

Algorithm Design Strategies

Divide-and-Conquer Strategy

- Three steps:
 1. Divide a problem into sub problems of smaller sizes
 2. Recursively solve smaller problems
 3. Merge the solutions of sub problems to arrive at a solution to original problem.
- Tower of Hanoi problem – Given 3 pegs move a stack of different sized disks from one peg to another making sure that a smaller disk is always on top of a larger disk.
- Integer multiplication
- Other examples : MergeSort , QuickSort, FFT

Tower of Hanoi

move(n,i,j):

Input : n disks and integers i and j s.t. $1 \leq i, j \leq 3$

Output: Disk moves to migrate all n disks from i to j

if $n = 1$

 move a disk from i to j

else

 otherPeg <- $6-(i+j)$

move($n-1, i, \text{otherPeg}$)

move($1, i, j$)

move($n-1, \text{otherPeg}, j$)

- **Timecomplexity:** $O(T(n))$ – T(n) is number of disk moves

$$T(n) = 2 T(n-1) + 1, n > 1 \text{ and } T(1) = 1 \Rightarrow T(n) = 2^n - 1$$

Divide-and-Conquer

- In most problems, we divide a problem into sub problems which are a fraction of size of the original problem.
- General time-complexity recurrence for divide-and-conquer:

$$\begin{aligned} T(n) &= a T(n/b) + f(n), \quad n \geq d \\ &= c \text{ for } n < d \end{aligned}$$

- Second term includes dividing and merging steps; first term includes number of basic problems to solve
- Based on $f(n)$, general asymptotic solutions for $T(n)$ can be derived in some cases.

Divide-and-conquer recurrence (Master theorem)

- At recursion level 1, there are "a" sub problems of size $\frac{n}{b}$ to solve and $f(n)$ additional time for divide/merge overhead
- At recursion level 2, there are a^2 sub problems of size $\frac{n}{b^2}$ to solve and $a^2 f(\frac{n}{b^2})$ additional time for divide/merge overhead.
- At i-th step there are a^i sub problems of size $\frac{n}{b^i}$ to solve and $a^{(i-1)} * f(\frac{n}{b^{i-1}})$ additional time for divide/merge overhead.
- Setting $i=\log_b(\frac{n}{d})$, we see that ultimately we need to spend time, say $T1(n)$ in solving $a^{\log_b(\frac{n}{d})} = \frac{n^{\log_b a}}{a^{\log_b d}}$ problems of size d (each of which takes “c” time units) and in divide/merge steps which takes a total of time units $T2(n) = \sum_{j=0}^{\log_b(\frac{n}{d})-1} a^j * f(\frac{n}{b^j})$

Divide-and-conquer recurrence (Master theorem)

- $T_1(n) = \frac{n^{\log_b a}}{a^{\log_b d}}$ $T_2(n) = \sum_{j=0}^{\log_b(\frac{n}{d})-1} a^j * f\left(\frac{n}{b^j}\right)$

Case 1 : If $f(n)$ is $O(n^{\log_b a - \epsilon})$ for small constant $\epsilon > 0$ then $T_1(n)$ dominates $T_2(n)$ asymptotically and **$T(n)$ is $\Theta(n^{\log_b a})$**

Case 2 : If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T_2(n)$ dominates $T_1(n)$ asymptotically and **$T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$**

Case 3 : If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ for small constant $\epsilon > 0$ and $a * f(n/b) \leq \delta f(n)$ for $n \geq d$ where $\delta < 1$ then $T_2(n)$ dominates $T_1(n)$ asymptotically and **$T(n)$ is $\Theta(f(n))$**

Master theorem examples

- $T(n) = 4 T(n/2) + cn$ for $k > 0$
a=4, b=2 and $f(n) = cn$. Case 1 applies here as $f(n)$ is $O(n^{\log_2 4 - \epsilon})$ for $\epsilon = 1$. Hence $T(n)$ is $\Theta(n^{\log_2 4})$ which is $\Theta(n^2)$
- $T(n) = 7 T(n/2) + cn^2$ for $k > 0$
a=7, b=2 and $f(n) = cn^2$. Case 1 applies here as $f(n)$ is $O(n^{\log_2 7 - \epsilon})$ for $\epsilon = 0.1$. Hence $T(n)$ is $\Theta(n^{\log_2 7})$
- $T(n) = 2 T(n/2) + c n$ for $c > 0$
a=2, b=2 and $f(n) = cn$. Case 2 applies as $f(n)$ is $\Theta(n^{\log_2 2} \log^k n)$ where $k = 0$ and also $f(n)$ is not $O(n^{\log_2 2 - \epsilon})$ for any $\epsilon > 0$, so case 1 does not apply. Hence $T(n)$ is $\Theta(n \log^{k+1} n)$ which is $\Theta(n \log n)$
- $T(n) = 2 T(n/2) + c n \log n$ for $c > 0$
a=2, b=2 and $f(n) = cn \log n$. Case 2 applies as $f(n)$ is $\Theta(n^{\log_2 2} \log^k n)$ where $k = 1$. Hence $T(n)$ is $\Theta(n \log^{k+1} n)$ which is $\Theta(n \log^2 n)$

Master theorem examples

- $T(n) = T(n/2) + c$ for $c > 0$

$a=1, b=2$ and $f(n) = c$. $f(n)$ is not $O(n^{\log_2 1 - \epsilon})$ for any $\epsilon > 0$, so case 1 does not apply. Also $f(n)$ is $\Theta(n^{\log_2 1} \log^k n)$ where $k=0$. Hence case 2 applies so $T(n)$ is $\Theta(n^{\log_2 1} \log^{k+1} n)$ which is $\Theta(\log n)$

- $T(n) = 2 T(n/2) + c n^{1.5}$ for $c > 0$

$a=2, b=2$ and $f(n) = c n^{1.5}$. $f(n)$ is not $O(n^{\log_2 2 - \epsilon})$ for any $\epsilon > 0$, so case 1 does not apply.

