is polynomially small,
i.e., at most n^α for $\alpha = c - 1$.
- We can make α arbitrarily large by choosing the
constant c in the $O(\lg n)$ bound accordingly.

Proof of theorem

THEOREM: With high probability, every search in an n -element skip list costs $O(\lg n)$

COOL IDEA: Analyze search backwards—leaf to root

- Search starts [ends] at leaf (node in bottom level)
- At each node visited:
 - If node wasn't promoted higher (got TAILS here), then we go [came from] left
 - If node was promoted higher (got HEADS here), then we go [came from] up
- Search stops [starts] at the root (or $-\infty$)

Proof of theorem

THEOREM: With high probability, every search in an n -element skip list costs $O(\lg n)$

COOL IDEA: Analyze search backwards—leaf to root

PROOF:

- Search makes “up” and “left” moves until it reaches the root (or $-\infty$)
- Number of “up” moves < number of levels
 $\leq c \lg n$ w.h.p. (Lemma)
- \Rightarrow w.h.p., number of moves is at most the number of times we need to flip a coin to get $c \lg n$ HEADS

Coin flipping analysis

CLAIM: Number of coin flips until $c \lg n$ HEADS
 $= \Theta(\lg n)$ with high probability

PROOF:

Obviously $\Omega(\lg n)$: at least $c \lg n$

Prove $O(\lg n)$ “by example”:

- Say we make $10 c \lg n$ flips
- When are there at least $c \lg n$ HEADS?

(Later generalize to arbitrary values of 10)



Coin flipping analysis

CLAIM: Number of coin flips until $c \lg n$ HEADS
 $= \Theta(\lg n)$ with high probability

PROOF:

- $\Pr\{\text{exactly } c \lg n \text{ HEADS}\} = \underbrace{\binom{10c \lg n}{c \lg n}}_{\text{orders}} \cdot \underbrace{\left(\frac{1}{2}\right)^{c \lg n}}_{\text{HEADS}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c \lg n}}_{\text{TAILS}}$
- $\Pr\{\text{at most } c \lg n \text{ HEADS}\} \leq \underbrace{\binom{10c \lg n}{c \lg n}}_{\text{overestimate on orders}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c \lg n}}_{\text{TAILS}}$

Coin flipping analysis (cont'd)

- Recall bounds on $\binom{y}{x}$: $\left(\frac{y}{x}\right)^x \leq \binom{y}{x} \leq \left(e \frac{y}{x}\right)^x$
- $\Pr\{\text{at most } c \lg n \text{ HEADS}\} \leq \binom{10c \lg n}{c \lg n} \cdot \left(\frac{1}{2}\right)^{9c \lg n}$
$$\leq \left(e \frac{10c \lg n}{c \lg n}\right)^{c \lg n} \cdot \left(\frac{1}{2}\right)^{9c \lg n}$$
$$= (10e)^{c \lg n} 2^{-9c \lg n}$$
$$= 2^{\lg(10e) \cdot c \lg n} 2^{-9c \lg n}$$
$$= 2^{[\lg(10e) - 9] \cdot c \lg n}$$
$$= 1/n^\alpha$$

for $\alpha = [9 - \lg(10e)] \cdot c$



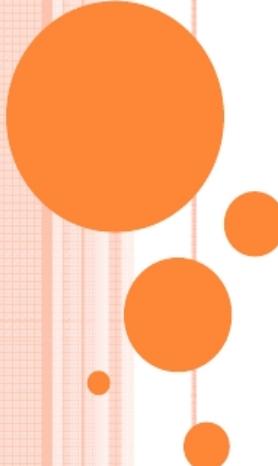
Coin flipping analysis (cont'd)

- $\Pr\{\text{at most } c \lg n \text{ HEADS}\} \leq 1/n^\alpha$ for $\alpha = [9 - \lg(10e)]c$
- **KEY PROPERTY:** $\alpha \rightarrow \infty$ as $10 \rightarrow \infty$, for any c
- So set 10 , i.e., constant in $O(\lg n)$ bound, large enough to meet desired α

This completes the proof of the coin-flipping claim and the proof of the theorem.



AMORTIZED ANALYSIS



Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

OUTLINE

Amortized Analysis

- Dynamic tables
- Aggregate method
- Accounting method
- Potential method



How large should a hash table be?

Goal: Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

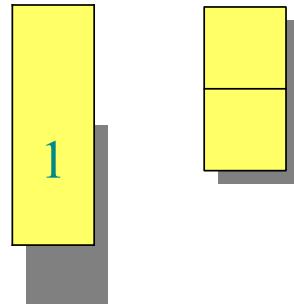
Problem: What if we don't know the proper size in advance?

Solution: *Dynamic tables.*

IDEA: Whenever the table overflows, “grow” it by allocating (via `malloc` or `new`) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

Example of a dynamic table

1. INSERT
2. INSERT

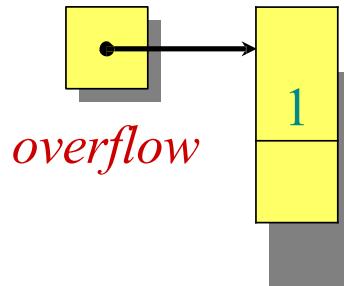


overflow



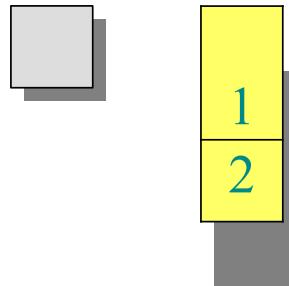
Example of a dynamic table

1. INSERT
2. INSERT



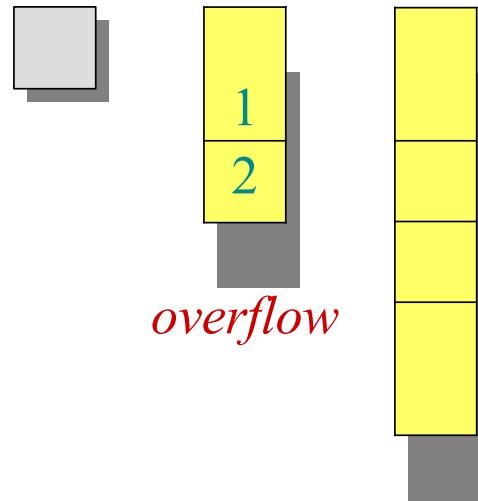
Example of a dynamic table

1. INSERT
2. INSERT



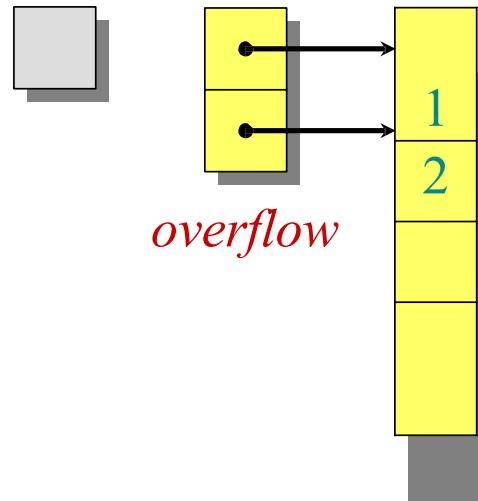
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



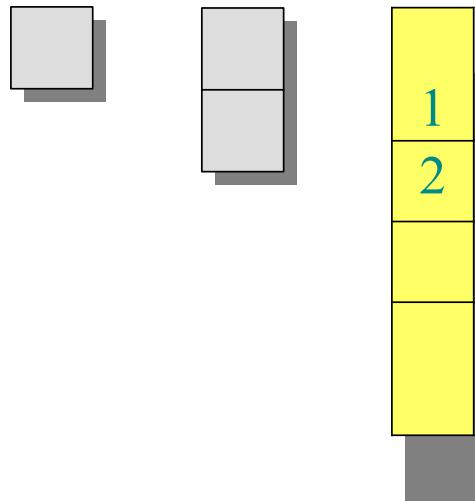
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



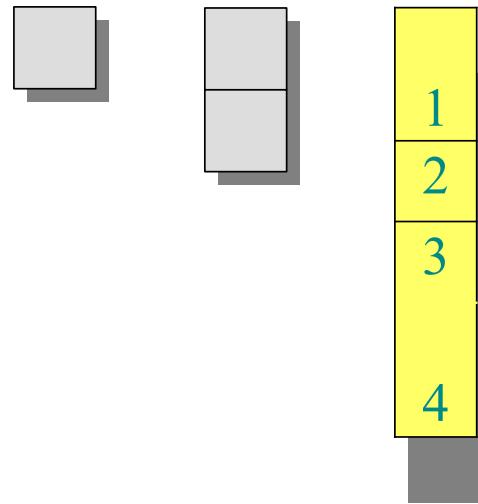
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



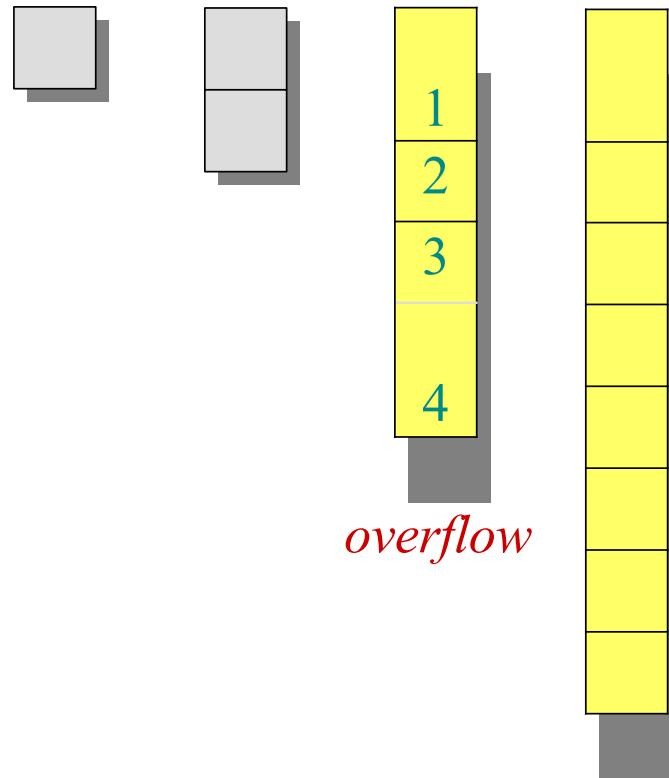
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT



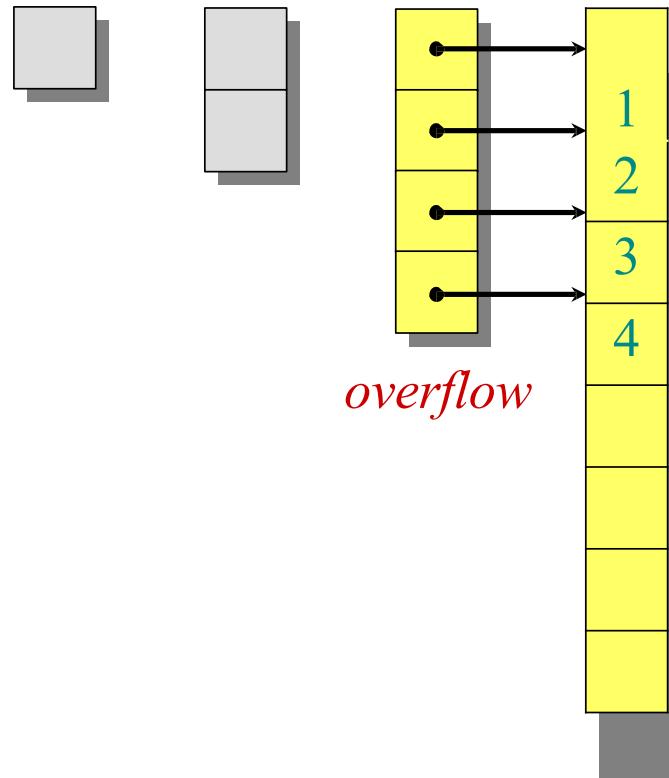
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



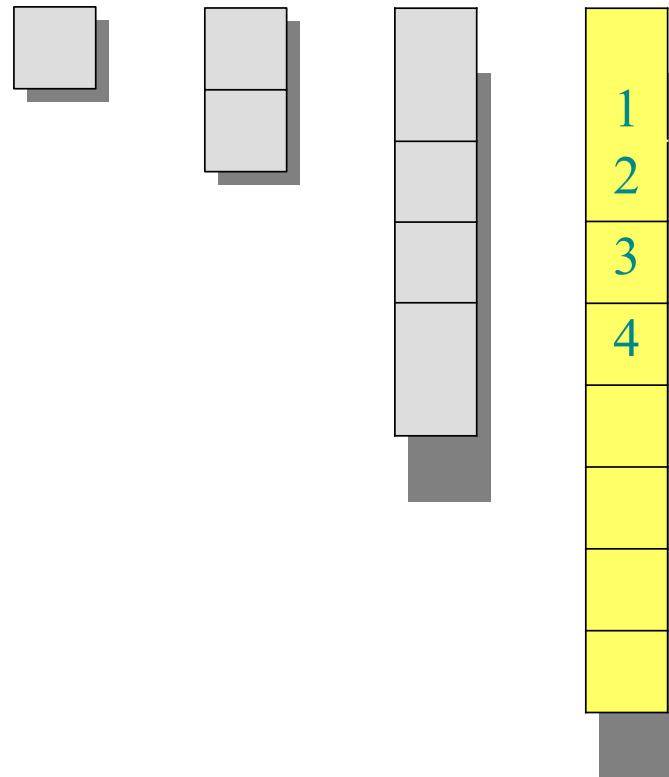
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



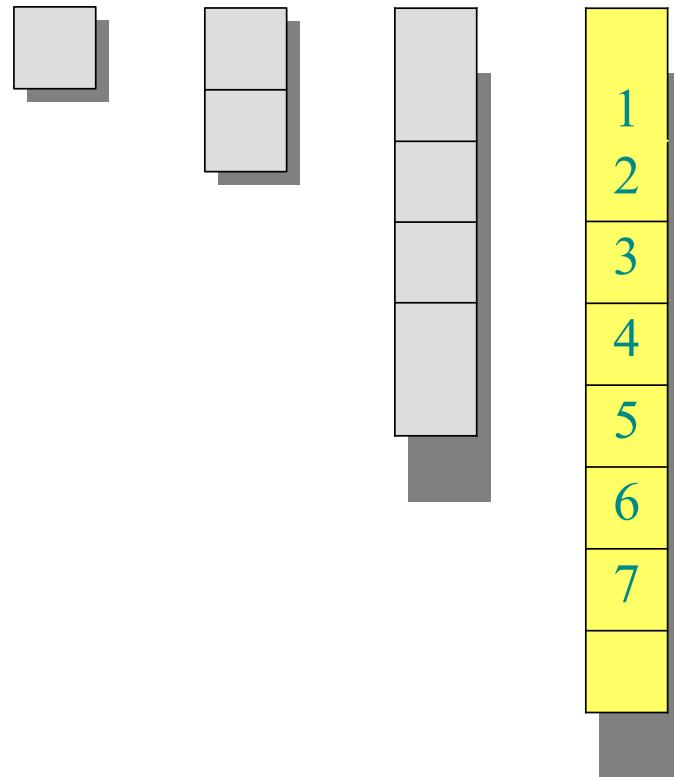
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

WRONG! In fact, the worst-case cost for n insertions is only $\Theta(n) \ll \Theta(n^2)$.

Let's see why.



Tighter analysis

Let $c_i =$ the cost of the i th insertion

$$= \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1



Tighter analysis

Let $c_i =$ the cost of the i th insertion

$$= \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
	1	2		4				8		

Tighter analysis (continued)



$$\begin{aligned}\text{Cost of } n \text{ insertions} &= \sum_{i=1}^n c_i \\ &\leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\ &\leq 3n \\ &= \Theta(n).\end{aligned}$$

Thus, the average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.



Amortized analysis

An *amortized analysis* is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Even though we're taking averages, however, probability is not involved!

- An amortized analysis guarantees the average performance of each operation in the *worst case*.



Types of amortized analyses

Three common amortization arguments:

- the *aggregate* method,
- the *accounting* method,
- the *potential* method.

We've just seen an aggregate analysis.

The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific *amortized cost* to be allocated to each operation.

Accounting method

- Charge i th operation a fictitious *amortized cost* \hat{c}_i , where \$1 pays for 1 unit of work (*i.e.*, time).
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the **bank** for use by subsequent operations.
- The bank balance must not go negative! We must ensure that

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i$$

for all n .

- Thus, the total amortized costs provide an upper bound on the total true costs.

Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

- \$1 pays for the immediate insertion.
 - \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:

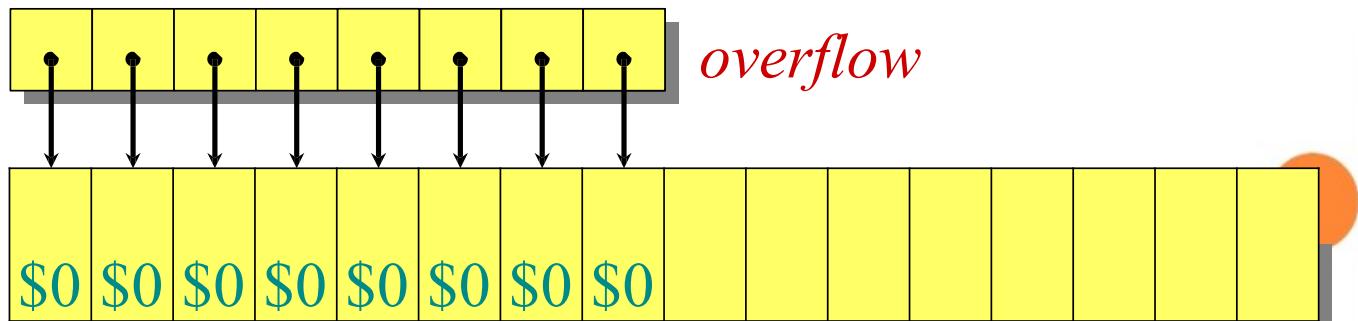
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

- $\$1$ pays for the immediate insertion.
- $\$2$ is stored for later table doubling.

When the table doubles, $\$1$ pays to move a recent item, and $\$1$ pays to move an old item.

Example:



Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

- \$1 pays for the immediate insertion.
 - \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:

Accounting analysis (continued)

Key invariant: Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
\hat{c}_i	2*	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

Potential method

IDEA: View the bank account as the potential energy (*à la* physics) of the dynamic set.

Framework:

- Start with an initial data structure D_0 .
- Operation i transforms D_{i-1} to D_i .
- The cost of operation i is c_i .
- Define a ***potential function*** $\Phi : \{D_i\} \rightarrow \mathbb{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i .
- The ***amortized cost*** \hat{c}_i with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.



Understanding potentials

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textcolor{red}{potential difference } \Delta\Phi_i}$$

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i .



The amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Summing both sides.



The amortized costs bound the true costs

The total amortized cost of n operations is

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

The series telescopes.



The amortized costs bound the true costs

The total amortized cost of n operations is

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) \\ &\geq \sum_{i=1}^n c_i \quad \text{since } \Phi(D_n) \geq 0 \text{ and } \Phi(D_0) = 0.\end{aligned}$$

Potential analysis of table doubling

Define the potential of the table after the i th insertion by $\Phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$. (Assume that $2^{\lceil \lg 0 \rceil} = 0$.)

Note:

- $\Phi(D_0) = 0$,
- $\Phi(D_i) \geq 0$ for all i .

Example:

•	•	•	•	•	•		
\$0	\$0	\$0	\$0	\$2	\$2		

$$\Phi = 2 \cdot 6 - 2^3 = 4$$

accounting method)

Calculation of amortized costs

The amortized cost of the i th insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$



Calculation of amortized costs

The amortized cost of the i th insertion is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \left\{ \begin{array}{ll} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise;} \end{array} \right\} \\ &\quad + (2i - 2^{\lceil \lg i \rceil}) - (2(i-1) - 2^{\lceil \lg (i-1) \rceil})\end{aligned}$$



Calculation of amortized costs

The amortized cost of the i th insertion is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\&= \left\{ \begin{array}{l} i \text{ if } i-1 \text{ is an exact power of 2,} \\ 1 \text{ otherwise;} \end{array} \right\} \\&\quad + (2i - 2^{\lceil \lg i \rceil}) - (2(i-1) - 2^{\lceil \lg (i-1) \rceil}) \\&= \left\{ \begin{array}{l} i \text{ if } i-1 \text{ is an exact power of 2,} \\ 1 \text{ otherwise;} \end{array} \right\} \\&\quad + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}.\end{aligned}$$



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\hat{c}_i = i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil}$$



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1)\end{aligned}$$



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1\end{aligned}$$



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\hat{c}_i = 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil}$$



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= 3 \quad (\text{since } 2^{\lceil \lg i \rceil} = 2^{\lceil \lg(i-1) \rceil})\end{aligned}$$



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= 3\end{aligned}$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil} \\&= 3\end{aligned}$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.

Exercise: Fix the bug in this analysis to show that the amortized cost of the first insertion is only 2.

Conclusions

- Amortized costs can provide a clean abstraction of data-structure performance.
- Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest or most precise.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.



可能考的题 (三种方法求解 T(n))

例：

0	0	0	0	$\rightarrow c=1$
0	0	0	1	$\rightarrow c=2$
0	0	1	0	$\rightarrow c=1$
0	0	1	1	$\rightarrow c=3$
0	1	0	0	$\rightarrow c=1$
0	1	0	1	$\rightarrow c=2$
0	1	1	0	$\rightarrow c=1$
1	0	0	0	$\rightarrow c=4$

必修
.