$f(n)$ is not $\Theta(n^{\log_2 2} \log^k n)$ for any $k \geq 0$ (remember $\log^k n$ is $o(n^r)$ for any $r > 0$). So case 2 does not apply.

But $f(n)$ is $\Omega(n^{\log_2 2 + \epsilon})$ for $\epsilon = 0.5$ and

$a * f(n/b) = 2 c * (\frac{n}{2})^{1.5} = \delta n^{1.5}$ where $\delta = \frac{c}{2^{0.5}}$. Hence case 3 applies. Hence $T(n)$ is $\Theta(f(n))$ which is $\Theta(n^{1.5})$

Integer multiplication

- We are interested in finding efficient algorithm for multiplying two n-bit integers where the time complexity is measured in number of bit operations (e.g. single bit operations such as addition with carry, shift etc.)
- Useful for software based integer multiplications as in Java BigInteger data types.
- Simple algorithm uses same approach as hand multiplication of 2 numbers (add and shift). This takes $(n-1)$ shifts of at most $2n$ bits and addition of n integers each of which is at most $2n$ bits. This approach takes $\Theta(n^2)$ time.
- We can do better than that using Divide-and-conquer.
- Assume n is a power of 2 for simplicity. Let X and Y be two n-bit integers. For n large enough, we divide X into two parts a and b where b has $n/2$ least significant bits of X and a has $n/2$ most significant bits of X . Same way we divide Y into two parts c and d .
- In practice, if n is not even, we divide them into approximately equal halves.

Algorithm Design Strategies (contd.)

Integer multiplication

- We are interested in finding efficient algorithm for multiplying two n-bit integers where the time complexity is measured in number of bit operations (e.g. single bit operations such as addition with carry, shift etc.)
- Useful for software based integer multiplications as in Java BigInteger data types.
- Simple algorithm uses same approach as hand multiplication of 2 numbers (add and shift). This takes $(n-1)$ shifts of at most $2n$ bits and addition of n integers each of which is at most $2n$ bits. This approach takes $\Theta(n^2)$ time.
- We can do better than that using Divide-and-conquer.
- Assume n is a power of 2 for simplicity. Let X and Y be two n-bit integers. For n large enough, we divide X into two parts a and b where b has $n/2$ least significant bits of X and a has $n/2$ most significant bits of X . Same way we divide Y into two parts c and d .
- In practice, if n is not even, we divide them into approximately equal halves.

Integer multiplication (contd.)

- Easy to see that $X = 2^{n/2} * a + b$ and $Y = 2^{n/2} * c + d$
- $X * Y = 2^n * (ac) + 2^{n/2} * (ad + bc) + (bd)$
- Note ac , ad , bc and bd are products of $n/2$ -bit integers.
Multiplying an integer by 2^i is same as shifting “ i ” bits to left.
- If we use divide-conquer straight away, we need to solve 4 sub problems of size $n/2$ recursively and in addition needs to perform $O(n)$ bit operations for shifting and adding
 $\Rightarrow T(n) = 4 T(n/2) + kn \Rightarrow T(n)$ is $\Theta(n^2)$, no improvement!!
- Instead we try to reduce multiplications at the expense of some additions/subtractions.
- $(a - b)(d - c) = ad - (bd + ac) + bc \Rightarrow (ad + bc) = (a - b)(d - c) + (ac + bd)$
- $X * Y = 2^n * (ac) + 2^{n/2} * [(a - b)(d - c) + (ac + bd)] + (bd)$

Integer multiplication (contd.)

- This requires :
 - (a) 2 subtractions ($a-b, d-c$) with $n/2$ -bit integers
 - (b) 3 multiplications [$ac, bd, (a-b)(d-c)$] of $n/2$ -bit integers
 - (c) 4 additions of at most n -bit integers
 - (d) 1 n -bit shift and 1 $n/2$ -bit shift of at most n -bit integers

(a), (c) and (d) takes $O(n)$ bit operations, (b) requires 3 recursive calls on $n/2$ -bit integers.

$$T(n) = 3 T(n/2) + k n, \text{ for } n > d$$

$$= c \text{ for } n = d$$

Applying Case 1 of Master theorem, we get $T(n)$ is $\Theta(n^{\log_2 3})$ which is better than $\Theta(n^2)$

Smooth functions and time complexity extensibility (optional)