DYNAMIC PROGRAMMING

Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen



OUTLINE

- Introduction to Dynamic Programming
- Famous Examples
 - Matrix-chain Multiplication
 - Longest Common Subsequence
 - Triangle Decomposition of Convex Polygon
 - The Optimal Binary Search Trees



INTRODUCTION TO DYNAMIC PROGRAMMING

Why?
What?
How?



Why?

The problem of Divide-and-conquer

- Treat the subproblems independently;
- If they are dependent, we will calculate redundantly, which leads low efficiency.

Optimization Problem

- Given a group of constraints and the cost function, find **a** solution with ***the*** optimal value (min or max) in the solution space;
- Lots of the optimization can be divided into subproblems, which are dependent, so the solution of the subproblem can be reused.



Why?

Those who cannot
remember the past are
doomed to repeat it.

-----George Santayana,
The life of Reason,
Book I: Introduction and
Reason in Common
Sense (1905)



WHAT

Dynamic Programming

- Divide into **subproblem**
- Solve each subproblem only once, store the solutions in a **list**, and access it when the solution is reused
- **Bottom-up**



WHAT

Elements of dynamic programming

- Optimal substructure

A problem *exhibits Optimal substructure* if an optimal solution to the problem is contained within its optimal solutions to subproblems.

- Overlapping subproblems

When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has overlapping subproblems.



OPTIMAL SUBSTRUCTURE

- 1) Show that a solution to the problem consists of making a **choice**.
- 2) Suppose the choices are **known**.
- 3) Determine which **subproblems** ensue.
- 4) **Cut-and-paste**.



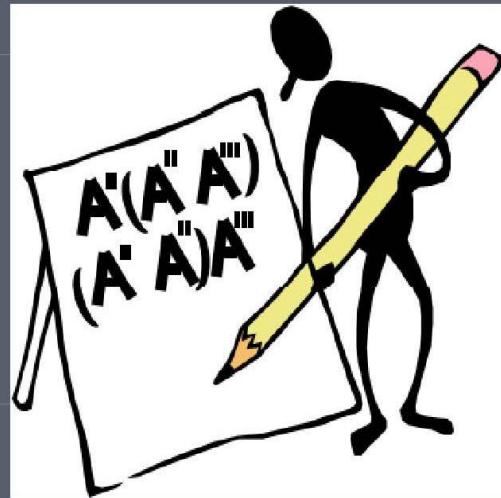
How?

The step of dynamic programming:

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed information



MATRIX-CHAIN MULTIPLICATION



PROBLEM DEFINITION

Inputs: Matrix chain $\langle A_1, A_2, \dots, A_n \rangle$

Outputs: $A_1 A_2 \dots A_n$

If A is a $p \times q$ matrix, and B a $q \times r$ matrix, then normally we spend $O(pqr)$ times to compute AB .

MOTIVATION

Matrix multiplication fulfill multiplication **associativity**.

Example:

$$\begin{aligned}& (A_1 A_2 A_3 A_4) \\& = (A_1 (A_2 (A_3 A_4))) \\& = ((A_1 A_2) (A_3 A_4)) \\& \dots \\& = (((A_1 A_2) A_3) A_4)\end{aligned}$$



MULTIPLICATION ORDER

The complexity depends on the order

- Suppose A_1 a 10×100 matrix, A_2 a 100×5 matrix, A_3 a 5×50 matrix
- $T((A_1 A_2) A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $T(A_1 (A_2 A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 750000$



SOLUTION SPACE

- Denote the number of different solutions as $P(n)$, the recursive function of $P(n)$

$$P(n) = \begin{cases} 1 & n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n>1 \end{cases}$$

- $P(n)$ is *Catalan number*

$$P(n) = C(n-1) = \frac{1}{n} \binom{2n-2}{n-1} = \Omega(4^n / n^{3/2})$$

$P(n)$ is so big to solve the problem through enumeration methods.



SOLUTION SPACE

- Denote the number of different solutions as $P(n)$, the recursive function of $P(n)$

$$P(n) = \begin{cases} 1 & n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n>1 \end{cases}$$

- $P(n)$ is *Catalan number*

$$P(n) = C(n-1) = \frac{1}{n} \binom{2n-2}{n-1} = \Omega(4^n / n^{3/2})$$

$P(n)$ is so big to solve the problem through enumeration methods.

Enumerating
won't work

STEP 1: STRUCTURE OF OPTIMAL SOLUTION

- Denote the multiplication $A_i A_{i+1} \dots A_j$ as $A [i:j]$
- Suppose we cut at k , and get a optimal order

$$(A_1 A_2 \dots A_n) = ((A_1 \dots A_k)(A_{k+1} \dots A_n))$$

- $A [1:n] = A [1:k] A [k+1:n]$
- we can prove $A [1:k]$ and $A [k+1:n]$ are optimal order too



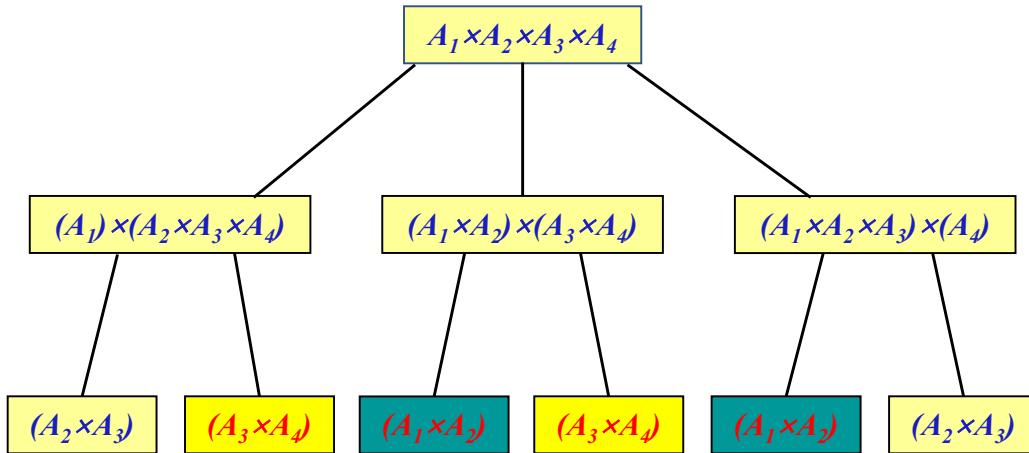
MULTIPLICATION ORDER

The complexity depends on the order

- Suppose A_1 a 10×100 matrix, A_2 a 100×5 matrix, A_3 a 5×50 matrix
- $T((A_1 A_2) A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $T(A_1 (A_2 A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 750000$



OVERLAPPING



STEP 2: RECURSION

Denotation

$m[i, j]$ ----the minimal cost(times) to calculate $A[i:j]$

$m[1, n]$ ----the minimal cost(times) to calculate $A[1:n]$

Recursion of cost

$$\begin{cases} m[i, j] = 0 & \text{if } i=j \\ m[i, j] = \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$



EXAMPLE

$$m[1,5] = \min_{1 \leq k < 5} \{ m[1, k] + m[k + 1, 5] + p_0 p_k p_5 \}$$

$m[1,1]$	$m[1,2]$	$m[1,3]$	$m[1,4]$	$m[1,5]$
	$m[2,2]$	$m[2,3]$	$m[2,4]$	$m[2,5]$
	$m[3,3]$		$m[3,4]$	$m[3,5]$
	$m[4,4]$			$m[4,5]$

$$m[2, 4] = \min \begin{cases} n[2, 2] + m[3, 4] + p_1 p_2 p_4 \\ n[2,3] + m[4, 4] + p_1 p_3 p_4 \end{cases} \quad m[5,5]$$

STEP 3 BOTTOM-UP

```
public static void matrixChain(int [] p, int [][] m, int [][] s)
{
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
        for (int i = 1; i <= n - r+1; i++) {
            int j=i+r-1;
            m[i][j] = m[i+1][j]+p[i-1]*p[i]*p[j];
            s[i][j] = i;
            for (int k = i+1; k < j; k++) {
                int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (t < m[i][j]) {
                    m[i][j] = t;
                    s[i][j] = k;
                }
            }
        }
}
```



STEP 3 BOTTOM-UP

```
public static void matrixChain(int [] p, int [][] m, int [][] s)
{
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++) {
        for (int i = 1; i <= n - r+1; i++) {
            int j=i+r-1;
            m[i][j] = m[i+1][j]+p[i-1]*p[i]*p[j];
            s[i][j] = i;
            for (int k = i+1; k < j; k++) {
                int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (t < m[i][j]) {
                    m[i][j] = t;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

Complexity:

It is based on the ternary recurrence of r, i, k . So the time complexity is $O(n^3)$ while the space complexity is $O(n^2)$.

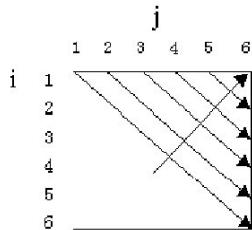
STEP 4 CONSTRUCTING

```
Print-Optimal-Parens( $s, i, j$ )
IF  $j=i$ 
THEN Print “ $A$ ”; $i$ ;
ELSE Print “(”
    Print-Optimal-Parens( $s, i, s[i, j]$ )
    Print-Optimal-Parens( $s, s[i, j]+1, j$ )
    Print “)”
```



EXAMPLE

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25



(a) 计算次序

i	1	2	3	4	5	6
j	0	15750	7875	9375	11875	15125
i	1	0	2625	4375	7125	10500
	2		0	750	2500	5375
	3			0	1000	3500
	4				0	5000
	5					0
	6					0

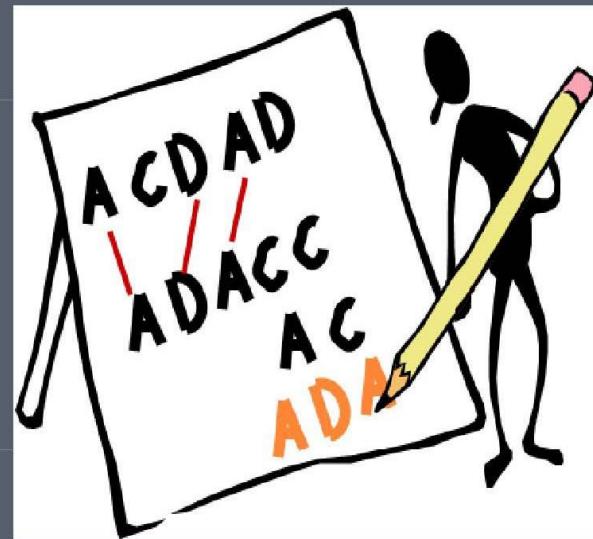
(b) m[i][j]

i	1	2	3	4	5	6
j	0	1	1	0	3	3
i	1	0	1	1	0	3
	2		0	2	0	3
	3			0	3	3
	4				0	5
	5					0
	6					0

(c) s[i][j]

$$m[2][5] = \min \begin{cases} m[2][2] + m[3][5] + p_1 p_2 p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13000 \\ m[2][3] + m[4][5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7125 \\ m[2][4] + m[5][5] + p_1 p_4 p_5 = 4375 + 0 + 35 \times 10 \times 20 = 11375 \end{cases}$$

LONGEST COMMON SEQUENCE



LONGEST COMMON SUBSEQUENCE (LCS)

- Definition
- Characterizing LCS
- Recursive solution
- Bottom-up to computing the length of LCS
- Construct Optimal solution



DEFINITION

Subsequence

Definition:

A sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a subsequence of $X = \langle x_1, x_2, \dots, x_m \rangle$ if there exists a strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j=1,2,\dots,k$, we have $x_{i_j} = z_j$

Example:

$Z = \langle A, B, C, D \rangle$ is a subsequence of $X = \langle A, C, B, C, A, D \rangle$ with corresponding index sequence $\langle 1, 3, 4, 6 \rangle$.

Common subsequence

We say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y

The longest-common-subsequence problem(LCS):

Input: $X = \langle x_1, x_2, \dots, x_m \rangle$ $Y = \langle y_1, y_2, \dots, y_n \rangle$

Output: the common subsequence Z that $\max(|Z|)$



EXAMPLE

s p r i n g t i m e
p i o n e e r
m a e l s t r o m
b e c a l m

h o r s e b a c k
s n o w f l a k e
h e r o i c a l l y
s c h o l a r l y

EXAMPLE

Subsequence

Definition:

A sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a subsequence of $X = \langle x_1, x_2, \dots, x_m \rangle$ if there exists a strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j=1,2,\dots,k$, we have $x_{i_j} = z_j$

Example:

$Z = \langle A, B, C, D \rangle$ is a subsequence of $X = \langle A, C, B, C, A, D \rangle$ with corresponding index sequence $\langle 1, 3, 4, 6 \rangle$.

Common subsequence

We say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y .

The longest-common-subsequence problem(LCS):

Input: $X = \langle x_1, x_2, \dots, x_m \rangle$ $Y = \langle y_1, y_2, \dots, y_n \rangle$

Output: the common subsequence Z that $\max(|Z|)$

Brute-force algorithm:

For every subsequence of X , check whether it's a subsequence of Y .

Time: $\Theta(n2^m)$

CHARACTERIZING LCS

The *i*th prefix of X ----- X_i

Given $X = \langle x_1, x_2, \dots, x_m \rangle$, the *i*th prefix of X is

$$X_i = \langle x_1, x_2, \dots, x_i \rangle$$

Example

$$X = (A, B, D, C, A)$$

$$X_1 = (A), X_2 = (A, B), X_3 = (A, B, D)$$

CHARACTERIZING LCS

Optimal substructure of an LCS

Let $X = \langle x_1, x_2, \dots, x_m \rangle$, $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

- { If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

RECURSIVE SOLUTION

- Define $c[i, j]$ as the length of an LCS of the sequences X_i and Y_j
- The recursive formula is

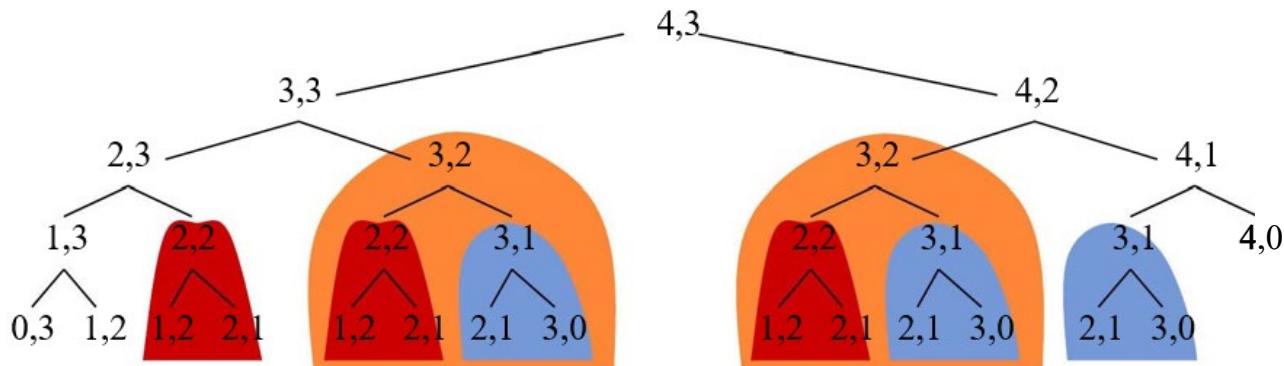
$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \text{Max}(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$



RECURSIVE SOLUTION

We could write a recursive algorithm based on this formulation.

Trv with “bozo”. “bat”.

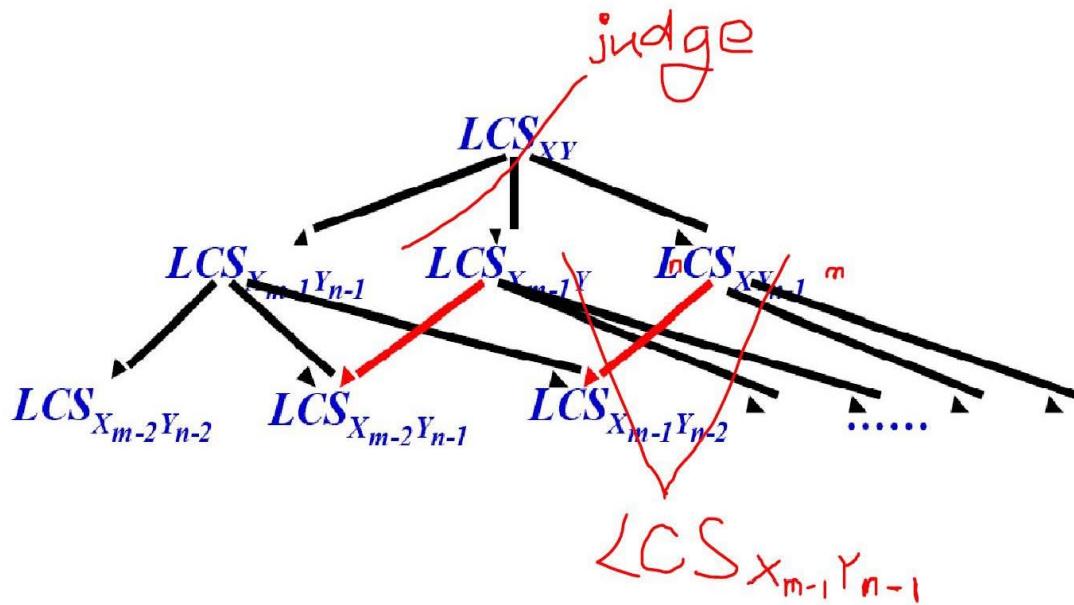


Lots of repeated subproblems.

Instead of recomputing, store in a table.



OVERLAPPING



COMPUTING THE LENGTH OF AN LCS

LCS-length(X, Y)

```
m←length( $X$ ); n←length( $Y$ );  
For  $i\leftarrow 1$  To  $m$  Do  $C[i,0]\leftarrow 0$ ;  
For  $j\leftarrow 1$  To  $n$  Do  $C[0,j]\leftarrow 0$ ;  
For  $i\leftarrow 1$  To  $m$  Do  
    For  $j\leftarrow 1$  To  $n$  Do  
        If  $X_i = Y_j$   
            Then  $C[i,j]\leftarrow C[i-1,j-1]+1$ ;  $B[i,j]\leftarrow “↖”$ ;  
        Else If  $C[i-1,j] \geq C[i,j-1]$   
            Then  $C[i,j]\leftarrow C[i-1,j]$ ;  $B[i,j]\leftarrow “↑”$ ;  
            Else  $C[i,j]\leftarrow C[i,j-1]$ ;  $B[i,j]\leftarrow “←”$ ;  
Return  $C$  and  $B$ .
```



CONSTRUCTING AN LCS

- Initial call is PRINT-LCS (B, X, m, n)
- $B[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j .
- When $b[i, j] = \nwarrow$, we have extended LCS by one character. So longest common subsequence = entries With \nwarrow in them.

Print-LCS(B, X, i, j)

IF $i=0$ or $j=0$ THEN Return;

IF $B[i, j]=“\nwarrow”$

THEN Print-LCS($B, X, i-1, j-1$); Print x_i ;

ELSE If $B[i, j]=“\uparrow”$

THEN Print-LCS($B, X, i-1, j$);

ELSE Print-LCS($B, X, i, j-1$).

EXAMPLE

$X = \langle A, B, C, B, D, A, B \rangle$ $Y = \langle B, D, C, A, B, A \rangle$

	A	B	C	B	D	A	B
B	0	0	0	0	0	0	0
D	0	0	1	1	1	2	2
C	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
B	0	1	2	2	3	3	4
A	0	1	2	2	3	3	4

Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0									
2	p	0									
3	a	0									
4	n	0									
5	k	0									
6	i	0									
7	n	0									
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0									
3	a	0									
4	n	0									
5	k	0									
6	i	0									
7	n	0									
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←
3	a	0									
4	n	0									
5	k	0									
6	i	0									
7	n	0									
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2←	2←	2←
4	n	0									
5	k	0									
6	i	0									
7	n	0									
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2←	2←	2←
4	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↖
5	k	0									
6	i	0									
7	n	0									
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2←	2←	2←
4	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↖
5	k	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↑
6	i	0									
7	n	0									
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2←	2←	2←
4	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↖
5	k	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↑
6	i	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↖	3←
7	n	0									
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2←	2←	2←
4	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↖
5	k	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↑
6	i	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↖	3←
7	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	3↑
8	g	0									



Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2←	2←	2←
4	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↖
5	k	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↑
6	i	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↖	3↑
7	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	3↑
8	g	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	4↑

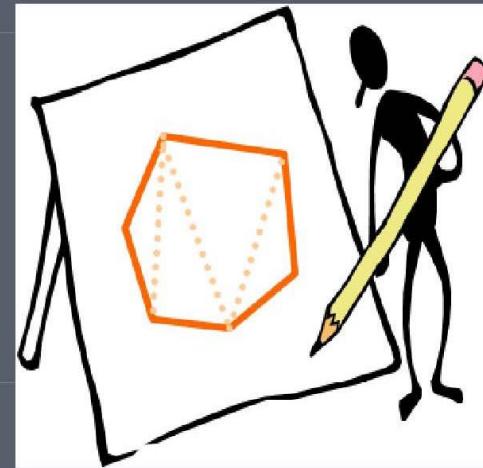


Demonstration

j	0	1	2	3	4	5	6	7	8	9	10
i	y_j	a	m	p	u	t	a	t	i	o	n
0	x_i	0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1↖	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2↖	2←	2←
4	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↖
5	k	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	3↑
6	i	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↖	3↑
7	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	4↖
8	g	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	4↖

Time: $O(mn)$

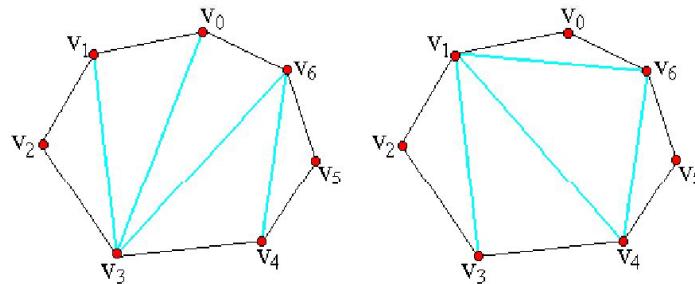
TRIANGLE DECOMPOSITION OF CONVEX POLYGON



(原理可能)

DEFINITION

- Convex polygon $P = \{v_0, v_1, \dots, v_{n-1}\}$
- Triangle decomposition: chords set T that decompose polygon into disjoint triangle
- We can improve that in a triangle decomposition of a n vertices convex polygon, there happened to be $n-3$ chords and $n-2$ triangles.



DEFINITION

Weighting function

\mathcal{W}

for example:

$$w(v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_i v_k|$$

Where $|v_i v_j|$ is the Euclid distance between
 v_i and v_j



DEFINITION

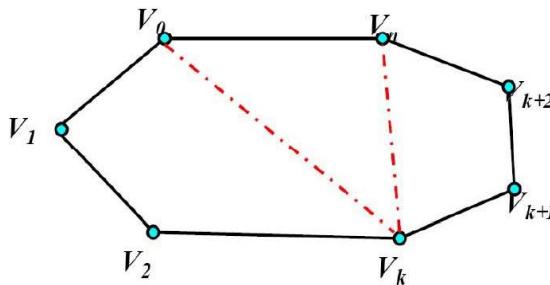
Optimal triangle decomposition

- Input: polygon P and weighting function \mathcal{W}
- Output: triangle decomposition T , to minimize

$$\sum_{S \in S_T} W(s)$$

THE STRUCTURE OF OPTIMAL SOLUTION

- $P=(v_0, v_1, \dots, v_n)$ is a $n+1$ vertices polygon
- T_p is an optimal triangle decomposition, v_k is the decomposition point.



- The structure of optimal solution

$$T_p = T(v_0, \dots, v_k) \cup T(v_k, \dots, v_n) \cup \{v_0v_k, v_kv_n\}$$

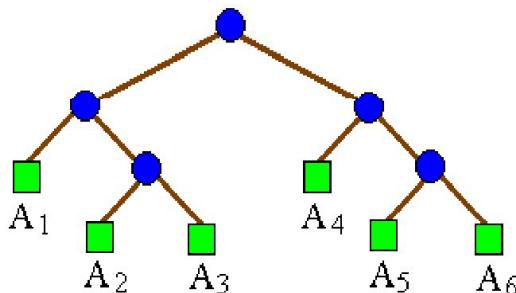


TRIANGLE DECOMPOSITION AND MATRIX-CHAIN-ORDER

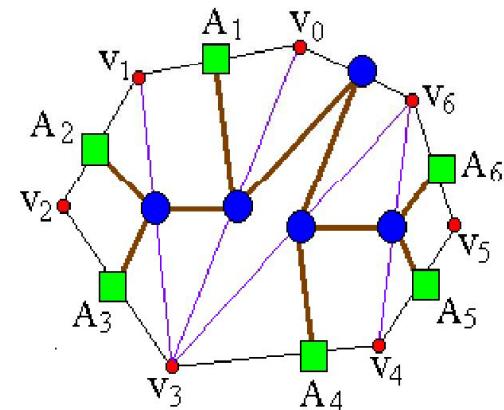
- A matrix chain is corresponding to a complete binary tree.

For example:

$((A_1(A_2A_3))(A_4(A_5A_6)))$ is converted to a complete binary tree.



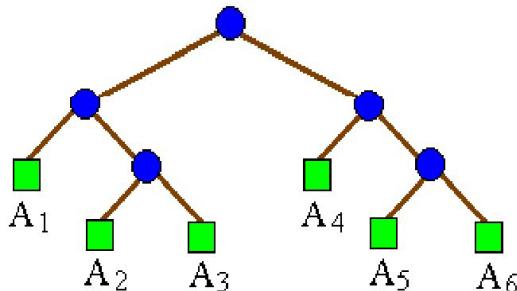
(a)



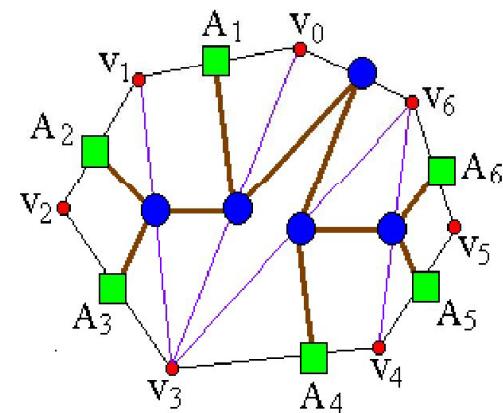
(b)

TRIANGLE DECOMPOSITION AND MATRIX-CHAIN-ORDER

Triangle decomposition of a n -vertices convex polygon is corresponding to a $n-1$ leaves complete binary tree.



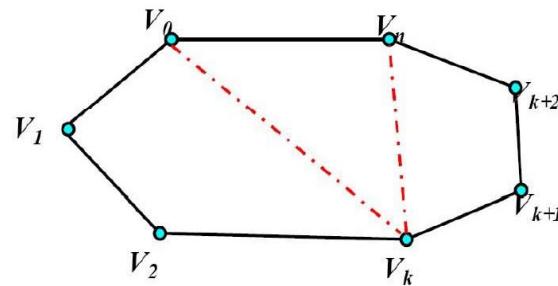
(a)



(b)

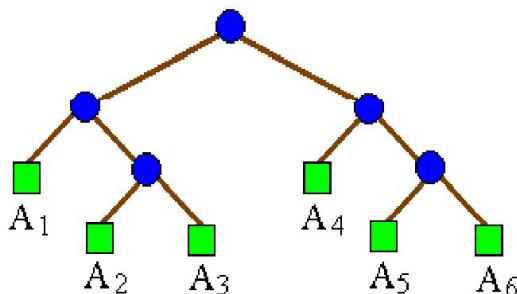
RECURSIVE SOLUTION

$$t[I][n] = \begin{cases} 0 & i = j \\ \min_{l \leq i \leq k < j \leq n} \{ t[I][k] + t[k+1][n] + w(v_0 v_k v_n) \} & i < j \end{cases}$$

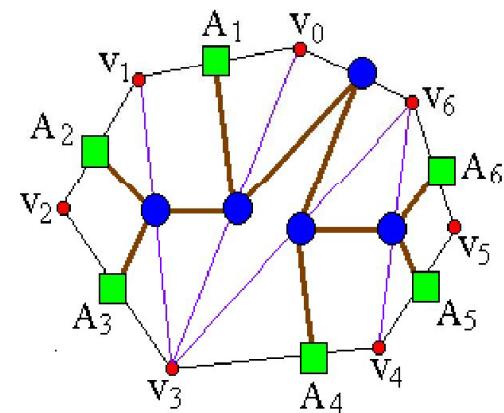


CONSTRUCT OPTIMAL SOLUTION

It is coordinate with the Matrix-chain-Order. So modify Matrix-chain-order, we can construct optimal triangle decomposition.

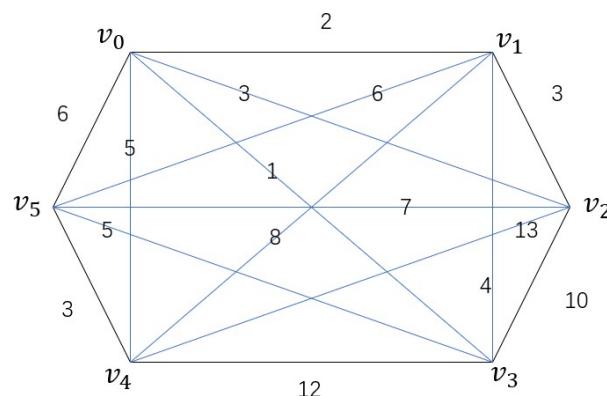


(a)



(b)

尝试可带计算器



$m[][]$	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

$g[][]$

	0	1	2	3	4	5
0	0	2	3	1	5	6
1	2	0	3	4	8	6
2	3	3	0	10	13	7
3	1	4	10	0	12	5
4	5	8	13	12	0	3
5	6	6	7	5	3	0

$s[][]$	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

三个顶点：

$$i = 1, j = 2 : \{v_0, v_1, v_2\}$$

$$k = 1 : m[1][2] = \min\{m[1][1] + m[2][2] + w(v_0 v_1 v_2)\} = 8$$

$$i = 2, j = 3 : \{v_1, v_2, v_3\}$$

$$k = 2 : m[2][3] = \min\{m[2][2] + m[3][3] + w(v_1 v_2 v_3)\} = 17$$

$$i = 3, j = 4 : \{v_2, v_3, v_4\}$$

$$k = 3 : m[3][4] = \min\{m[3][3] + m[4][4] + w(v_2 v_3 v_4)\} = 35$$

$$i = 4, j = 5 : \{v_3, v_4, v_5\}$$

$$k = 4 : m[4][5] = \min\{m[4][4] + m[5][5] + w(v_3 v_4 v_5)\} = 20$$



m[][]	1	2	3	4	5
1	0	8			
2		0	17		
3			0	35	
4				0	20
5					0

s[][]	1	2	3	4	5
1	0	1			
2		0	2		
3			0	3	
4				0	4
5					0



四个顶点：

$$i = 1, j = 3 : \{v_0, v_1, v_2, v_3\}$$

$$m[1][3] = \min \begin{cases} k = 1, m[1][1] + m[2][3] + w(v_0v_1v_3) = 24 \\ k = 2, m[1][2] + m[3][3] + w(v_0v_2v_3) = 22 \end{cases};$$

$$i = 2, j = 4 : \{v_1, v_2, v_3, v_4\}$$

$$m[2][4] = \min \begin{cases} k = 2, m[2][2] + m[3][4] + w(v_1v_2v_4) = 59 \\ k = 3, m[2][3] + m[4][4] + w(v_1v_3v_4) = 41 \end{cases};$$

$$i = 3, j = 5 : \{v_2, v_3, v_4, v_5\}$$

$$m[3][5] = \min \begin{cases} k = 3, m[3][3] + m[4][5] + w(v_2v_3v_5) = 42 \\ k = 4, m[3][4] + m[5][5] + w(v_2v_4v_5) = 58 \end{cases};$$

m[][]	1	2	3	4	5
1	0	8	22		
2		0	17	41	
3			0	35	42
4				0	20
5					0

s[][]	1	2	3	4	5
1	0	1	2		
2		0	2	3	
3			0	3	3
4				0	4
5					0



五个顶点：

$$i = 1, j = 4 : \{v_0, v_1, v_2, v_3, v_4\}$$

$$m[1][4] = \min \begin{cases} k = 1, m[1][1] + m[2][4] + w(v_0v_1v_4) = 56 \\ k = 2, m[1][2] + m[3][4] + w(v_0v_2v_4) = 64; \\ k = 3, m[1][3] + m[4][4] + w(v_0v_3v_4) = 40 \end{cases}$$

$$i = 2, j = 5 : \{v_1, v_2, v_3, v_4, v_5\}$$

$$m[2][5] = \min \begin{cases} k = 2, m[2][2] + m[3][5] + w(v_1v_2v_5) = 58 \\ k = 3, m[2][3] + m[4][5] + w(v_1v_3v_5) = 52; \\ k = 4, m[2][4] + m[5][5] + w(v_1v_4v_5) = 58 \end{cases}$$

m[][]	1	2	3	4	5
1	0	8	22	40	
2		0	17	41	52
3			0	35	42
4				0	20
5					0

s[][]	1	2	3	4	5
1	0	1	2	3	
2		0	2	3	3
3			0	3	3
4				0	4
5					0



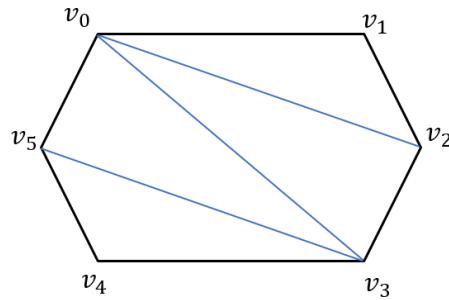
六个顶点：

$$i = 1, j = 5 : \{v_0, v_1, v_2, v_3, v_4, v_5\}$$

$$m[1][4] = \min \begin{cases} k = 1, m[1][1] + m[2][5] + w(v_0v_1v_5) = 66 \\ k = 2, m[1][2] + m[3][5] + w(v_0v_2v_5) = 66 \\ k = 3, m[1][3] + m[4][5] + w(v_0v_3v_5) = 54 \\ k = 4, m[1][4] + m[5][5] + w(v_0v_4v_5) = 54 \end{cases};$$

$m[][]$	1	2	3	4	5
1	0	8	22	40	54
2		0	17	41	52
3			0	35	42
4				0	20
5					0

$s[][]$	1	2	3	4	5
1	0	1	2	3	3
2		0	2	3	3
3			0	3	3
4				0	4
5					0



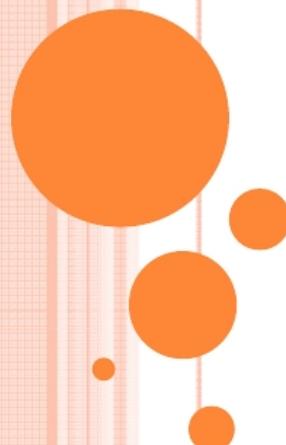
可能有 1 色.



(参考)

Greedy Algorithms

证明该贪心



Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

Greedy Algorithms

- Similar to dynamic programming. Used for optimization problems.
- Optimization problems typically go through a sequence of steps, with a set of choices at each step.
- For many optimization problems, using dynamic programming to determine the best choices is overkill (过度的杀伤威力).
- Greedy Algorithm: Simpler, more efficient

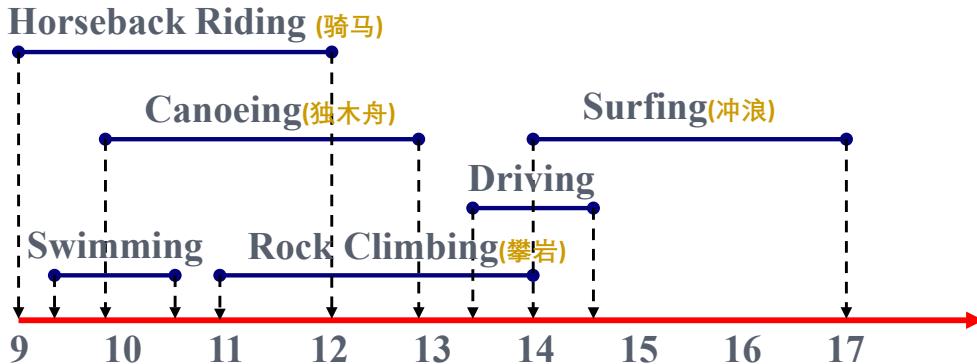


Greedy Algorithms

- 1, the activity-selection problem (活动安排)
- 2, basic elements of the GA; knapsack prob. (贪婪算法的基本特征；背包问题)
- 3, an important application: the design of data compression (Huffman) codes. (哈夫曼编码)



First example: Activity Selection



- How to make an arrangement to have the more activities?
 - ◆ S1. Shortest activity first (最短活动优先原则)
 - Swimming , Driving
 - ◆ S2. First starting activity first (最早开始活动优先原则)
 - Horseback Riding , Driving
 - ◆ S3. First finishing activity first (最早结束活动优先原则)
 - Swimming , Rock Climbing , Surfing



An activity-selection problem

- *n activities require exclusive use of a common resource.*



Example, scheduling the use of a classroom.

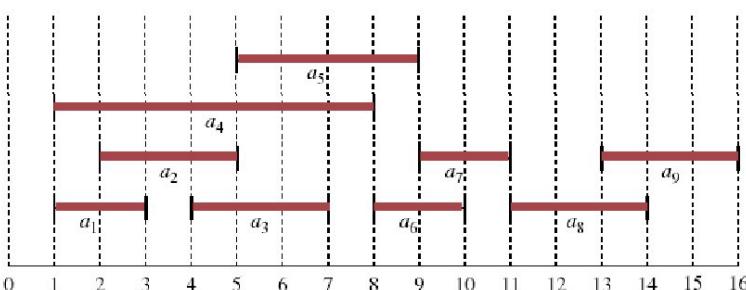
(n 个活动, 1 项资源, 任一活动进行时需唯一占用该资源)

- ◆ Set of activities $S = \{a_1, a_2, \dots, a_n\}$.
- ◆ a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i is start time and f_i is finish time.
- ◆ *Goal:* Select the largest possible set of nonoverlapping (mutually compatible) activities. (安排一个活动计划, 使得相容的活动数目最多)
- ◆ Other objectives: Maximize income rental fees , ...

An activity-selection problem

- n activities require *exclusive* use of a common resource.
 - ◆ Set of activities $S = \{a_1, a_2, \dots, a_n\}$
 - ◆ a_i needs resource during period $[s_i, f_i)$
- Example: S sorted by finish time:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Maximum-size mutually compatible set:

$$\{a_1, a_3, a_6, a_8\}.$$

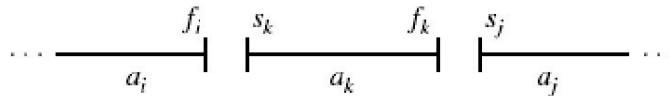
Not unique: also

$$\{a_2, a_5, a_7, a_9\}.$$

Optimal substructure of activity selection

Space of subproblems

- $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ = activities that start after a_i finishes & finish before a_j starts



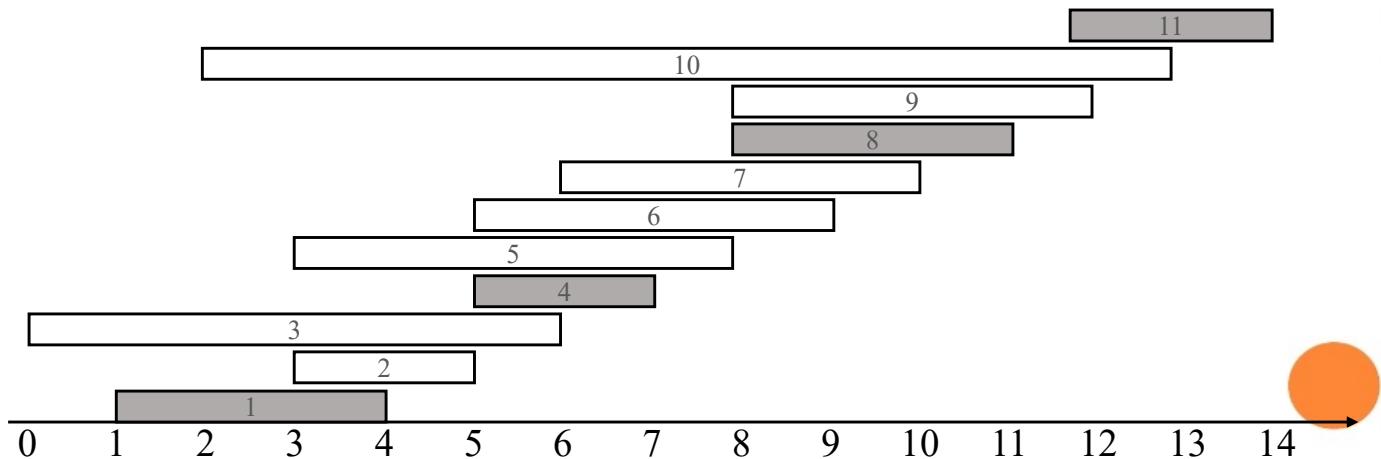
- Activities in S_{ij} are compatible with
 - ◆ all activities that finish by f_i (完成时间早于 f_i 的活动) , and
 - ◆ all activities that start no earlier than s_j .
- To represent the entire problem, add fictitious activities:
 - ◆ $a_0 = [-\infty, 0]$; $a_{n+1} = [\infty, \infty+1]$
 - ◆ We don't care about $-\infty$ in a_0 or " $\infty+1$ " in a_{n+1} .
- Then $S = S_{0,n+1}$. Range for S_{ij} is $0 \leq i, j \leq n + 1$.



Optimal substructure of activity selection

Space of subproblems

- $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$
- Assume that activities are sorted by monotonically increasing finish time (以结束时间单调增的方式对活动进行排序)
- $f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n < f_{n+1}$ (if $i \leq j$, then $f_i \leq f_j$) (16.1)



Optimal substructure of activity selection

- If $f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n < f_{n+1}$ (if $i \leq j$, then $f_i \leq f_j$) (16.1)
- Then $i \geq j \Rightarrow S_{ij} = \emptyset$

Proof If there exists $a_k \in S_{ij}$, then

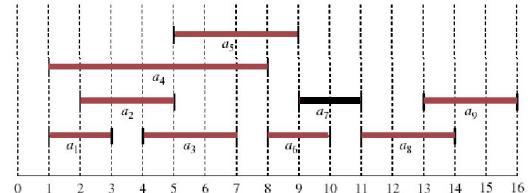
$$f_i \leq s_k < f_k \leq s_j < f_j \Rightarrow f_i < f_j.$$

But $i \geq j \Rightarrow f_i \geq f_j$. Contradiction.

- So only need to worry about S_{ij} with $0 \leq i < j \leq n + 1$.

All other S_{ij} are \emptyset

Optimal substructure of activity selection



- Suppose that a solution to S_{ij} includes a_k . Have 2 sub-prob
 - ◆ S_{ik} (start after a_i finishes, finish before a_k starts)
 - ◆ S_{kj} (start after a_k finishes, finish before a_j starts)
- Solution to $S_{ij} = (\text{solution to } S_{ik}) \cup \{a_k\} \cup (\text{solution to } S_{kj})$
Since a_k is in neither of the subproblems, and the subproblems are disjoint, $|\text{solution to } S| = |\text{solution to } S_{ik}| + 1 + |\text{solution to } S_{kj}|$.
- **Optimal substructure:** If an optimal solution to S_{ij} includes a_k , then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well. (use usual cut-and-paste argument).
- Let $A_{ij} = \text{optimal solution to } S_{ij}$,
so $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, (16.2)
assuming: S_{ij} is nonempty; and we know a_k .



A recursive solution

- Let $c[i, j] = \text{size of maximum-size subset of mutually compatible activities in } S_{ij}$. ($c[i, j]$ 表示 S_{ij} 相容的最大活动数)

$$i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i, j] = 0.$$

- If $S_{ij} \neq \emptyset$, suppose that a_k is used in a maximum-size subsets of mutually compatible activities in S_{ij} . Then $c[i, j] = c[i, k] + 1 + c[k, j]$.
- But of course we don't know which k to use, and so

$$c[i, j] = \begin{cases} 0, & \text{if } S_{ij} = \emptyset \\ \max_{\substack{i < k < j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\}, & \text{if } S_{ij} \neq \emptyset \end{cases} \quad (16.3)$$

Why this range of k ? Because $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\} \Rightarrow a_k$ can't be a_i or a_j .



Converting a DP solution to a greedy solution

$$c[i, j] = \begin{cases} 0, & \text{if } S_{ij} = \emptyset \\ \max_{\substack{i < k < j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\}, & \text{if } S_{ij} \neq \emptyset \end{cases} \quad (16.3)$$

- It may be easy to design an algorithm to the problem based on recurrence (16.3).
 - ◆ Direct recursion algorithm (complexity?)
 - ◆ Dynamic programming algorithm (complexity?)
- Can we simplify our solution?

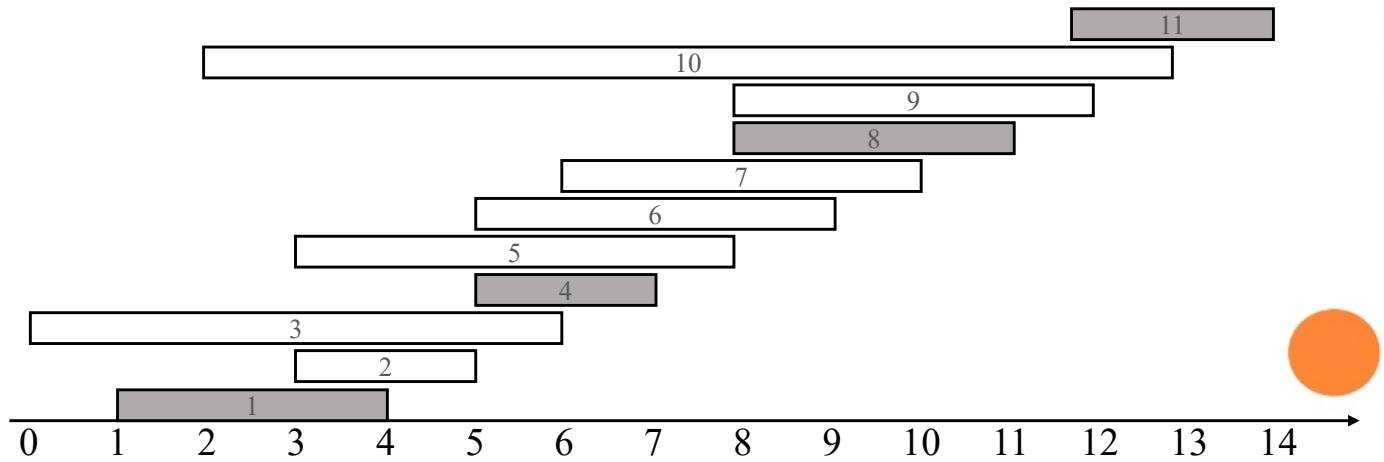


Converting a DP solution to a greedy solution

□ Theorem 16.1

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$. Then

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij} . (a_m 包含在某个最大相容活动子集中)
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. (仅剩下下一个非空子问题 s_{mj})



Converting a DP solution to a greedy solution

□ Theorem 16.1

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$. Then

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij} . (a_m 包含在某个最大相容活动子集中)
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. (仅剩下下一个非空子问题 s_{mj})

Proof

2. Suppose there is some $a_l \in S$. Then $f_i < s_l < f_l \leq s_m < f_m \Rightarrow f_l < f_m$. Then $a_l \in S_{ij}$ and it has an earlier finish time than f_m , which contradicts our choice of a_m . Therefore, there is no $a_l \in S_{im} \Rightarrow S_{im} = \emptyset$.



Converting a DP solution to a greedy solution

□ Theorem 16.1

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$. Then

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij} . (a_m 包含在某个最大相容活动子集中)

Proof

1. Let A_{ij} be a maximum-size subset of mutually Compatible activities in S_{ij} . Order activities in A_{ij} in monotonically increasing order of finish time. Let a_k be the first activity in A_{ij} .
 - ◆ If $a_k = a_m$, done (a_m is used in a maximum-size subset).
 - ◆ Otherwise, construct $B_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m). Activities in B_{ij} are disjoint. (Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish. $f_m \leq f_k \Rightarrow a_m$ doesn't overlap anything else in B_{ij}). Since $|B_{ij}| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is B_{ij} .

Converting a DP solution to a greedy solution

$$c[i, j] = \begin{cases} 0, & \text{if } S_{ij} = \emptyset \\ \max_{\substack{i < k < j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + l\}, & \text{if } S_{ij} \neq \emptyset \end{cases} \quad (16.3)$$

□ Theorem 16.1

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$. Then

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij} . (a_m 包含在某个最大相容活动子集中)
 2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. (仅剩下一个非空子问题 s_{mj})
- This theorem is great:

	before theorem	after theorem
# of sub-prob in optimal solution	2	1
# of choices to consider	$O(j - i - 1)$	1

Converting a DP solution to a greedy solution

□ Theorem 16.1

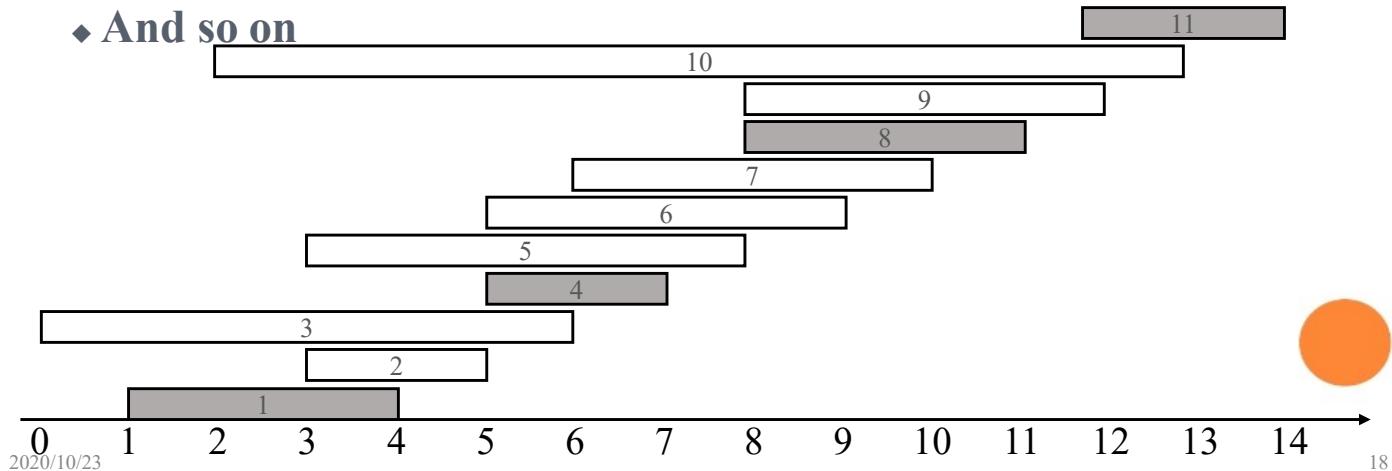
Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$. Then

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij} . (a_m 包含在某个最大相容活动子集中)
 2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem. (仅剩下一个非空子问题 s_{mj})
- Now we can solve a problem S_{ij} in a top-down fashion
 - ◆ Choose $a_m \in S_{ij}$ with earliest finish time: the greedy choice. (it leaves as much opportunity as possible for the remaining activities to be scheduled)
(留下尽可能的时间来安排活动, 贪心选择)
 - ◆ Then solve S_{mj} .



Converting a DP solution to a greedy solution

- What are the subproblems?
 - ◆ Original problem is $S_{0,n+1}$ [$a_0 = [-\infty, 0]$; $a_{n+1} = [\infty, \infty+1]$]
 - ◆ Suppose our first choice is a_{m1} (in fact, it is a_1)
 - ◆ Then next subproblem is $S_{m1,n+1}$
 - ◆ Suppose next choice is a_{m2} (it must be a_2 ?) Next subproblem is $S_{m2,n+1}$
 - ◆ And so on



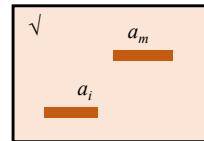
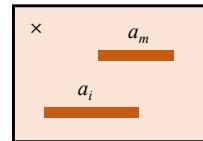
Converting a DP solution to a greedy solution

- What are the subproblems?
 - ◆ Original problem is $S_{0,n+1}$ [$a_0 = [-\infty, 0)$; $a_{n+1} = [\infty, \infty+1]$]
 - ◆ Suppose our first choice is a_{m1} (in fact, it is a_1)
 - ◆ Then next subproblem is $S_{m1,n+1}$
 - ◆ Suppose next choice is a_{m2} (it must be a_2 ?) Next subproblem is $S_{m2,n+1}$
 - ◆ And so on
- Each subproblem is $S_{mi,n+1}$.
- And the subproblems chosen have finish times that increase. (所选的子问题，其完成时间是增序排列)
- Therefore, we can consider each activity **just once**, in monotonically increasing order of finish time.



A recursive greedy algorithm

- Original problem is $S_{0,n+1}$
- Each subproblem is $S_{mi,n+1}$
- Assumes activities already sorted by monotonically increasing finish time.
(If not, then sort in $O(n \lg n)$ time.) Return an optimal solution for $S_{i,n+1}$:



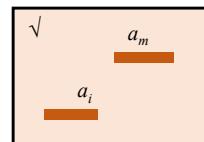
REC-ACTIVITY-SELECTOR(s, f, i, n)