- In many divide-and-conquer algorithms , we derive $T(n)$ assuming n to be a power of some integer $b > 1$.
- Can we extend this result for any n asymptotically ?
- We can do this as long as $T(n)$ involves “smooth” functions.
- **Definition 1 :** A function $f(n)$ is eventually non-decreasing if there exists $n_0 \geq 0$ such that for all $n_2 \geq n_1 \geq n_0$, $f(n_2) \geq f(n_1)$
- **Definition :** A function $f(n)$ is smooth iff $f(n)$ is “eventually” non-decreasing and $f(2n)$ is $O(f(n))$
- Can show that for a smooth function f , $f(bn)$ is $O(f(n))$ for any fixed positive integer b .
- Examples of smooth functions : n , $\log n$, n^2 .
- Is 2^n a smooth function ? No

Smooth functions and time complexity extensibility (optional)

- Extensibility Theorem:

Suppose $T(n)$ is $O(f(n))$ when n is a power of b for some constant integer $b > 1$ and $T(n)$ is asymptotically non-decreasing (usually the case with time-complexities)

Then we can say $T(n)$ is $O(f(n))$ for any n provided $f(n)$ is a smooth function.

Dynamic Programming Paradigm

- Discovered by Richard Bellman for solving various optimal decision problems.
- Applicable for problems with objective functions that satisfy “**optimal substructure property**” (or principle of optimality). It allows an objective function to be broken down into a series of recursive functions each with smaller number of decision variables.
- At each stage, no decision made but we compute best solution for each of all possible states at that stage. (Decision graph)
- We then work backward from final stage as only one possible state at that stage.

Dynamic Programming

- Efficiency of DP is due to avoiding repeated computation for a state at a decision stage. How we arrive at that state is not important. T
- Recursive formulation but no repeated computation!!
- Related to “memoization” which is caching of a result during recursive computation.
e.g. Pascal triangle
- Table underlying DP computations.
- **Example problems:** matrix chain product, multiple joins of relations in RDB, Optimizing string edits, Longest Common Subsequence(LCS), DNA sequence alignment,
Hidden Markov Models (speech recognition)

Minimum edit distance

Problem:

Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$
compute minimum cost to transform X to Y using following
operations:

- (a) Insert a new character into X – $c_{\text{ins}}(\cdot)$ units cost
- (b) Delete a character from X – $c_{\text{del}}(\cdot)$ units cost
- (c) Replace a character from X = $c_{\text{rep}}(\cdot)$ units cost

Example: $X = \text{mitten}$ $Y = \text{smiley}$

One transformation : mitten -> smitten -> smiten -> smilen -> smiley

Another : mitten -> sitten -> smitten -> smilten -> smiltey -> smiley

Applications of Min Edit Distance

- Text editor
- Computational Biology – Align two DNA sequences of bases to assess their similarity measures (align each letter to a letter or gap)


-AGGCTATCACCTGACCTCCAGGCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGT CGATTGCCCGAC
- Natural Language Processing and speech recognition

Minimum Edit Distance

- Use DP approach
- Define $F(i,j)$ – minimum cost of transforming $x_i \dots x_m$ to $y_j \dots y_n$, for $1 \leq i \leq m+1, 1 \leq j \leq n+1$.
- We define $F(m+1, n+1) = 0$

$F(i, n+1) = c_{\text{del}}(x_i) + c_{\text{del}}(x_{\{i+1\}}) + \dots + c_{\text{del}}(x_m)$, for $1 \leq i \leq m$

$F(m+1, j) = c_{\text{ins}}(y_j) + c_{\text{ins}}(y_{\{j+1\}}) + \dots + c_{\text{ins}}(y_n)$, for $1 \leq j \leq m+1$

- By principle of optimality, for $1 \leq i \leq m, 1 \leq j \leq n$

$$\begin{aligned} F(i,j) &= F(i+1,j+1) \text{ if } x_i = y_j \\ &= \min (c_{\text{del}}(x_i) + F(i+1,j), c_{\text{ins}}(y_j) + F(i, j+1), \\ &\quad c_{\text{rep}}(x_i, y_j) + F(i+1, j+1)) \end{aligned}$$

- Time Complexity is $O(mn)$ and space complexity $O(mn)$ if we keep track of specific edit operations.

Algorithm Design Strategies (contd.)

Matrix Chain Product

- Multiplying a $p \times q$ matrix with a $q \times r$ matrix takes $p \times q \times r$ scalar multiplications (standard matrix multiplication). The result is a $p \times r$ matrix
- Matrix multiplication is associative $\Rightarrow A \times (B \times C) = (A \times B) \times C$
- Suppose A is $d_0 \times d_1$ matrix, B is $d_1 \times d_2$ matrix and C is $d_2 \times d_3$ matrix.
- $A \times (B \times C)$ takes $d_1 d_2 d_3 + d_0 d_1 d_3$ multiplications
- $(A \times B) \times C$ takes $d_0 d_1 d_2 + d_0 d_2 d_3$ multiplications
- **Problem:**

Given a sequence of n matrices A_0, A_1, \dots, A_{n-1} where A_i is a $d_i \times d_{i+1}$ matrix, find the order of multiplication of matrices so as to minimize number of multiplications. Order can be specified by parenthesizing multiplication expression.

Matrix Chain Product

- Brute-force approach: Consider all possible orders of multiplication
- Let $N(k)$ – number of orderings of k matrices
- $N(1) = N(2) = 1$
- $N(k) = \sum_{l=1}^{k-1} N(l)N(k - l)$, $k > 2$
- $N(k)$ is called a Catalan number –
 $1, 1, 2, 5, 14, 42, 132, 429, 1430, \dots$
- Grows fast exponentially and time complexity is
 $\Omega(4^n / n^{3/2})$
- DP approach takes only $O(n^3)$ time.

DP formulation

- Consider the subsequence A_i, A_{i+1}, \dots, A_j of matrices where $i \leq j$. In any optimal solution that requires $(A_i \times A_{i+1} \times \dots \times A_j)$, optimal way to do this should occur (principle of optimality)
- Let $N_{i,j}$ be minimum number of multiplications required for multiplying these matrices
- Let $N_{i,i} = 0$
- $N_{i,j} = \min \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$ where min over all $i \leq k < j$
- Though recursive, we store $N_{i,j}$'s in a table to avoid re-computing it.
- Computing order: increasing order of $(j-i)$ from 0, ..., $n-1$

Matrix Chain Product DP complexity

- $j-i = 0 : N_{i,i} = 0, i=0, 1, 2, \dots, (n-1) - n$ computations

$j-i = 1 : N_{i,i+1} = N_{i,i} + N_{i+1,i+1} + d_i d_{i+1} d_{i+2} - (n-1)$
computations each with 2 integer multiplications and 2 integer additions and 0 min op

$j-i = 2 : N_{i,i+2} = \min \{ N_{i,i} + N_{i+1,i+2} + d_i d_{i+1} d_{i+3}, N_{i,i+1} + N_{i+2,i+3} + d_i d_{i+2} d_{i+3} \}, - (n-2)$ computations each with 2 integer multiplications and 2 integer additions and 1 min op that requires 1 comparison

$j-i = k : N_{i,i+k} -- (n-k)$ computations each with $2k$ integer multiplications and $2k$ integer additions and a min op that requires $(k-1)$ comparisons.

- **Time complexity :**

$T(n) \leq n + \sum_{k=1}^{n-1} 5k(n - k)$ is $O(n^3)$ integer operations.

Matrix chain product DP example

- A_0 (3x5), A_1 (5x6), A_2 (6x2), A_3 (2x4)

$N_{i,j}$ entries , $0 \leq i \leq j \leq 3$, Also store $k_{i,j}$ as value of $i \leq k < j$ that gives minimum value.

$$N_{0,0} = N_{1,1} = N_{2,2} = N_{3,3} = 0, k_{i,i} = i, 0 \leq i \leq 3$$

$$N_{0,1} = N_{0,0} + N_{1,1} + d_0 d_1 d_2 = 0 + 0 + 3 \times 5 \times 6 = 90; \quad k_{0,1} = 0$$

$$N_{1,2} = N_{1,1} + N_{2,2} + d_1 d_2 d_3 = 0 + 0 + 5 \times 6 \times 2 = 60; \quad k_{1,2} = 1$$

$$N_{2,3} = N_{2,2} + N_{3,3} + d_2 d_3 d_4 = 0 + 0 + 6 \times 2 \times 4 = 48; \quad k_{2,3} = 2$$

$$\begin{aligned} N_{0,2} &= \min(N_{0,0} + N_{1,2} + d_0 d_1 d_3, N_{0,1} + N_{2,2} + d_0 d_2 d_3) \\ &= \min(0 + 60 + 3 \times 5 \times 2, 90 + 0 + 3 \times 6 \times 2) = 90; \quad k_{0,2} = 0 \end{aligned}$$

$$\begin{aligned} N_{1,3} &= \min(N_{1,1} + N_{2,3} + d_1 d_2 d_4, N_{1,2} + N_{3,3} + d_1 d_3 d_4) \\ &= \min(0 + 48 + 5 \times 6 \times 4, 60 + 0 + 5 \times 2 \times 4) = 100; \quad k_{1,3} = 2 \end{aligned}$$

$$\begin{aligned} N_{0,3} &= \min(N_{0,0} + N_{1,3} + d_0 d_1 d_4, N_{0,1} + N_{2,3} + d_0 d_2 d_4, \\ &\quad N_{0,2} + N_{3,3} + d_0 d_3 d_4) \\ &= \min(0 + 100 + 3 \times 5 \times 4, 90 + 48 + 3 \times 6 \times 4, 90 + 0 + 3 \times 2 \times 4) = 114 \\ k_{0,3} &= 2 \end{aligned}$$

Matrix chain product DP example (contd.)