```
1  $m \leftarrow i+1$ 
2 while  $m \leq n$  and  $s_m < f_i$  // Find first activity in  $S_{i,n+1}$ .
3   do  $m \leftarrow m+1$ 
4 if  $m \leq n$ 
5   then return  $\{a_m\} \cup \text{REC-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6 else return  $\emptyset$ 
```

A recursive greedy algorithm

```
REC-ACTIVITY-SELECTOR(s, f, i, n)
1  $m \leftarrow i+1$ 
2 while  $m \leq n$  and  $s_m < f_i$  // Find first activity in  $S_{i,n+1}$ .
3   do  $m \leftarrow m+1$ 
4 if  $m \leq n$ 
5   then return  $\{a_m\} \cup \text{REC-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6 else return  $\emptyset$ 
```

- *Initial call:* REC-ACTIVITY-SELECTOR(s, f, **0, n**).
- *Idea:* The **while** loop checks $a_{i+1}, a_{i+2}, \dots, a_n$ until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).
 - ◆ If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution, along with a_m .
 - ◆ If the loop never finds a compatible a_m ($m > n$), then just return empty set.



A recursive greedy algorithm

```
REC-ACTIVITY-SELECTOR(s, f, i, n)
1  $m \leftarrow i+1$ 
2 while  $m \leq n$  and  $s_m < f_i$  // Find first activity in  $S_{i,n+1}$ .
3   do  $m \leftarrow m+1$ 
4 if  $m \leq n$ 
5   then return  $\{a_m\} \cup \text{REC-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6 else return  $\emptyset$ 
```

- Time: $\Theta(n)$ —each activity examined exactly once.

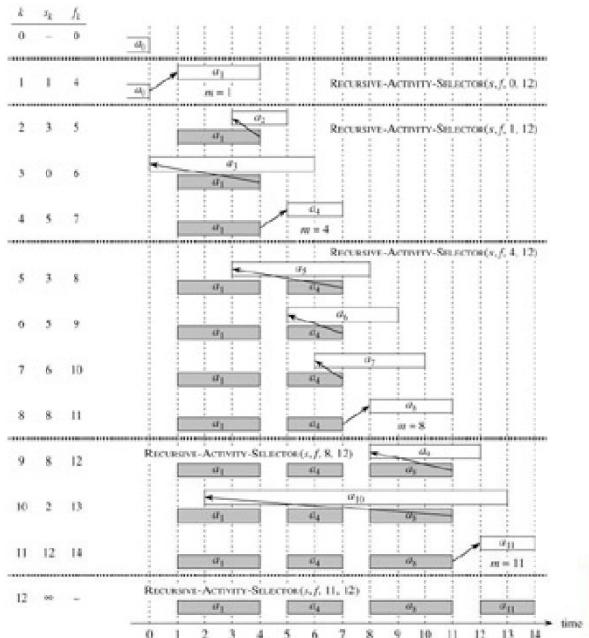
$$\begin{aligned}T(n) &= m_1 + T(n - m_1) = m_1 + m_2 + T(n - m_1 - m_2) \\&= m_1 + m_2 + m_3 + T(n - m_1 - m_2 - m_3) = \dots \\&= \sum m_k + T(n - \sum m_k)\end{aligned}$$

because: $n - \sum m_k = 1$, then $\sum m_k = n - 1$, $\sum m_k + T(1) = \Theta(n)$



A recursive greedy algorithm

- *Initial call:* REC-ACTIVITY-SELECTOR($s, f, 0, n$).
- *Idea:* The **while** loop checks $a_{i+1}, a_{i+2}, \dots, a_n$ until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).
 - ◆ If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m, n+1}$, and return this solution, along with a_m .
 - ◆ If the loop never finds a compatible a_m ($m > n$), then just return empty set.

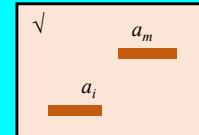
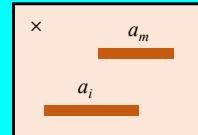


An iterative greedy algorithm

- REC-ACTIVITY-SELECTOR is almost "tail recursive".
- We easily can convert the recursive procedure to an iterative one. (Some compilers perform this task automatically)

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
```

```
1  $A \leftarrow \{a_1\}$ 
2  $i \leftarrow 1$ 
3 for  $m \leftarrow 2$  to  $n$ 
4   do if  $s_m \geq f_i$ 
5     then  $A \leftarrow A \cup \{a_m\}$ 
6      $i \leftarrow m // a_i$  is most recent addition to  $A$ 
7 return  $A$ 
```



Review

- Greedy Algorithm Idea: When we have a choice to make, make the one that looks best *right now*. Make a *locally optimal* choice in hope of getting a *globally optimal solution*. (希望当前选择是最好的，每一个局部最优选择能产生全局最优选择)
- Greedy Algorithm: Simpler, more efficient



Greedy Algorithms

- 1, the activity-selection problem (活动安排)
- 2, basic elements of the GA; knapsack prob. (贪婪算法的基本特征；背包问题)
- 3, an important application: the design of data compression (Huffman) codes. (哈夫曼编码)



Elements of the greedy strategy

- The choice that seems best at the moment is chosen. (每次决策时，当前所做的选择看起来是“最好”的)
- What did we do for activity selection?
 1. Determine the optimal substructure.
 2. Develop a recursive solution.
 3. Prove that at any stage of recursion, one of the optimal choices is the greedy choice.
 4. Show that all **but one of the subproblems resulting from the greedy choice** are empty. (通过贪婪选择，只有一个子问题非空)
 5. Develop a recursive greedy algorithm.
 6. Convert it to an iterative algorithm.



Elements of the greedy strategy

- These steps looked like dynamic programming.
- Typically, we streamline these steps (简化这些步骤)
- Develop the substructure with an eye toward
 - ◆ making the greedy choice,
 - ◆ leaving just one subproblem.
- For activity selection, we showed that the greedy choice implied that in S_{ij} , only i varied, and j was fixed at $n+1$,
- So, we could have started out with a greedy algorithm in mind:
 - ◆ define $S_i = \{a_k \in S : f_i \leq s_k\}$, (所有在 ai 结束之后开始的活动)
 - ◆ show the greedy choice, first a_m to finish in S_i
 - combined with optimal solution to S_m

⇒ optimal solution to S_i .



Elements of the greedy strategy

- Typical streamlined steps (简化这些步骤)
 1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve. (最优问题为：做一个选择，留下一个待解的子问题)
 2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe. (证明存在一个基于贪婪选择的最优解，因此贪婪选择是安全的)
 3. Show that greedy choice and optimal solution to subproblem < optimal solution to the problem. (说明由贪婪选择和子问题的最优解 < 原问题的最优解)



Elements of the greedy strategy

- No general way to tell if a greedy algorithm is optimal, but two key ingredients are (没有一般化的规则来说明贪婪算法是否最优，但有两个基本要点)
 1. **greedy-choice property** (贪婪选择属性)
 2. **optimal substructure**



Greedy-choice property

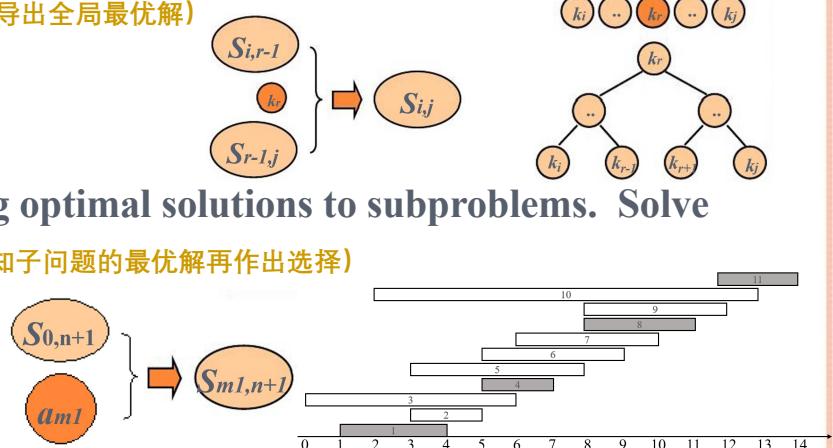
- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. (通过局部最优解可导出全局最优解)

- *Dynamic programming*

- ◆ Make a choice at each step.
 - ◆ Choice depends on knowing optimal solutions to subproblems. Solve subproblems first. (依赖于已知子问题的最优解再作出选择)
 - ◆ Solve *bottom-up*.

- *Greedy*

- ◆ Make a choice at each step.
 - ◆ Make the choice *before* solving the subproblems. (先作选择，再解子问题)
 - ◆ Solve *top-down*.



Greedy-choice property

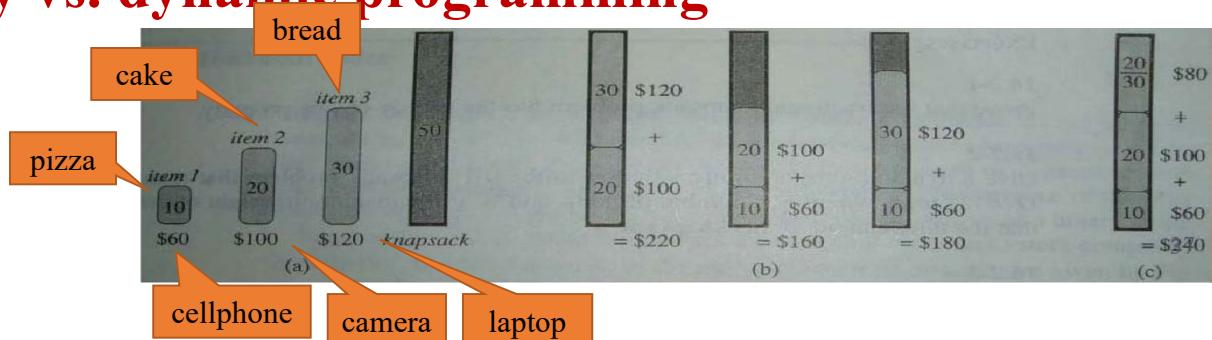
- We must **prove** that a greedy choice at each step yields a globally optimal solution. **Difficulty!** **Cleverness** may be required!
- Typically, Theorem 16.1, shows that the solution (A_{ij}) can be modified to use the greedy choice (a_m) , resulting in one similar but smaller subproblem (A_{mj}) .
- We can **get efficiency gains** from greedy-choice property. (*For example, in activity-selection, sorted the activities in monotonically increasing order of finish times, needed to examine each activity just once.*)
 - ◆ **Preprocess** input to put it into greedy order



Optimal substructure

- *optimal substructure*: an optimal solution to the problem contains within it optimal solutions to subproblems.
- Just show that **optimal solution to subproblem** and **greedy choice** \Rightarrow **optimal solution to problem**. (说明子问题的最优解和贪婪选择) \Rightarrow 原问题的最优解

Greedy vs. dynamic programming



- **0-1 knapsack problem** (0-1 背包问题, 小偷问题)

- ◆ n items (n 个物品)
- ◆ Item i is worth $\$v_i$, weighs w_i P (物品 i 价值 v_i , 重 w_i)
- ◆ Find a most valuable subset of items with total weight $\leq W$. (背包的最大负载量为 w , 如何选取物品, 使得背包装的物品价值最大)
- ◆ Have to either take an item or not take it can't take part of it. (每个物品是一个整体, 不能分割, 因此, 要么选取该物品, 要么不选取)

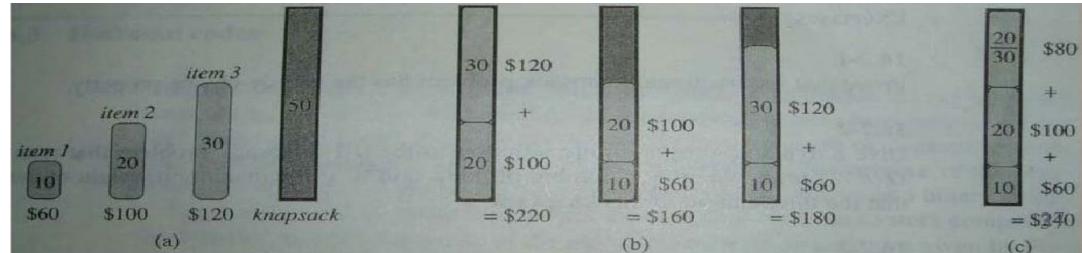
- **Fractional knapsack problem** (分数背包问题, 小偷问题)

- ◆ Like the 0-1 knapsack problem, but can take fraction of an item.