Final answer : $N_{0,3} = 114$

Optimal order computed as : $k_{0,3} = 2$ i.e $(A_0 \times A_1 \times A_2) \times A_3 \rightarrow$ look at entry (0,2) to compute $(A_0 \times A_1 \times A_2)$

$k_{0,2} = 0$ i.e. $(A_0 \times A_1 \times A_2)$ computed as $A_0 \times (A_1 \times A_2) \rightarrow$ look at entry (1,2) to compute $(A_1 \times A_2)$

$k_{1,2} = 1$ i.e. $A_1 \times A_2$ computed as $(A_1 \times A_2)$. Final order : $(A_0 \times (A_1 \times A_2)) \times A_3$

	j=0	j=1	j=2	j=3
i=0	0 ($k_{0,0} = 0$)	90 ($k_{0,1} = 0$)	90 ($k_{0,2} = 0$)	114 ($k_{0,3} = 2$)
i=1	x	0 ($k_{1,1} = 1$)	60 ($k_{1,2} = 1$)	100 ($k_{1,3} = 2$)
i=2	x	x	0 ($k_{2,2} = 2$)	48 ($k_{2,3} = 2$)
i=3	x	x	x	0 ($k_{3,3} = 3$)

Greedy Algorithms

- An optimization problem involves finding a solution that minimizes or maximizes an objective function of decision variables with or without constraints
- A global optimal choice strategy finds best solution among all possible solutions to the problem.
- A local partial solution strategy finds best partial solution among a limited set of solutions.
- A problem satisfies **greedy-choice** property if a global optimal solution can be reached by a sequence of local partial solution choices starting from a **well-defined state**.
- Otherwise this strategy provides only a heuristic and optimal solution is not guaranteed.
- Well-defined starting state may require some pre-processing.

Optimal substructure

- Recall a problem exhibits “optimal substructure” property if optimal solution contains within itself optimal solutions to sub problems.
- This is the basis of DP wherein we split the optimization function as a sequence of recursive functions ; they depend only on the “state” we arrive at by choice of values for a subset (typically of size 1) of decision variables.
- Also called “principle of optimality” by Richard Bellman.
- Optimal substructure is a necessary condition for greedy algorithms.

Coin change problem

Given :

Unlimited limited supply of n coins, each of which has value $\{c_1, c_2, \dots, c_n\}$,

Required:

Minimum number of coins to make change for an amount V. Assume $c_n = 1$

- Satisfies “optimal substructure” property – if in an optimal solution we make change for value “v” out of “V”, then it contains optimal solution to sub problem of making change for $V-v$ regardless of how we made change for “v”
- Can use a DP formulation for it

Coin change – DP formulation

- Define $F(i, v)$ – min number of coins needed to make change for v from set $\{c_i, c_{i+1}, \dots, c_n\}$, $1 \leq i \leq n$, $1 \leq v \leq V$
 $N(i, v)$ – number of coins of type c_i used in this solution
- Boundary conditions : $F(n+1, v) = N(n + 1, v) = \infty$, if $v > 0$, else $F(n+1, v) = N(n + 1, v) = 0$
- $$F(i, v) = \min_{0 \leq j \leq \lfloor \frac{v}{c_i} \rfloor} j + F(i + 1, v - j * c_i), \quad 0 \leq v \leq V,$$

$$1 \leq i \leq n-1$$

$$N(i, v) = \operatorname{argmin} F(i, v)$$
- Need $F(1, V)$ and $N(1, V)$
- Time complexity of DP algorithm : $O(nV)$

Coin change - greedy solution

- Assume $c_1 \geq c_2 \dots \geq c_n$
- At each stage i , only one choice considered for making change for v
- $F(i, v) = \lfloor \frac{v}{c_i} \rfloor + F(i + 1, v - \lfloor \frac{v}{c_i} \rfloor * c_i)$ and $N(i, v) = \lfloor \frac{v}{c_i} \rfloor$
- Iterative greedy algorithm:
 $v \leftarrow V$
 $F \leftarrow 0$
for $i \leftarrow 1$ to n
 $N(i) \leftarrow \lfloor \frac{v}{c_i} \rfloor$
 $F \leftarrow F + N(i)$
 $v \leftarrow v \bmod c_i$
- Time complexity : $O(n)$

Coin change – greedy choice

- Does it have “greedy choice” property ?
- Not always. Consider only quarters, dimes and pennies (i.e. $c_1=25$, $c_2=10$ and $c_3=1$) and $V = 30$
 - Greedy choice will give $N(1) =1$, $N(2)=0$, $N(3)=5$, $F=6$
 - Is it optimal ?
 - No. $N(1) =0$, $N(2)=3$, $N(3)=0$, $F = 3$
- But for quarters, dimes, nickels and pennies (i.e. $c_1=25$, $c_2=10$, $c_3=5$ and $c_4=1$) , it satisfies “greedy choice property”
 - Ignore pennies, remaining value divisible by 5
 - Any state $F(2, v)$ with $v \geq 25$ not part of optimal (as you can reduce number of coins either by replacing 2 dimes+1 nickel or 3 dimes by a quarter and nickel)
 - Any state $F(3, v)$ with $v \geq 10$ not part of optimal as you can reduce number of coins by replacing 2 nickels by a dime

Algorithm Design Strategies (contd.)

Fractional knapsack problem

- **Problem :** $\text{Max } \sum_{i=1}^n b_i t_i$
s.t. $\sum_{i=1}^n w_i t_i \leq W, \quad 0 \leq t_i \leq 1, \forall i$
- Select items (may be partially) with weights w_i 's and values b_i 's so as to fill a knapsack of weight W .
- In 0-1 knapsack problem, an item cannot partially fill a knapsack. It is a harder problem to solve.
- Define relative value $v_i = b_i/w_i, \forall i$. An item with a larger value of b_i and smaller value of w_i is relatively more valuable.

Let $w_i t_i = x_i, \forall i$.