Greedy vs. dynamic programming

- *0-1 knapsack problem* (0-1 背包问题, 小偷问题)
- *Fractional knapsack problem* (分数背包问题, 小偷问题)
- Both have optimal substructure property.
 - ◆ 0-1 : choose the most valuable load j that weighs $w_j \leq W$, remove j , choose the most valuable load i that weighs $w_i \leq W-w_j$
 - ◆ fractional: choose a weight w from item j (part of j), then remove the part, the remaining load is the most valuable load weighing at most $W-w$ that the thief can take from the $n-1$ original items plus w_j-w pounds from item j .
(若先选 item j 的一部分, 其重 w , 则接下来的最优挑选方案为从余下的 $n-1$ 个物品〔除 j 外〕和 j 的另外重 w_j-w 的部分中挑选, 其重量不超过 $W-w$)
- But the fractional problem has the greedy-choice property, and the 0-1 problem does not.

Greedy vs. dynamic programming



- Fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.
- To solve the fractional problem, rank decreasingly items by v_i/w_i
- Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i
- Time: $O(n \lg n)$ to sort, $O(n)$ to greedy choice thereafter.

```
FRACTIONAL-KNAPSACK(v,w,W)
1 load  $\leftarrow 0$ 
2 i  $\leftarrow 1$ 
3 while load  $< W$  and i  $\leq n$ 
4   do if  $w_i \leq W - \text{load}$ 
5     then take all of item i
6     else take  $W - \text{load}$  of  $w_i$  from
          item i
7   add what was taken to load
8   i  $\leftarrow i + 1$ 
```

Greedy vs. dynamic programming

- 0-1 knapsack problem has not the greedy-choice property

- $W = 50$.

- Greedy solution:

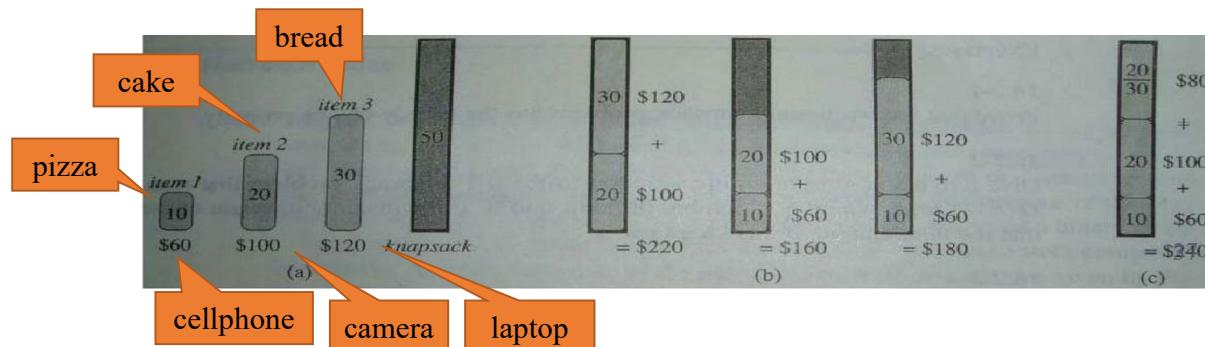
- take items 1 and 2
- value = 160, weight = 30

20 pounds of capacity leftover.

- Optimal solution:

- Take items 2 and 3
- value=220, weight=50

No leftover capacity. (没有剩余空间)



Greedy Algorithms

- 1, the activity-selection problem (活动安排)
- 2, basic elements of the GA; knapsack prob. (贪婪算法的基本特征；背包问题)
- 3, an important application: the design of data compression (Huffman) codes. (哈夫曼编码)



Huffman codes

- Huffman codes: widely used and very effective technique for compressing data.
 - ◆ savings of 20% to 90%
- Consider the data to be a sequence of characters
 - ◆ Abaaaabbbdcfffeaeeaec
- Huffman's greedy algorithm:
 - uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string. (依据字符出现的频率表，使用二进串来建立一种表示字符的最佳方法)



Huffman codes

- Wish to store compactly 100,000-character data file
only six different characters appear. frequency table

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Many ways (encodes) to represent such a file of information
- *binary character code* (or *code* for short): each character is represented by a unique binary string.
 - ◆ *fixed-length code*: if use 3-bit codeword, the file can be encoded in 300,000 bits. Can we do better?

Huffman codes

- 100,000-character data file

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- *binary character code* (or *code for short*)

◆ **variable-length code:** by giving frequent characters short codewords and infrequent characters long codewords, here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. (高频出现的字符以短字码表示；低频→长字码)

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits}$$



Huffman codes

- 100,000-character data file

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- *binary character code* (or *code for short*)
 - ◆ **fixed-length code:** 300,000 bits
 - ◆ **variable-length code:** 224,000 bits, a savings of approximately 25%. In fact, this is an optimal character code for this file.



Prefix codes

- prefix codes (prefix-free codes): no codeword is a prefix of some other codeword. (字首码, 前缀代码, 前置代码〔前缀无关码〕 : 没有字码是其他字码的前缀)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Encoding (编码) is always simple for any binary character code
 - Concatenate (连接) the codewords representing each character. For example, “abc”, with the variable-length prefix code as $0 \cdot 101 \cdot 100 = 0101100$, where we use ‘.’ to denote concatenation.
- Prefix codes simplify decoding (解码)

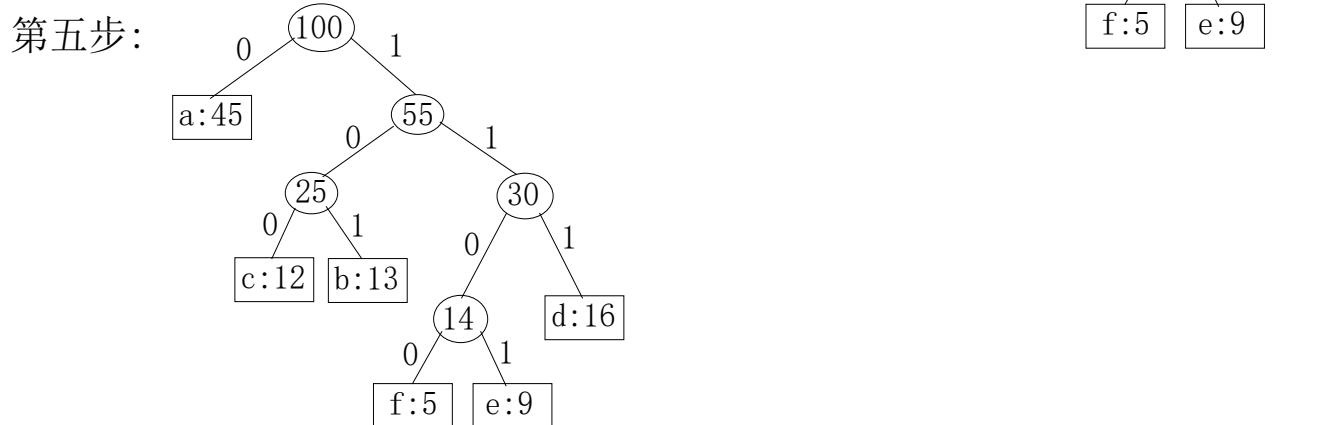
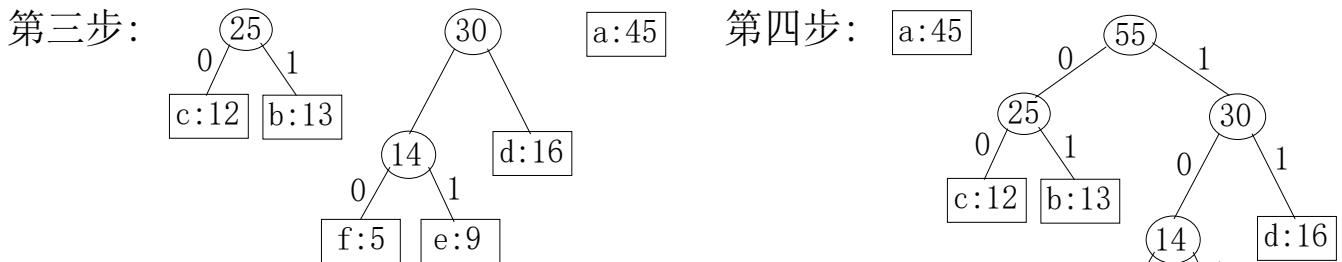
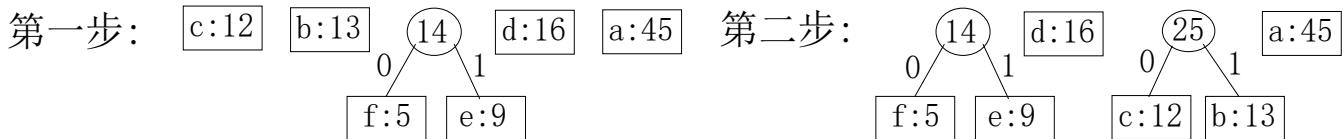
Prefix codes

- prefix codes (prefix-free codes): no codeword is a prefix of some other codeword. (字首码, 前缀代码, 前置代码〔前缀无关码〕 : 没有字码是其他字码的前缀)

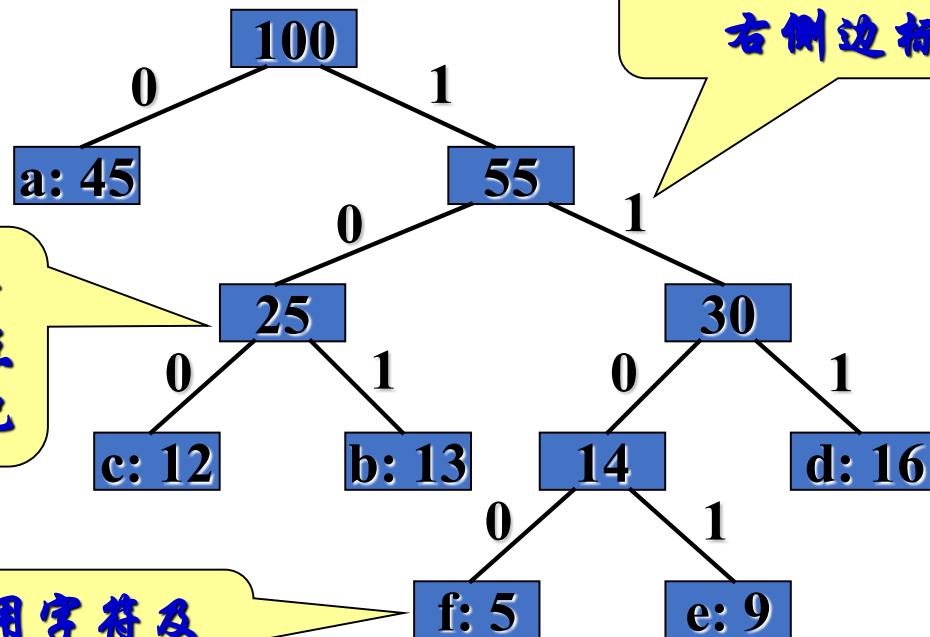
Variable-length codeword	a	b	c	d	e	f
	0	101	100	111	1101	1100

- Encoding is always simple for any binary character code
- Prefix codes **simplify decoding**
 - Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous (明确的) .
 - We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file.
 - Exam: 001011101 uniquely as 0·0·101·1101, which decodes to “aabe”.





• 编码 树



- 编码树 T 的代价
 - 设 C 是字母表, $\forall c \in C$
 - $f(c)$ 是 c 在文件中出现的频率
 - $d_T(c)$ 是叶子 c 在树 T 中的深度, 即 c 的编码长度
 - T 的代价是编码一个文件的所有字符的代码位数:

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

- 优化编码树问题

输入：字母表 $C = \{c_1, c_2, \dots, c_n\}$,

频率表 $F = \{f(c_1), f(c_2), \dots, f(c_n)\}$

输出：具有最小 $B(T)$ 的 C 前缀编码树

贪心思想：

循环地选择具有最低频率的两个结点，
生成一棵子树，直至形成树



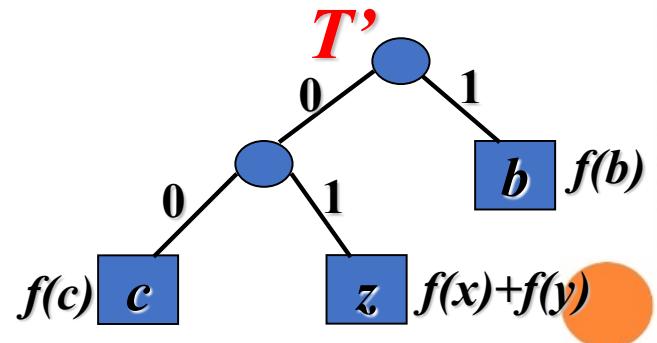
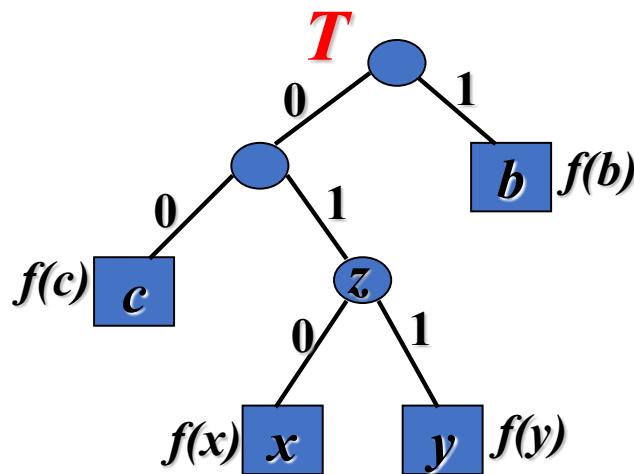
优化解的结构分析

- 我们需要证明
 - 优化前缀树问题具有优化子结构
 - 优化前缀树问题具有贪心选择性



• 优化子结构

引理1.设 T 是字母表 C 的优化前缀树， $\forall c \in C, f(c)$ 是 c 在文件中出现的频率。设 x, y 是 T 中任意两个相邻叶结点， z 是它们的父结点，则 z 作为频率是 $f(z) = f(x) + f(y)$ 的字符， $T' = T - \{x, y\}$ 是字母表 $C' = C - \{x, y\} \cup \{z\}$ 的优化前缀编码树。



证. 证 $B(T)=B(T')+f(x)+f(y)$.

$$\forall v \in C - \{x, y\}, d_T(v) = d_{T'}(v), f(v)d_T(v) = f(v)d_{T'}(v).$$

由于 $d_T(x) = d_T(y) = d_{T'}(z) + 1$,

$$\begin{aligned} & f(x)d_T(x) + f(y)d_T(y) \\ &= (f(x) + f(y))(d_{T'}(z) + 1) \\ &= (f(x) + f(y))d_{T'}(z) + (f(x) + f(y)) \end{aligned}$$

由于 $f(x) + f(y) = f(z)$, $f(x)d_T(x) + f(y)d_T(y) = f(z)d_{T'}(z) + (f(x) + f(y))$.

于是 $B(T) = B(T') + f(x) + f(y)$.

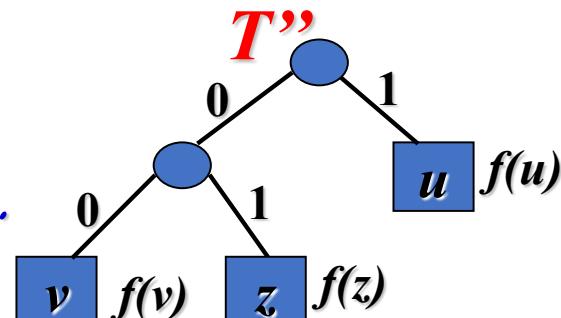
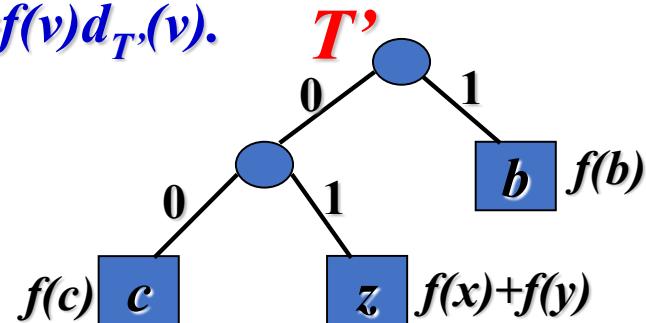
若 T' 不是 C' 的优化前缀编码树,

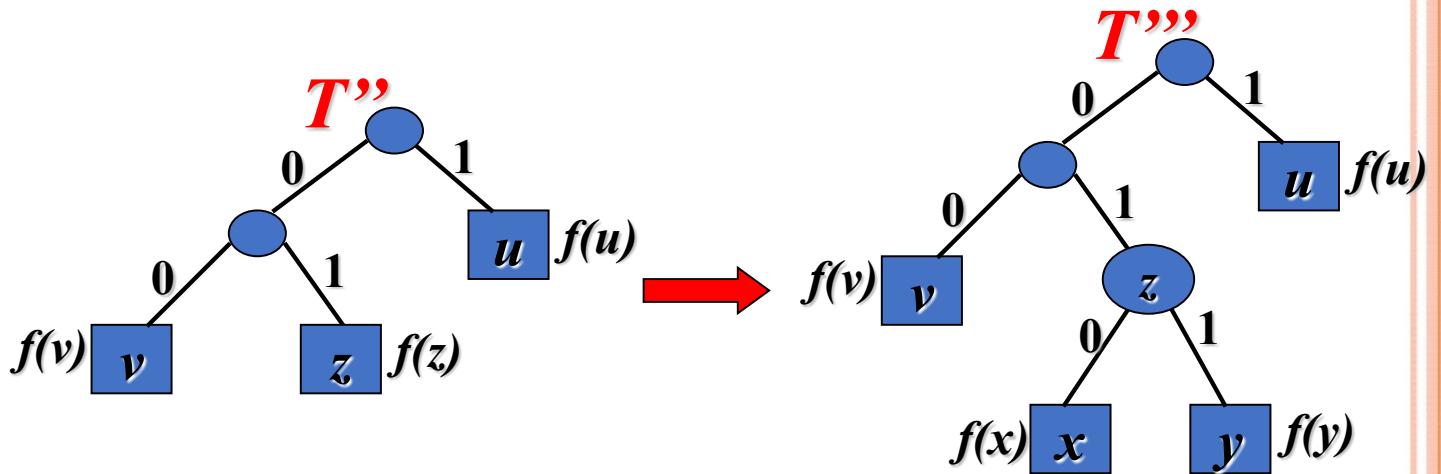
则必存在 T'' , 使 $B(T'') < B(T')$.

因为 z 是 C' 中字符, 它必为 T'' 中的叶子.

把结点 x 与 y 加入 T'' , 作为 z 的子结点,

则得到 C 的一个如下前缀编码树 T''' :





T''' 代价为：

$$\begin{aligned}
 B(T''') &= \dots + (f(x) + f(y))(d_{T''}(z) + 1) \\
 &= \dots + f(z)d_{T''}(z) + (f(x) + f(y)) \quad (d_{T''}(z) = d_{T'}(z)) \\
 &= B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = B(T)
 \end{aligned}$$

与 T 是优化的矛盾，故 T' 是 C' 的优化编码树。

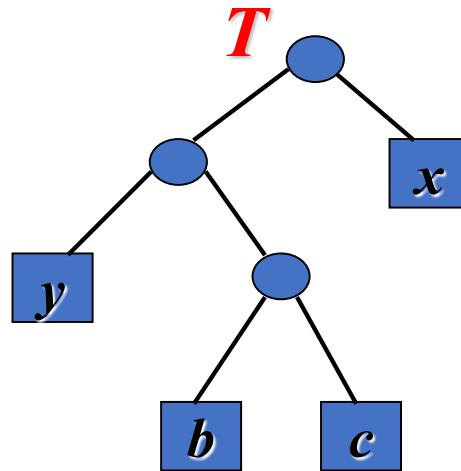


- 贪心选择性

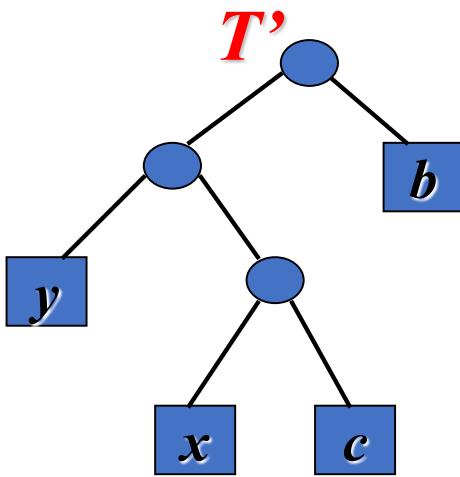
引理2. 设 C 是字母表， $\forall c \in C$ ， c 具有频率 $f(c)$ ， x 、 y 是 C 中具有最小频率的两个字符，则存在一个 C 的优化前缀树， x 与 y 的编码具有相同长度，且仅在最末一位不同。



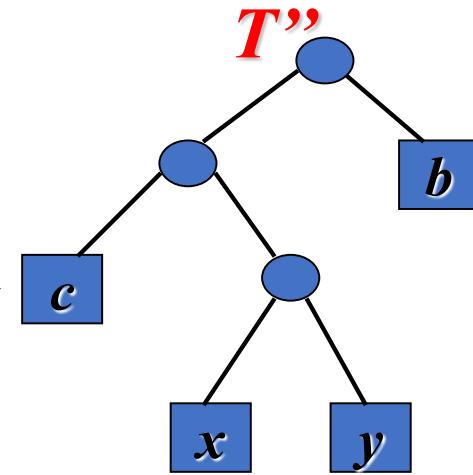
证: 设 T 是 C 的优化前缀树, 且 b 和 c 是具有最大深度的两个兄弟字符:



不失一般性, 设 $f(b) \leq f(c)$, $f(x) \leq f(y)$. 因 x 与 y 是具有最低频率的字符, $f(b) \geq f(x)$, $f(c) \geq f(y)$. 变换 T 的 b 和 x , 从 T 构造 T' :



交换 y 和 c
构造 T''



证 T'' 是最优化前缀树.

$$\begin{aligned} & B(T) - B(T') \\ &= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x) \\ &= (f(b) - f(x))(d_T(b) - d_{T'}(x)). \end{aligned}$$

$\because f(b) \geq f(x), d_T(b) \geq d_{T'}(x)$ (因为 b 的深度最大)

$$\therefore B(T) - B(T') \geq 0, B(T) \geq B(T')$$

同理可证 $B(T') \geq B(T'')$. 于是 $B(T) \geq B(T'')$.

由于 T 是最优化的, 所以 $B(T) \leq B(T'')$.

于是, $B(T) = B(T'')$, T'' 是 C 的最优化前缀编码树.

在 T'' 中, x 和 y 具有相同长度编码, 且仅最后一位不同.

定理. Huffman算法产生一个优化前缀编码树

证. 由于引理1、引理2成立，而且Huffman算法按照引理2的贪心选择性确定的规则进行局部优化这样，所以Huffman算法产生一个优化前缀编码树。

考硬币题. 1.2.5

若查应用题 .

LINEAR PROGRAMMING

Prof. Zhenyu He

Harbin Institute of Technology (Shenzhen)

Autumn 2020

A Political Problem

Policy	urban	suburban	rural
Build roads	-2	5	3
Gun control	8	2	-5
Farm subsidies	0	0	10
Gasoline tax	10	0	-2

The effect of policies on voters.



We format this problem as

Minimize:

$$x_1 + x_2 + x_3 + x_4$$

Subject to:

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$$

$$x_1, x_2, x_3, x_4 \geq 0$$



General linear programs

$$f(x_1, x_2, \dots, x_n) = a_1x_1 + a_2x_2 + \dots + a_nx_n$$

Linear equality

$$f(x_1, x_2, \dots, x_n) = b$$

Linear inequality

$$f(x_1, x_2, \dots, x_n) \leq b$$

or

$$f(x_1, x_2, \dots, x_n) \geq b$$



Considering one linear program with two variables

Maximize:

$$x_1 + x_2$$

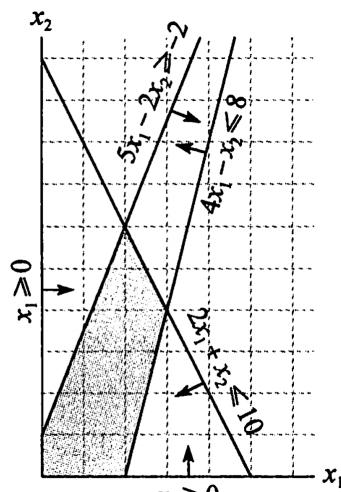
Subject to:

$$4x_1 - x_2 \leq 8$$

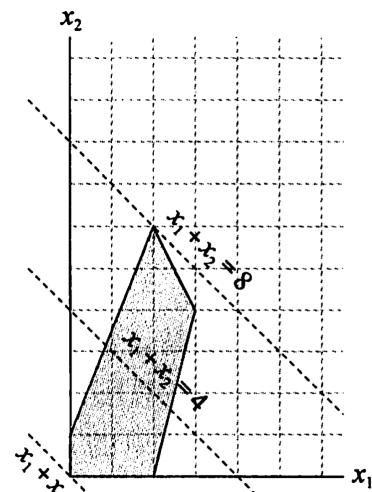
$$2x_1 + x_2 \leq 10$$

$$5x_1 - 2x_2 \geq -2$$

$$x_1, x_2 \geq 0$$



(a)



(b)

Although we cannot easily graph linear programs with more than two variables, the same intuition holds.

If we have three variables, then each constraint is described by a half-space in three dimensional space. The intersection of these half-space forms the feasible region.

If we have n variables, then each constraint is described by a half-space in n dimensional space. The intersection of these half-space forms the feasible region.



Standard Form

In standard form, we are given **n** real numbers C_1, C_2, \dots, C_n ;

m real numbers b_1, b_2, \dots, b_m ;

and **mn** real numbers a_{ij} for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$;

Finding **n** real numbers x_1, x_2, \dots, x_n that

Maximize:

$$\sum_{j=1}^n c_j x_j$$

Subject to:

$$\sum_{j=1}^n a_{ij} x_j \leq b_j \quad \text{for } i = 1, 2, \dots, m$$

$$x_j \geq 0 \quad \text{for } j = 1, 2, \dots, n$$



This linear program can be rewritten as,

Maximize:

$$c^T x$$

Subject to:

$$Ax \leq b$$

$$x \geq 0$$

where $A = (a_{ij})$ $b = (b_i)$ $c = (c_j)$ $x = (x_j)$

A tuple (A, b, c) can represent a linear program in standard form.



Converting linear programs into standard form

For example, if we have the linear program ;

Minimize:

$$-2x_1 + 3x_2$$

Subject to:

$$x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0$$



By negating the coefficients of the objective function,
we get

Maximize:

$$2x_1 - 3x_2$$

Subject to:

$$x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0$$



To ensure that each variable has a non-negativity constraint,
we have

Maximize:

$$2x_1 - 3x'_2 + 3x''_2$$

Subject to:

$$x_1 + x'_2 - x''_2 = 7 \quad \longrightarrow \quad \begin{aligned} x_1 + x'_2 - x''_2 &\leq 7 \\ x_1 + x'_2 - x''_2 &\geq 7 \end{aligned}$$

$$x_1 - 2x'_2 + 2x''_2 \leq 4$$

$$x_1, x'_2, x''_2 \geq 0$$



For consistency in variable names,
we have

Maximize:

$$2x_1 - 3x_2 + 3x_3$$

Subject to:

$$x_1 + x_2 - x_3 \leq 7$$

$$-x_1 - x_2 + x_3 \leq -7$$

$$x_1 - 2x_2 + 2x_3 \leq 4$$

$$x_1, x_2, x_3 \geq 0$$



Converting linear programs into slack form;

We introduce slack variables x_4, x_5, x_6 , the linear programs we just discussed can be written as

Minimize:

$$Z = 2x_1 - 3x_2 + 3x_3$$

Subject to:

$$x_4 = 7 - x_1 - x_2 + x_3$$

$$x_5 = -7 + x_1 + x_2 - x_3$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$$



As in standard form, we use b_i , c_j and a_{ij} to denote constant terms and coefficients. Thus we can concisely define a slack form by a tuple (N, B, A, b, c, v) to denote the slack form

$$Z = v + \sum_{j \in N} c_j x_j$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for} \quad i \in B$$

Here, the equations are indexed by B and the variables on the right-hand side are indexed by N .

For example, in the slack form

$$Z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2}$$

We have $B = \{1,2,4\}$ $N = \{3,5,6\}$

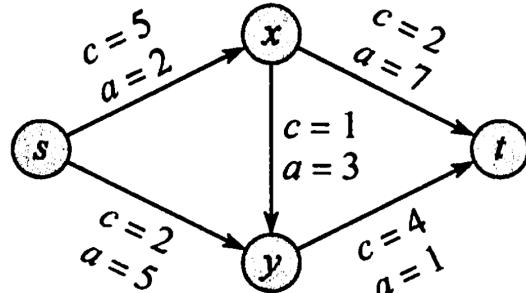
$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix} \quad C = \{c_3, c_5, c_6\}^T = \left\{ -\frac{1}{6}, -\frac{1}{6}, -2/3 \right\}^T$$

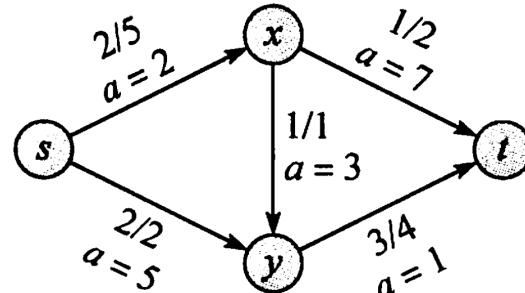
$v = 28$



Minimum cost flow



(a)



(b)

Maximize:

$$\sum_{(u,v) \in E} a(u,v)f(u,v)$$

Subject to:

$$f(u,v) \leq c(u,v) \quad \text{for each } u, v \in V$$

$$f(u,v) = -f(v,u) \quad \text{for each } u, v \in V$$

$$\sum_{v \in V} f(u,v) = 0 \quad \text{for each } u \in V - \{s, t\}$$



The simplex algorithm

Consider the following linear program in standard form:

Maximize:

$$3x_1 + x_2 + 2x_3$$

Subject to:

$$x_1 + x_2 + 3x_3 \leq 30$$

$$2x_1 + 2x_2 + 5x_3 \leq 24$$

$$4x_1 + x_2 + 2x_3 \leq 36$$

$$x_1, x_2, x_3 \geq 0$$



In order to use the simplex algorithm, we must convert the linear program into slack form

Maximize:

$$Z = 3x_1 + x_2 + 2x_3$$

Subject to:

$$x_4 = 30 - x_1 - x_2 - 3x_3$$

$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3$$

$$x_6 = 36 - 4x_1 - x_2 - 2x_3$$

This system has 3 equations and 6 variables, and therefore has an infinite number of solutions.

Basic solution: $(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$

If a basic solution is also feasible, we call it a basic feasible solution.



Since the third constraint is the tightest constraint, we switch x_1 and x_6 .

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$

The linear program is rewritten as

$$Z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}$$

The operation is call pivot.

New basic solution: $(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_6) = (9, 0, 0, 21, 6, 0)$

New objective function: $Z = 27$

Since the third constraint is the tightest constraint again, we switch x_3 and x_5 .

The linear program is rewritten as

$$Z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{16x_6}{5}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5}{8}x_5 - \frac{x_6}{16}$$

New basic solution: $(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_6) = \left(\frac{33}{4}, 0, \frac{3}{2}, \frac{69}{4}, 0, 0\right)$

New objective function:

$$Z = \frac{111}{4}$$



Now the only way to increase the objective value is to increase x_2 .

The linear program is rewritten as

$$Z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2}$$

All coefficients in the objective function are negative, as means the basic solution is the optimal solution.

Basic solution(also optimal solution): $(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_6) = (8, 4, 0, 18, 0, 0)$

Objective function(also final solution) : $Z = 28$



FAST FOURIER TRANSFORM

Prof. Zhenyu He

Harbin Institute of Technology (Shenzhen)

Autumn 2020

Polynomials

A polynomial in the variable x over an algebraic field \mathbf{F} is a representation of a function $\mathbf{A}(x)$ as a formal sum,

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

Where, a_0, a_1, \dots, a_{n-1} are the coefficients of the polynomial.

Degree of $\mathbf{A}(x)$ is k if the highest nonzero coefficient is a_k .

Obviously, n is the degree bound of $\mathbf{A}(x)$.



Polynomial addition and multiplication

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \quad \text{and} \quad B(x) = \sum_{j=0}^{n-1} b_j x^j$$

If $C(x) = A(x) + B(x)$,then $c_j = a_j + b_j$

If $C(x) = A(x)B(x)$,then $c_j = \sum_{k=0}^j a_k b_{j-k}$



Coefficient representation of Polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

Coefficient vector $a = (a_0, a_1, \dots, a_{n-1})$

The operation of evaluating $A(x)$ at a given point x_0

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots + x_0(a_{n-2} + x_0(a_{n-1}))))$$

The complexity of Polynomial addition: $\theta(n)$

The complexity of Polynomial multiplication: $\theta(n^2)$



Point-value representation of Polynomial

Point-value representation of Polynomial $A(x)$ is,

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and a

$$y_k = A(x_k)$$



Theorem 30.1 Uniqueness of an interpolating polynomial

For any set $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k value are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \dots, n-1$

Proof:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$



Uniqueness of an interpolating polynomial

The matrix on the left is denoted as $V(x_0, x_1, \dots, x_{n-1})$, and is known as a Vandermonde matrix, of which the determinant is

$$\prod_{0 \leq j < k \leq n-1} (x_k - x_j)$$

The matrix is invertible if the x_k are distinct.

Thus, the coefficients can be solved for uniquely given the point-value representation.

$$a = V(x_0, x_1, \dots, x_{n-1})^{-1} y$$



Addition of point-value representation

$$C(x) = A(x) + B(x)$$

$$A(x) = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

$$B(x) = \{(x_0, y'_0), (x_1, y'_1), \dots, (x_{n-1}, y'_{n-1})\}$$

$$C(x) = \{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \dots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$$

The complexity of Polynomial addition: $\theta(n)$



Multiplication of point-value representation

$$C(x) = A(x)B(x)$$

Since the degree-bound of C is $2n$, we need $2n$ point-value pairs for a point-value presentation of C .

Extend point-value of A and B to

$$A(x) = \{(x_0, y_0), (x_1, y_1), \dots, (x_{2n-1}, y_{2n-1})\}$$

$$B(x) = \{(x_0, y'_0), (x_1, y'_1), \dots, (x_{2n-1}, y'_{2n-1})\}$$

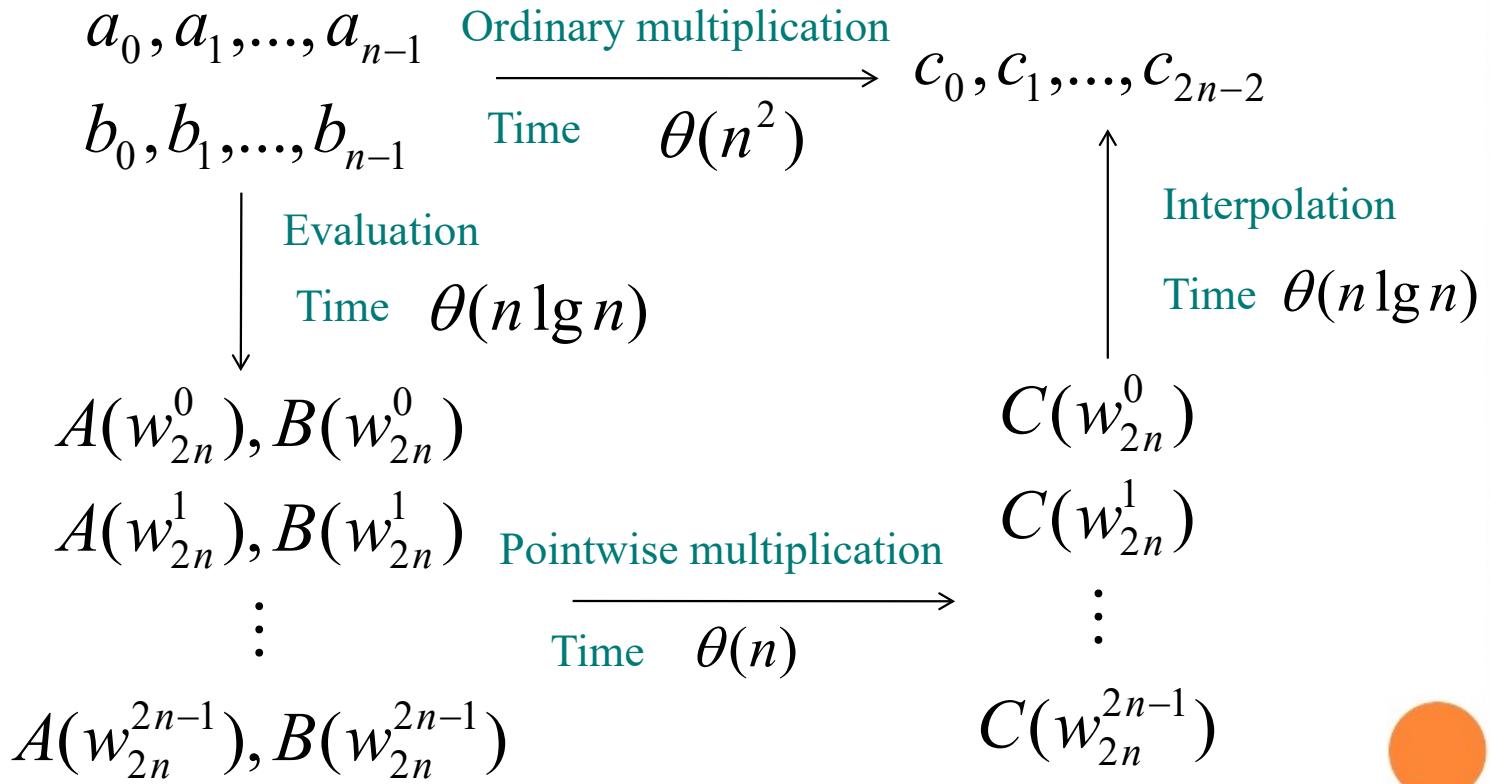
The a point-value representation of C is

$$C(x) = \{(x_0, y_0y'_0), (x_1, y_1y'_1), \dots, (x_{2n-1}, y_{2n-1}y'_{2n-1})\}$$

The complexity of Polynomial multiplication: $\theta(n)$



Graphical outline of efficient polynomial multiplication



DFT

Evaluating a polynomial, $A(x) = \sum_{j=0}^{n-1} a_j x^j$

of degree bound n at $w_n^0, w_n^1, w_n^2, \dots, w_n^{n-1}$

$$y_k = A(w_n^k) = \sum_{j=0}^{n-1} a_j w_n^{kj} \quad k = 0, 1, \dots, n-1$$

The vector $y = (y_0, y_1, \dots, y_{n-1})$ is the discrete Fourier transform of the coefficient vector $a = (a_0, a_1, \dots, a_{n-1})$.



Framework of FFT

The FFT method employs a divide-and-conquer strategy.

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

Where, $A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

So that, the problem of evaluating $A(x)$ at $w_n^0, w_n^1, \dots, w_n^{n-1}$

reduces to

- (1) Evaluating the degree-bound $n/2$ polynomial $A^{[0]}(x)$ and $A^{[1]}(x)$ at $(w_n^0)^2, (w_n^1)^2, \dots, (w_n^{n-1})^2$



Framework of FFT

(2) Combing the result.

According to having lemma, the polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$ th roots of unity.

The recurrence for the running time is,

$$T(n) = 2T(n/2) + \theta(n) = \theta(n \lg n)$$

.



1-D discrete Fourier transform

$$X(j) = \sum_{k=0}^{N-1} A(k) \bullet W^{jk} \quad (j = 0, 1, \dots, N-1) \quad (1)$$

$$A(k) = \frac{1}{N} \sum_{j=0}^{N-1} X(j) \bullet W^{-jk} \quad (K = 0, 1, \dots, N-1) \quad (2)$$

Where, $W = e^{2\pi i / N}$.