- **Problem :** $\text{Max } \sum_{i=1}^n v_i x_i$
s.t. $\sum_{i=1}^n x_i \leq W, \quad 0 \leq x_i \leq w_i, \forall i$

Fractional knapsack problem

- Satisfies “optimal substructure” property. Why?
- If in an optimal solution to this problem, after selecting a few items capacity of v remains then optimal solution to a knapsack problem with capacity v must be part of the complete optimal solution.
- **Greedy approach:**
Let items be ordered such that $v_1 \geq v_2 \dots \geq v_n$ and they are filled in that order; for last item if weight exceeds remaining capacity, use partial amount. This is a greedy approach.
 - At most one item will be partially filled in this approach.
 - Does this algorithm satisfy greedy choice property ?

Example of fractional knapsack

- $b_1 = 7, b_2 = 5, b_3 = 4, b_4 = 3$
- $w_1 = 4, w_2 = 3, w_3 = 2, w_4 = 1, W = 6$

Strategy 1: Fill by least weight to highest weight

Choose w_4, w_3, w_2 – total value = $3+4+5 = 12$

Strategy 2: Fill by largest value to smallest value

Choose $w_1, w_2 * 2/3$ – total value = $7 + 5 * 2/3 = 10.33$

(Optimal) Strategy 3 : Fill by largest relative value to smallest

$$v_1 = 7/4 = 1.75, v_2 = 5/3 = 1.67, v_3 = 4/2 = 2$$

$$v_4 = 3/1 = 3$$

Choose $w_4, w_3, \frac{3}{4} * w_1$

$$\text{Total value} = 3 + 4 + \frac{3}{4} * 7 = 12.25$$

Fractional knapsack greedy alg.

- **Optimality of greedy choice:**

We prove by contradiction.

Let there be an optimal solution such that for two items such that $v_i > v_j$, we do not fully fill item i but use some amount of item j thereby not using the greedy choice.

i.e. $x_i < w_i$ and $x_j > 0$.

We can then replace amount of j as much as possible by an equal amount of item i. This amount = $\min(x_j, w_i - x_i)$.

Additional value obtained

= $(v_i - v_j) \min(x_j, w_i - x_i) > 0$ violating optimality of solution.

- Time-complexity – $O(n \log n)$
- Fast considering there are 2^n possible subsets of n items.

Meeting scheduling problem

Problem: Given a set S of n meetings, each with start and finish times s_i and f_i respectively for $1 \leq i \leq n$, find a mapping $\phi : S \rightarrow \{1, 2, \dots, M\}$ (M conf. rooms) such that

- (a) For two meetings m_i and m_j s.t. $\phi(m_i) = \phi(m_j)$ (assigned to same room), either $f_i \leq s_j$ or $f_j \leq s_i$ (they do not conflict in time)
- (b) M should be as small as possible (minimum number of rooms).

- **Greedy approach:**

- (a) Sort meetings according to start times s_i 's. Start with a single room.
- (b) For each meeting in sequence
 - (i) check if it does not conflict with any of the meetings scheduled so far, schedule it at earliest opportunity in a room.
 - (ii) Else schedule it in a new meeting room.

Meeting scheduling problem

- **Proof of correctness:**

We prove by contradiction.

Suppose the optimal solution requires $m \leq k - 1$ meeting rooms while the greedy algorithm requires k rooms.

Let m_i be the first meeting scheduled in last room k by greedy approach.

- ⇒ m_i conflicts with at least one meeting scheduled in each of the rooms $1..k-1$
- ⇒ all these meetings have start times not later than s_i but have finish times later than s_i . These meetings conflict with each other.
- ⇒ At least k meetings conflict with each other, a contradiction as the algorithm considers non-conflicting schedule and this cannot be done with $< k$ rooms

- **Time complexity:**

$O(n \log n)$ time for pre-processing.

In each scheduling step, how do we check efficiently if the new meeting does not conflict with previous meetings ?

Keep track of earliest finish time among the tasks that start latest in each room and the rooms associated with the tasks. If the next task's start time is later than this finish time, schedule it in that room. Else it cannot be scheduled in any of the rooms used so far and has to be scheduled in a new room. By keeping the earliest finish times of latest tasks in each room in a heap, we can do this in $O(\log n)$ time in each step.

Example of meeting scheduling

- $m_1 : 10(s_1)-14(f_1)$, $m_2 : 9-11$, $m_3 : 8-10$, $m_4 : 7-12$, $m_5 : 10-15$, $m_6 : 13-15$
- Meetings sorted by start times: m_4 , m_3 , m_2 , m_1 , m_5 , m_6
- Let f_k be earliest finish time of latest task scheduled in a room and r_k , the corresponding room after step k (i.e. k meetings have been scheduled).

$R_1 : m_4 \text{ (7-12)}$

$R_2 : m_3 \text{ (8-10)}, m_1 \text{ (10-14)}$

$R_3 : m_2 \text{ (9-11)}, m_6 \text{ (13-15)}$

$R_4 : m_5 \text{ (10-15)}$

$f_1 = 12, f_2 = 10, f_3 = 10, f_4 = 11, f_5 = 11, f_6 = 12$

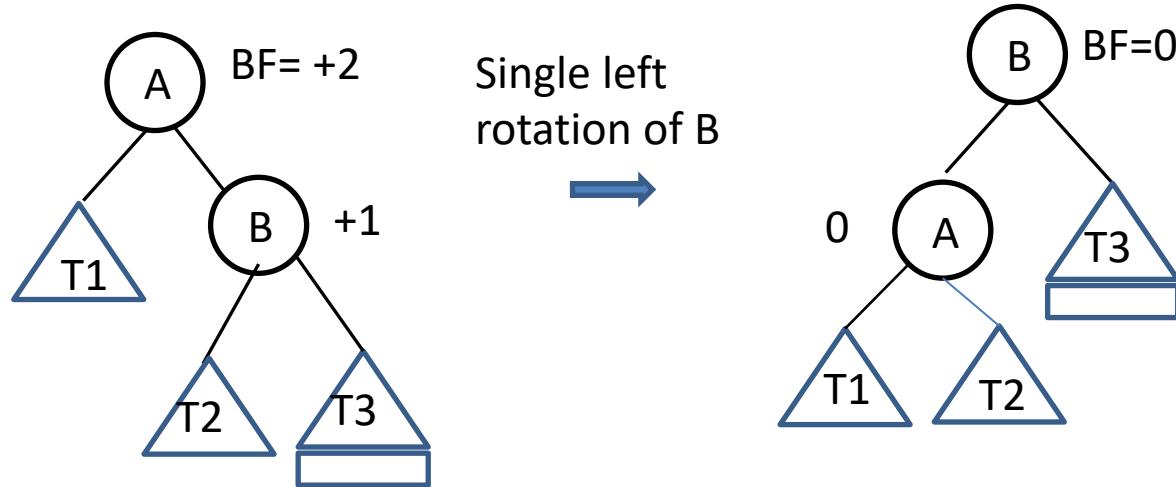
$r_1 = 1, r_2 = 2, r_3 = 2, r_4 = 3, r_5 = 3, r_6 = 1$

AVL tree insertion examples

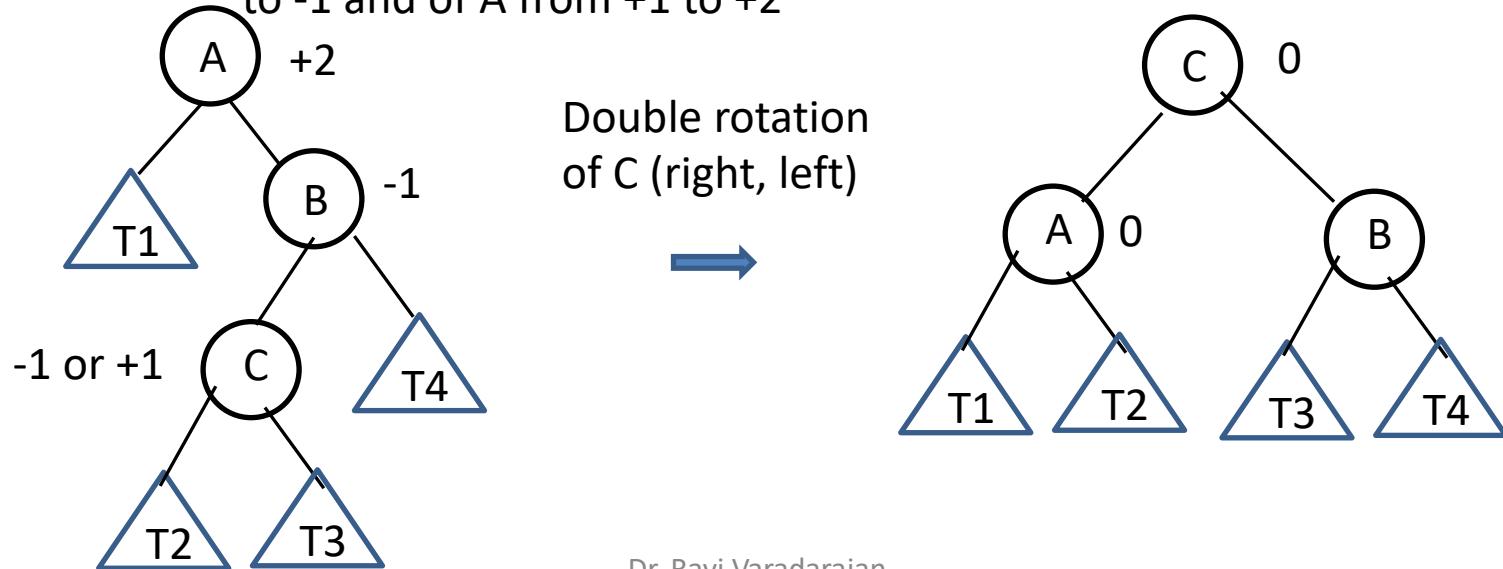
$O(\log n)$ AVL tree insertion

- May cause balance factor at an ancestor node of inserted node to change to -2 or +2.
- Fixing it requires only one “rotation” operation which takes $O(1)$ time as it requires only pointer changes
- 4 cases:
 - (a) Node A’s BF changes from +1 to +2, its right child node B’s BF changes from 0 to +1
Left rotate B to move its right subtree up one level
 - (b) Node A’s BF changes from +1 to +2, its right child node B’s BF changes from 0 to -1 as its left child C’s BF changes from 0 to +1 or -1
Double rotate C (right followed by left) to make A and B its children
 - (c) & (d) are “mirror” cases of (a) and (b)