Eq.(1) can be rewritten in matrix form as

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ \vdots \\ X(N-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W^{1\cdot 1} & W^{1\cdot 2} & \cdots & W^{1\cdot(N-1)} \\ 1 & W^{2\cdot 1} & W^{2\cdot 2} & \cdots & W^{2\cdot(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W^{N\cdot 1} & W^{2\cdot(N-1)} & \cdots & W^{(N-1)\cdot(N-1)} \end{bmatrix} \begin{bmatrix} A(0) \\ A(1) \\ A(2) \\ \vdots \\ A(N-1) \end{bmatrix} \quad (3)$$

Eq.(3) can be simply denoted as

$$X = F_N A .$$



Examples

$$F_1 = [1] \quad F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Please note that,

(a) $W^0 = 1, W^{N/2} = -1$

(b) $W^{N+r} = W^r, W^{N/2+r} = -W^r$



Idea of Fast Fourier Transform (FFT)

By exchanging the 2nd and 3rd column of F_4 , we have

$$F_4 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & i & -i \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -i & i \end{bmatrix}$$

Denote,

$$\Pi_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \Omega_2 = \begin{bmatrix} 1 \\ i \end{bmatrix}$$



Idea of Fast Fourier Transform (FFT) (con't)

Then, we have

$$F_4 \Pi_4 = \begin{bmatrix} F_2 & \Omega_2 F_2 \\ F_2 & -\Omega_2 F_2 \end{bmatrix} \quad (4)$$

Thus,

$$F_4 \begin{bmatrix} A(0) \\ A(1) \\ A(2) \\ A(3) \end{bmatrix} = \begin{bmatrix} F_2 & \Omega_2 F_2 \\ F_2 & -\Omega_2 F_2 \end{bmatrix} \begin{bmatrix} A(0) \\ A(2) \\ A(1) \\ A(3) \end{bmatrix} = \begin{bmatrix} I & \Omega_2 \\ I & -\Omega_2 \end{bmatrix} \begin{bmatrix} F_2 \begin{bmatrix} A(0) \\ A(2) \end{bmatrix} \\ F_2 \begin{bmatrix} A(1) \\ A(3) \end{bmatrix} \end{bmatrix} \quad (5)$$



Idea of Fast Fourier Transform (FFT) (con't)

Similarly, if $N=2M$,

$$F_N \Pi_N = \begin{bmatrix} F_M & \Omega_M F_M \\ F_M & -\Omega_M F_M \end{bmatrix} \quad (6)$$

Here, $\Omega_M = \text{diag}(1, W, \dots, W^{M-1})$

So,

$$F_N A = \begin{bmatrix} I & \Omega_M \\ I & \Omega_M \end{bmatrix} \begin{bmatrix} F_M A_1 \\ F_M A_2 \end{bmatrix} \quad (7)$$

Here, $A_1 = [A(0), A(2), \dots, A(N-2)]^T$

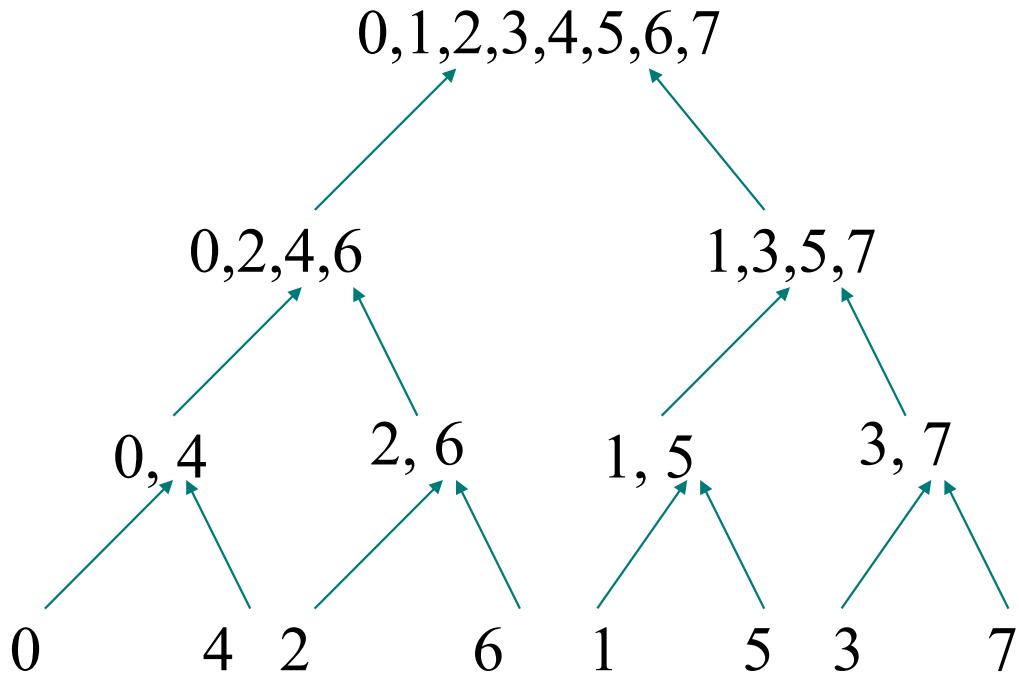
and $A_2 = [A(1), A(3), \dots, A(N-1)]^T$



Idea of Fast Fourier Transform (FFT) (con't)

One example

If $N=8$



Implementation of FFT

Set $N = 2^m$

$$\begin{aligned} X(k) &= \sum_{r=0}^{N/2-1} x(2r)W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1)W_N^{(2r+1)k} \\ &= \sum_{r=0}^{N/2-1} x(2r)W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1)W_{N/2}^{rk} \end{aligned} \quad (8)$$

Here, $W_{N/2} = e^{\frac{2\pi i}{N/2}}$ and $r = 0, 1, \dots, \frac{N}{2} - 1$

Implementation of FFT (con't)

Set

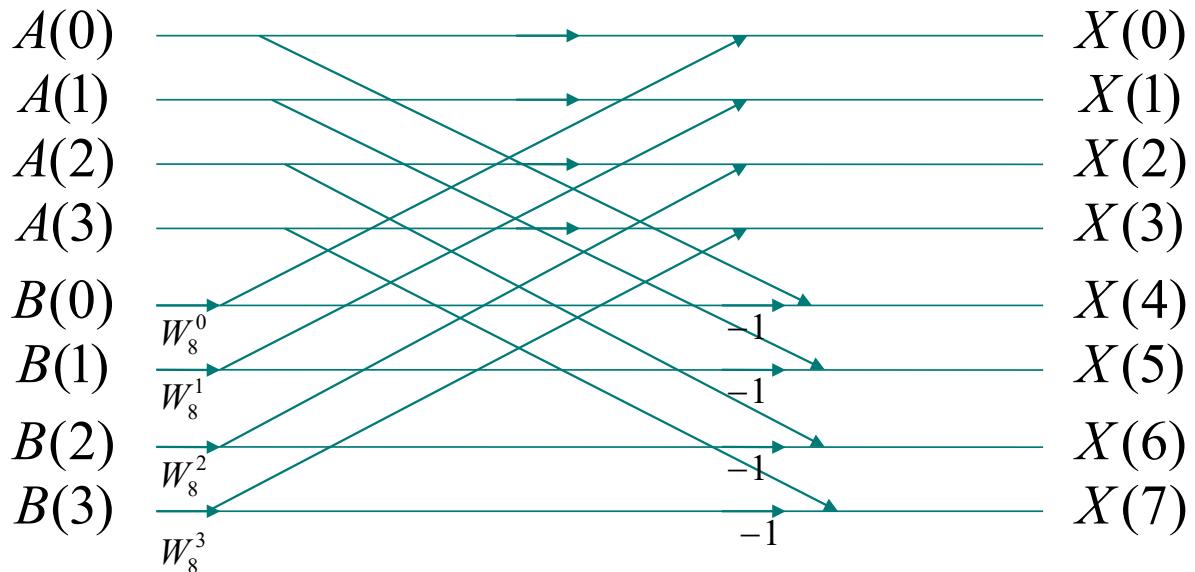
$$A(k) = \sum_{r=0}^{N/2-1} x(2r) W_{N/2}^{rk} \quad k = 0, 1, \dots, \frac{N}{2} - 1$$
$$B(k) = \sum_{r=0}^{N/2-1} x(2r+1) W_{N/2}^{rk} \quad (9)$$

Then,

$$X(k) = A(k) + W_N^k B(k) \quad X(k + N/2) = A(k) - W_N^k B(k) \quad (10)$$

$X(k)$: DFT with N points; $A(k), B(k)$: DFT with $N/2$ points.

Implementation of FFT (con't)



Implementation of FFT (con't)

$$\begin{aligned} A(k) &= \sum_{l=0}^{N/4-1} x(4r)W_{N/2}^{2lk} + \sum_{l=0}^{N/4-1} x(4l+2)W_{N/2}^{(2l+1)k} \\ &= \sum_{l=0}^{N/4-1} x(4l)W_{N/4}^{lk} + W_{N/2}^k \sum_{l=0}^{N/4-1} x(4l+3)W_{N/4}^{lk} \end{aligned}$$

Here, $r = 2l$ and $l = 0, 1, \dots, N/4 - 1$



Implementation of FFT (con't)

Set

$$C(k) = \sum_{l=0}^{N/4-1} x(4l) W_{N/4}^{lk} \quad k = 0, 1, \dots, N/4 - 1$$
$$D(k) = \sum_{l=0}^{N/4-1} x(4l + 2) W_{N/2}^{lk}$$

Then

$$A(k) = C(k) + W_{N/2}^k D(k)$$

$$A(k + \frac{N}{4}) = C(k) - W_{N/2}^k D(k)$$



Implementation of FFT (con't)

Similarly, set

$$E(k) = \sum_{l=0}^{N/4-1} x(4l+1)W_{N/4}^{lk} \quad k = 0, 1, \dots, N/4 - 1$$

$$F(k) = \sum_{l=0}^{N/4-1} x(4l+3)W_{N/2}^{lk}$$

Then

$$B(k) = E(k) + W_{N/2}^k F(k)$$

$$B(k + \frac{N}{4}) = E(k) - W_{N/2}^k F(k)$$



Implementation of FFT (con't)

If N=8, then $C(k)$ $D(k)$ $F(k)$ $E(k)$

all are DFT with 2 points.

$$C(0) = x(0) + x(4)$$

$$E(0) = x(1) + x(5)$$

$$C(1) = x(0) - x(4)$$

$$E(1) = x(1) - x(5)$$

$$D(0) = x(2) + x(6)$$

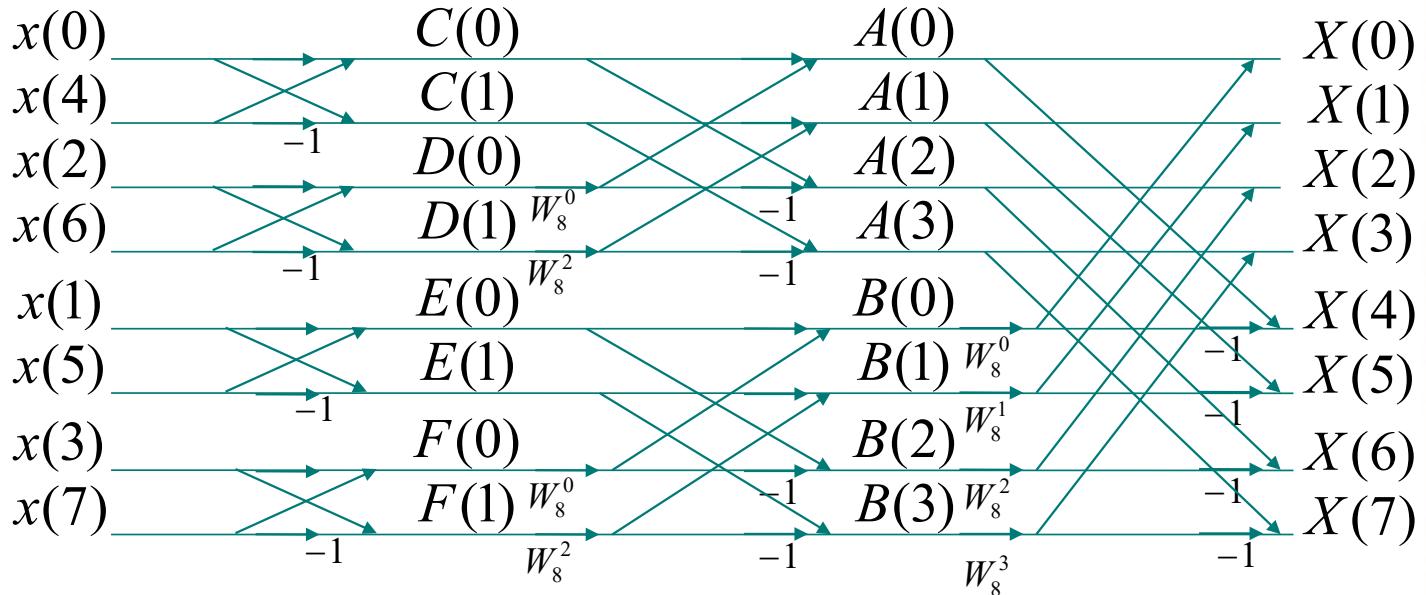
$$F(0) = x(3) + x(7)$$

$$D(1) = x(2) - x(6)$$

$$F(1) = x(3) - x(7)$$



Implementation of FFT (con't)



bit-reverse

Original index 1, binary code (001) \rightarrow (100), output index 4.



NP - complete

若要：

- ① P. NP. NPC. NP-hard 问题的证明.
- ② 设计近似算法.

NP-complete

Prof. Zhenyu He

Harbin Institute of Technology, Shenzhen

大纲

- 1. 问题的复杂程度
- 2. P问题与NP问题
- 3. NP-hard问题与NPC问题
- 4. NPC问题的常用解决策略



1. 问题的复杂程度

- 1.1 问题的引入
- 1.2 多项式时间
- 1.3 决策与优化问题



1.1 问题

- 面对一个问题，我们首先会考虑，它是否可以用现有算法解决。其次考虑，现有算法是否足够快。
- 一个算法时间复杂度是 $O(n^4)$ ，它完全是可以接受的。因为它在一个非常大的输入下，仍然能在合理的时间内得到结果。
- 那么问题的复杂程度该如何定义呢？



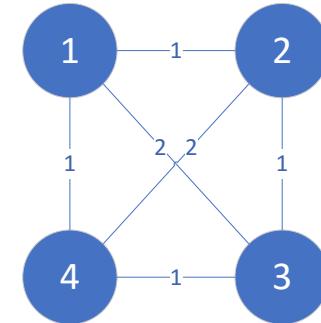
1.2 多项式时间 (Polynomial time)

- 我们采用多项式时间界限来定义一个问题的难易程度。
- 当我们判定一个问题是否可以找到一个多项式时间界限的算法，有以下三种情况：
 - Yes (找到即可)
 - No (证明)
 - 证明该问题的所有算法是指数时间的
 - 证明根本不存在多项式时间界限的算法去解决该问题。
 - Not sure (目前没找到，不代表没有，也就是后面的NPC问题)



1.2 多项式时间 (Polynomial time)

- 旅行商问题 (Traveling Salesman Problem) :
- 找出一条通过所有城镇并回到原点的最短路径, $1 \rightarrow 2 \rightarrow 3$ ($1 \rightarrow 3 \rightarrow 2$) $\rightarrow \dots$ 这两种走法的距离不一样。
- 可能的路线有 : $n!$



- 一个朴素的思想就是遍历所有路径, 需要 $O(n!)$ 的时间。
- 采用分支界限法时间复杂度为 $O(2^n)$, 采用动态规划时间复杂度为 $O(2^n * n^2)$ 。
- 找不到多项式时间内的算法对于计算机来说就是 “难以解决的问题”。

1.2 多项式时间 (Polynomial time)

- 一个问题可以在多项式时间内解决，称该问题是易解决的。
- 一个问题不能在多项式时间内解决，称该问题是不易解决的。



1.3 决策与优化问题

- 决策问题：预期输出为“是”或“否”，1或0的计算问题
- 优化问题：我们试图构造一个解，最大化或最小化某些值的计算问题
- Example : TSP问题
 - 优化问题：给定一个带权图，找出一条通过所有点并回到原点的最短路径。
 - 判定问题：给定一个带权图和一个数k，是否存在一条通过所有点并回到原点的路径的总权重小于等于k。

P问题与NP问题



2 P问题与NP问题

- 2.1 P问题
- 2.2 NP问题
- 2.3 P问题与NP问题的关系
- 2.4 解决 $NP = P$ 的意义



2.1 P问题

- 问题：
 - 求序列{3,1,2,4,7}的中位数（排序后，找 $n/2$ 位置的数）
 - 判断 $k=3$ 是否是上述序列的中位数（思考）
- 显然可以找到一个算法可以在多项式解决上述问题。



2.1 P问题

- P问题的定义：能在多项式时间内解决的问题称为P问题
- 还有哪些P问题呢？
 - 求序列{3,1,2,4,7}的最大值
 - 判断k=4是否是上述序列的最大值



2.2 NP问题 (non-deterministic polynomial)

- 问题：对41607317做质因数分解。
- 思考：如果给出一个可能的答案8699和4783，能否在一个多项式时间内的验证这两个数是正确答案？



2.2 NP问题:3-SAT问题

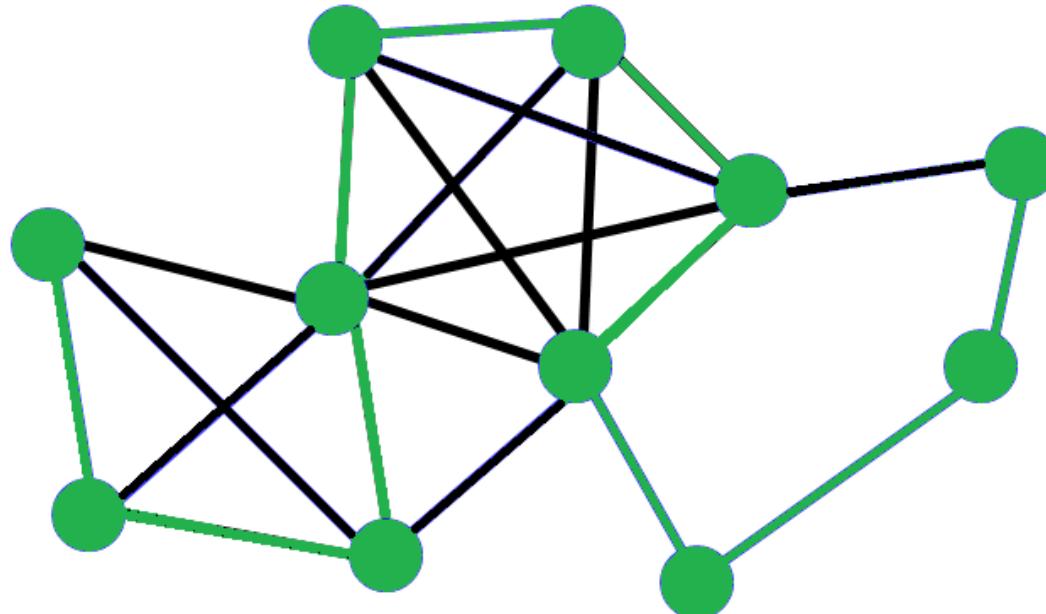
- 问题：bool集合 $\{x_1, x_2, x_3, x_4, x_5\}$, 是否存在一组赋值使 $(x_1 \mid\mid x_2 \mid\mid x_3) \&\& (x_1 \mid\mid !x_2 \mid\mid !x_3) \&\& (!x_1 \mid\mid x_4 \mid\mid x_5) == \text{true}$? (3-SAT问题)

x_1	x_2	x_3	x_4	x_5
0	0	0	0	0
0	0	0	0	1
...

- 思考：
 - 如何解决上述问题(2^5)
 - 如果给一组解 $xi=\{0,1,1,0,1\}$ 是否存在一个多项式时间的算法验证该解的正确性
 - 是否存在一个多项式时间的算法来解决上述问题（目前没找到）

2.2 NP问题：哈密尔顿环问题

- 哈密尔顿环问题 (HSP) : 给定无向图 $G=(V,E)$, 判定其是否经过图中每个顶点且仅一次的回路。



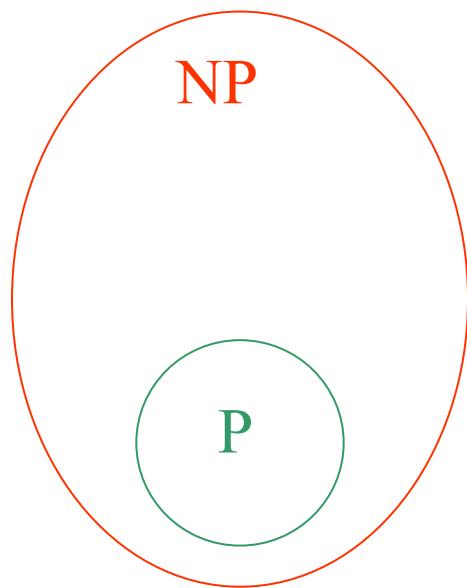
2.2 NP问题

- NP问题的定义：在多项式时间内能被验证的问题。
- 考虑：
 - 求中位数是NP问题吗？（我们已经知道它是一个P问题）
 - 3-SAT问题是一个NP问题吗，是一个P问题吗？



2.3 P问题与NP问题的关系

- 显然所有P问题是NP问题
 - 在多形式时间内可解，那么一定在多项式时间内可验证



2.3 P问题与NP问题的关系

- 那么所有NP问题都是P问题吗？
 - 求中位数是一个NP问题，同时它也是一个P问题
 - 3-SAT是一个NP问题，它是一个P问题吗？
- 目前没有人证明 $P=NP$ ，这依然是一个待解决的问题。



2.4 解决NP = P的意义

如果验证了 $NP = P$ ，说明一个问题如果能在多项式时间被验证，就可以找到一个算法能在多项式时间内解决这问题。计算机会变得更加“聪明”，能够在十分混乱的局面，快速找到一条捷径。



2.4 解决NP = P的意义

1. 运筹学中的很多难题都将被轻松解决，比如说整数规划问题、旅行商问题等等。
2. 当获知了所有的信息以后，计算机可以在极短的时间里，对证券市场、天气、球赛结果做出非常准确的预测。
3. 蛋白质的折叠问题也是一个 NPC 问题，新的算法无疑是生物与医学界的一个福音。

.....



NP-hard与NPC



3 NP-hard与NPC

- 3.1 归约与多项式时间归约
- 3.2 NP-hard与NPC问题
- 3.3 P,NP,NP-H,NP-C的关系
- 3.4 讨论NPC问题的意义



3.1 归约与多项式时间归约

- 问题A：求解一个一元一次方程
- 问题B：求解一个一元二次方程

$$ax + b = c \xrightarrow{\text{归约}} 0x^2 + ax + b = c$$

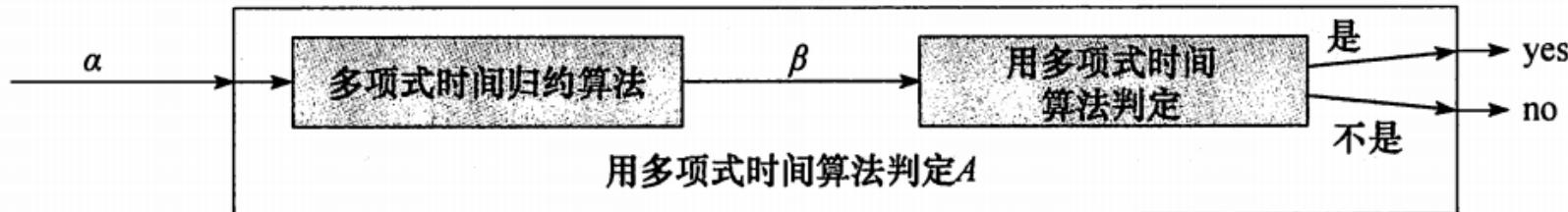
- 如果存在能有效解决问题B的算法，也可以作为解决
问题A的子程序，我们则称问题A “**可归约**” 到问题
B。
- 如果我可以把问题A中的一个实例转化为问题B中
的一个实例，然后通过解决问题B间接解决问题A，那
么就认为B比A更难。

3.1 归约与多项式时间归约

考虑一个判定问题A，希望再多项式时间内解决该问题；同时有另一个不同的判定问题B，我们知道如何在多项式时间内解决它。最后假设有一个过程，它可以将A的任何实例 α 转化为B的具有以下特征的某个实例 β ：

- 转换操作需要多项式时间
- 两个实例的解是相同的。也就是说， α 的解是“Yes”，当且仅当 β 的解也是“Yes”。

我们称这一过程为**多项式时间归约算法**。我们就可以求解问题A的任一实例，如下图：



3.1 归约与多项式时间归约

- 哈密尔顿环问题 (HSP) : 给定无向图 $G=(V,E)$, 判定其是否经过图中每个顶点且仅一次的回路。
- 旅行商问题 (Traveling Salesman Problem) : 找出一条通过所有城镇并回到原点的最短路径, $1 \rightarrow 2 \rightarrow 3$ ($1 \rightarrow 3 \rightarrow 2$) $\rightarrow \dots \dots$ 这两种走法的距离不一样。
- 思考 : 如何将HSP问题归约为旅行商问题 ?



3.1 归约与多项式时间归约

Hamilton回路可以归约为TSP问题(Travelling Salesman Problem, 旅行商问题)：

- 在Hamilton回路问题中，我们假设两点相连即这两点距离为0，两点不直接相连则令其距离为1，于是问题转化为在TSP问题中，是否存在一条长为0的路径。Hamilton回路存在当且仅当TSP问题中存在长为0的回路



3.1 归约与多项式时间归约

- 通过对某些问题归约，我们能够寻找复杂度高但是应用范围广的算法代替复杂度低但是只能应用于很小一类的算法。



3.2 NP-hard与NPC问题

- NP-hard : 所有NP问题都可以在多项式时间复杂度内归约到问题D, 则D问题成为NP-hard问题。
- NPC (NP-completeness) : 所有NP问题都可以在多项式时间复杂度内归约到NP问题D, 则NP问题D成为NPC问题。

NPC问题D应该满足以下两点性质：

- D问题是NP-hard问题
- D问题是NP问题



3.2 NP-hard与NPC问题

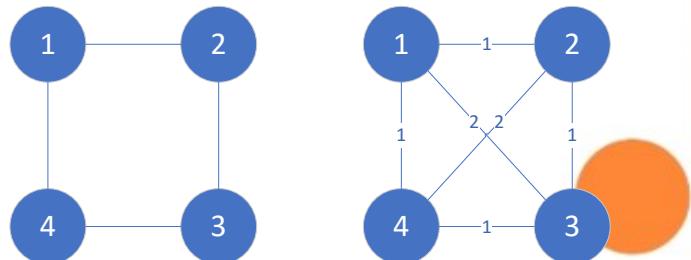
- 通俗的讲，一个问题可以在多项式时间内验证，但目前没有找到多项式时间的算法解决这个问题，我们称这类问题为NPC问题。
- 第一个被证明的NPC问题——3-SAT问题
- 哈密顿环问题也是一个NPC问题。
- 证明一个NP问题D是NPC问题，只需将一个已知的NPC问题通过多项式归约到问题D，则D也是一个NPC问题。



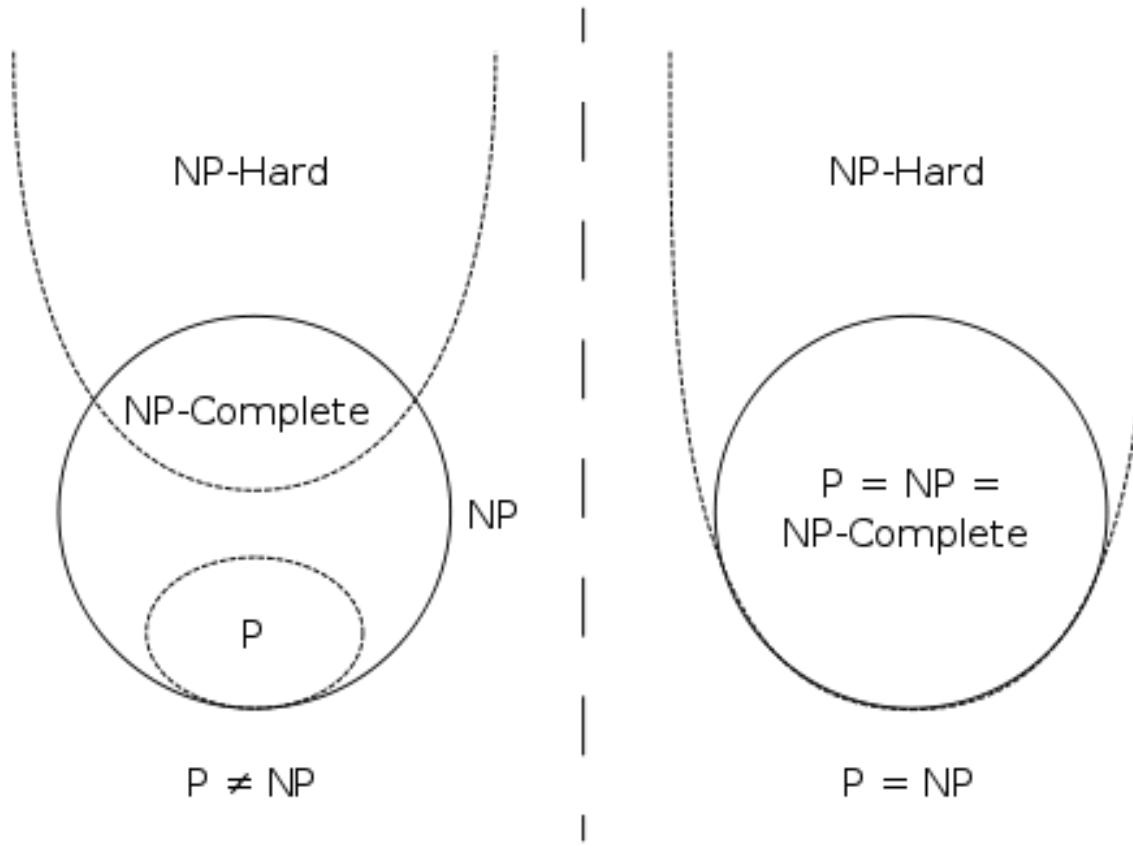
3.2 NP-hard与NPC问题

- 证明TSP是一个NPC问题（已知HSP是NPC问题）。
- 构造归约函数T：给定一个无权图G和一个顶点一致的带权完全图G'。G中两个节点之间有边，则G'对应的两个节点之间的边的权重为0，否则为1。然后令K=0。
- 归约函数T可在多项式时间内实现。
- TSP问题本身也是一个NP问题（可以验证）。

- 所以TSP问题也是一个NPC问题



3.3 P,NP,NP-H,NP-C的关系



3.4 讨论NP-C问题的意义

- 为什么把NP里最难的问题拿出来讲？(NPC)



NPC问题的常用解决策略



4 NPC问题的常用解决策略

- 4.1 指数级算法
- 4.2 近似算法



4.1 指数级算法

- 常用来解决NPC问题的指数级算法有：
 - 动态规划
 - 分支界限法
 - 回溯法



回溯法概念

- 回溯算法实际上是一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。
- 许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。



回溯法基本思想

- 在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。

回溯法解题步骤

- 回溯法的基本解题步骤：

- (1) 针对所给问题，确定问题的解空间：首先应明确定义问题的解空间，问题的解空间应至少包含问题的一个（最优）解。
- (2) 确定结点的扩展搜索规则
- (3) 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。



回溯法举例：0-1背包问题

- 有N件物品和一个容量为V的背包。第*i*件物品的价格（即体积，下同）是 $w[i]$ ，价值是 $c[i]$ 。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。
- 这是最基础的背包问题，总的来说就是：选还是不选，这是个问题
- 相当于用 $f[i][v]$ 表示前*i*个物品装入容量为*v*的背包中所可以获得的最大价值。
- 对于一个物品，只有两种情况

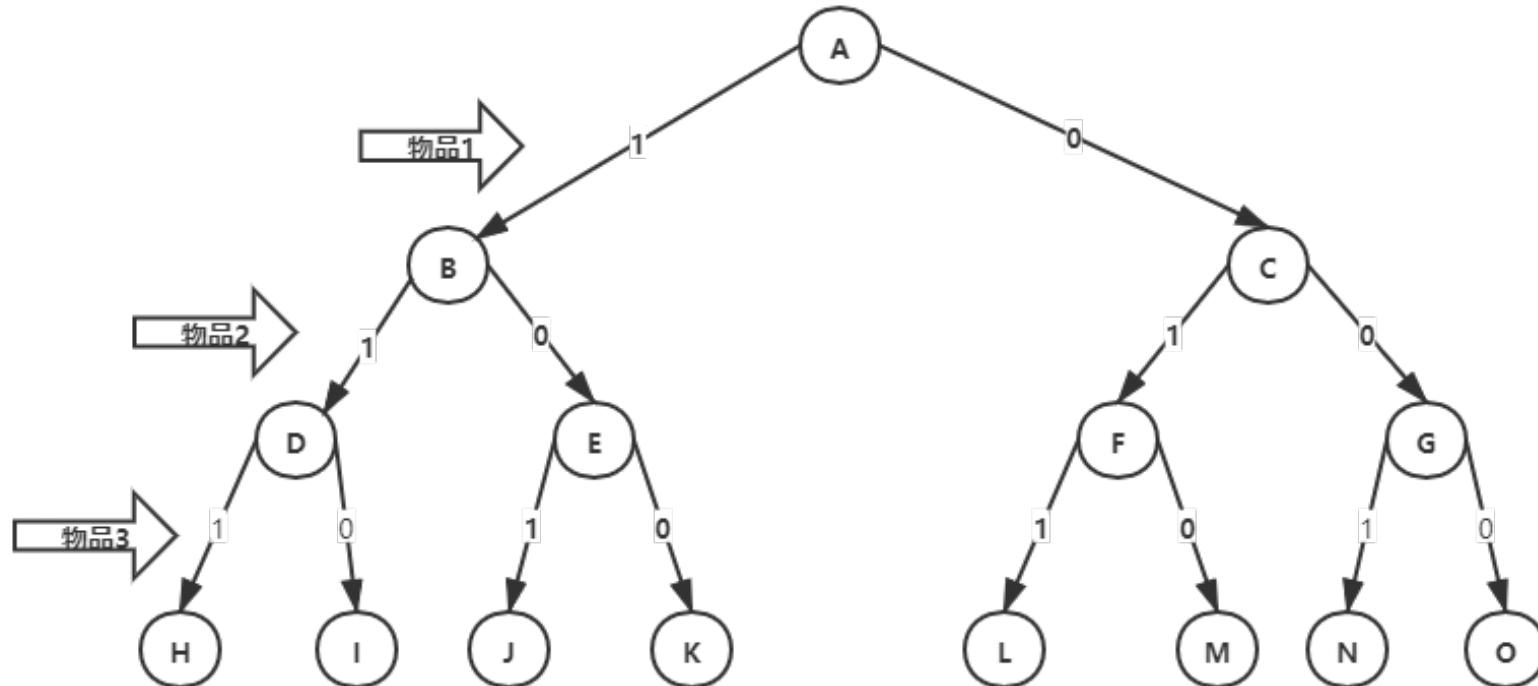
 情况一：第*i*件不放进去，这时所得价值为： $f[i-1][v]+0$

 情况二：第*i*件放进去，这时所得价值为： $f[i-1][v-c[i]]+w[i]$



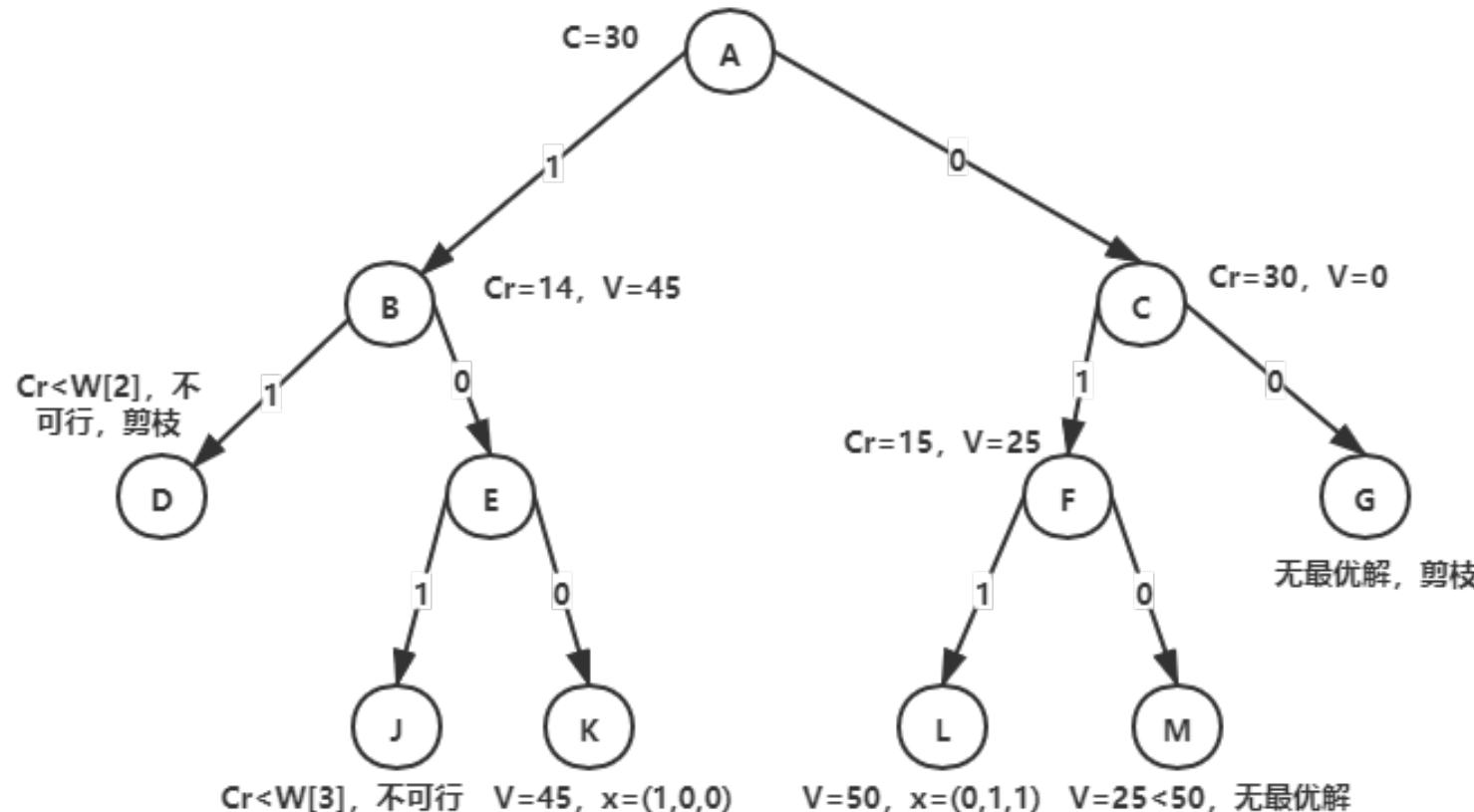
回溯法举例：0-1背包问题

- 用回溯法解问题时，应明确定义问题的解空间。问题的解空间至少包含问题的一个(最优)解。对于 $n=3$ 时的 0/1 背包问题，可用一棵完全二叉树表示解空间，如图所示：



回溯法举例：0-1背包问题

例如 $n=3, C=30$ (最大容量), $w=\{16, 15, 15\}$, $v=\{45, 25, 25\}$



- 可见，最终搜索了4条路径，比原本解空间8条路径减少了一半，进行了一定程度上的优化，但其复杂度仍为 $O(2^n)$.

4.2 近似算法

近似算法的设计思想

- 放弃求解最优解，用近似最优解代替最优解，以此换取：
 - 算法设计上的简化
 - 时间复杂度的降低
- 近似算法是可行的：
 - 问题的输入数据是近似的
 - 问题的解允许有一定程度的误差
 - 近似算法可在较短的时间内得到问题的近似解



近似算法的设计思想和性能

- 衡量近似算法性能的标准：
 - 时间复杂度：必须是多项式阶的
 - 解的近似程度：近似比
- 近似比：若一个最优化问题的最优值为 c^* ，求解该问题的一个近似算法求得的近似最优值为 c ，则该近似算法的近似比为：

$$\eta = \max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\}$$

对于最小化问题, $c \geq c^*$
对于最大化问题, $c \leq c^*$

$\eta > 1$, 且 η 越大, 近似解越差。

- 通常情况下, 该近似比是问题的输入规模的一个函数 $\rho(n)$:

$$\max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\} \leq \rho(n)$$

- 如果一个算法的近似比达到 $\rho(n)$, 则称该算法为 $\rho(n)$ 近似算法



近似算法的性能

- 近似算法的相对误差 λ ：

$$\lambda = \left| \frac{c - c^*}{c^*} \right|$$

- λ 表示一个近似最优解与最优解相差的程度

- 若问题的输入规模为 n , 存在一个函数 $\varepsilon(n)$, 使得：

$$\left| \frac{c - c^*}{c^*} \right| \leq \varepsilon(n)$$

则 $\varepsilon(n)$ 称为近似算法的相对误差界。且有：

$$\varepsilon(n) \leq \rho(n) - 1$$



近似算法举例：顶点覆盖问题

- 问题描述：无向图 $G=(V,E)$ 的顶点覆盖是它的顶点集 V 的一个子集 $V' \subseteq V$ ，使得若 (u,v) 是 G 的一条边，则 $v \in V'$ 或 $u \in V'$ 。顶点覆盖 V' 的大小是它所包含的顶点个数 $|V'|$ 。
- 顶点覆盖问题：找最小顶点覆盖

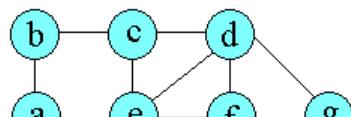
VertexSet ApproxVertexCover (Graph G)

```
{ Cset=∅ ;  
  E1=E ;  
  while (E1 != ∅) {  
    从E1中任取一条边e=(u,v) ;  
    Cset=Cset ∪ {u,v} ;  
    从E1中删去与u和v相关联的所有边 ;  
  }  
  return Cset  
}
```

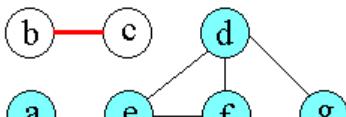
Cset用来存储顶点覆盖中的各顶点。初始为空，不断从边集E1中选取一边 (u, v) ，将边的端点加入Cset中，并将E1中已被u和v覆盖的边删去，直至Cset已覆盖所有边。即E1为空。

- 算法时间复杂度是多少？ $O(V+E)$

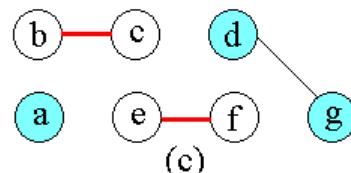
近似算法举例：顶点覆盖问题



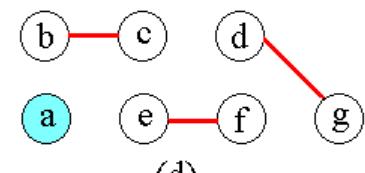
(a)



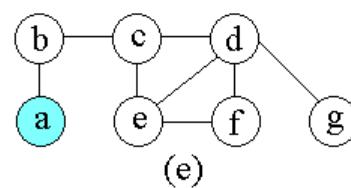
(b)



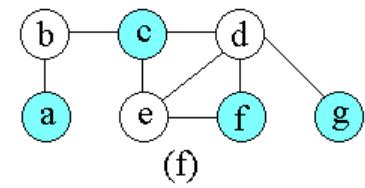
(c)



(d)



(e)



(f)

图(a)~(e)说明了算法的运行过程及结果。(e)表示算法产生的近似最优顶点覆盖Cset，它由顶点b, c, d, e, f, g所组成。
(f)是图G的一个最小顶点覆盖，它只含有3个顶点：b, d和e。

算法**approxVertexCover**的近似比为2。 ($c=6$, $c^*=3$)

近似算法举例：顶点覆盖问题

- 定理：算法ApproxVertexCover是2近似算法。
- 证明：

令 A 表示由算法ApproxVertexCover选取的一组边。令 C^* 为最优顶点覆盖。然后， C^* 必须至少包含 A 中每条边的一个端点。又由于 A 中没有两个边共享端点，因此 A 中没有两个边被 C^* 的同一顶点覆盖。

因此，我们有最优顶点覆盖的规模下界： $|C^*| \geq |A|$ 。

另一方面，该算法选择 A 中每条边的两个端点，有 $|C| = 2|A|$ 。
因此， $|C| = 2|A| \leq 2|C^*|$ 。



结束