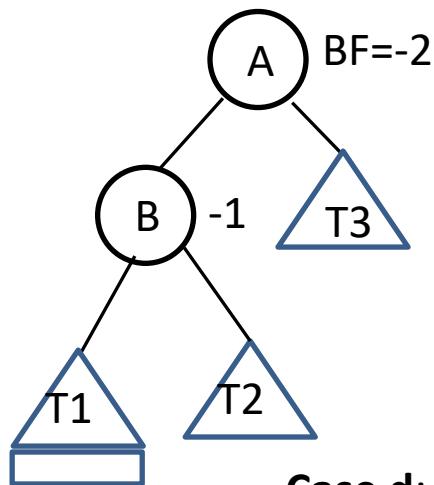
Case a: Insertion into T3 changes BF of B from 0 to +1 and of A from +1 to +2



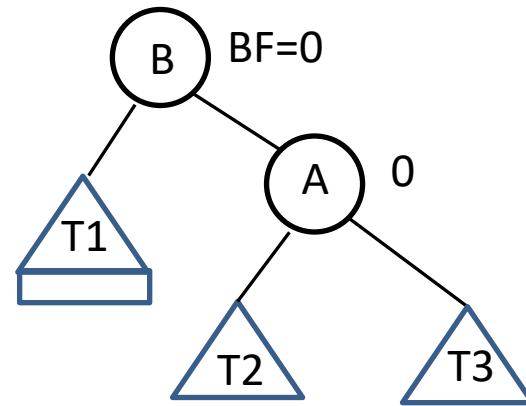
Case b: Insertion into T2 or T3 changes BF of B from 0 to -1 and of A from +1 to +2



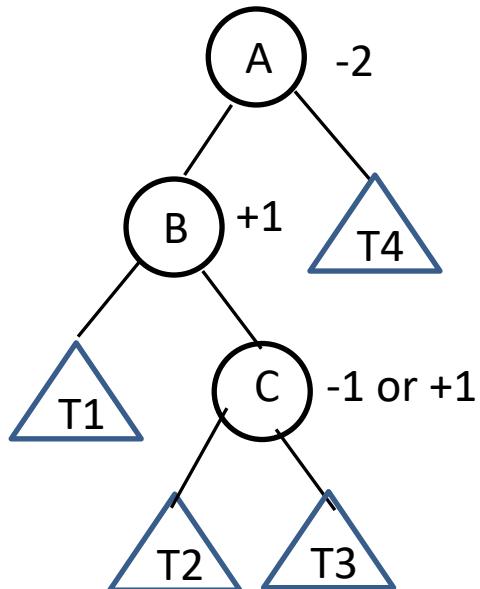
Case c: Insertion into T1 changes BF of B from 0 to -1
and of A from -1 to -2



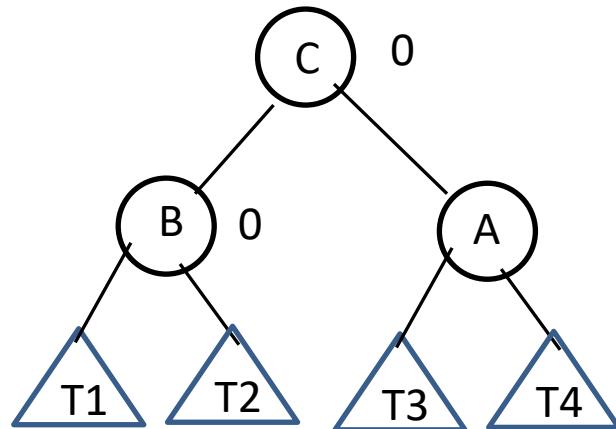
Single right rotation of B



Case d: Insertion into T2 or T3 changes BF of B from 0 to +1 and of A from -1 to -2



Double rotation
of C (left, right)

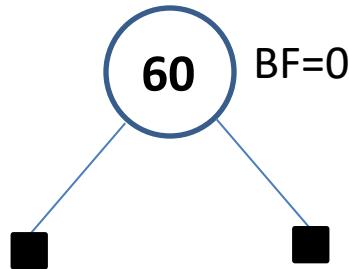


AVL Insertion examples

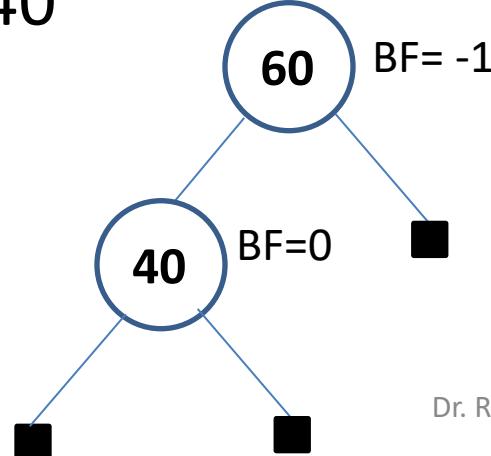
- Balance factor at a node = Height of right subtree – height of left subtree rooted at that node
- Start with empty tree (single external node) :



- Insert 60

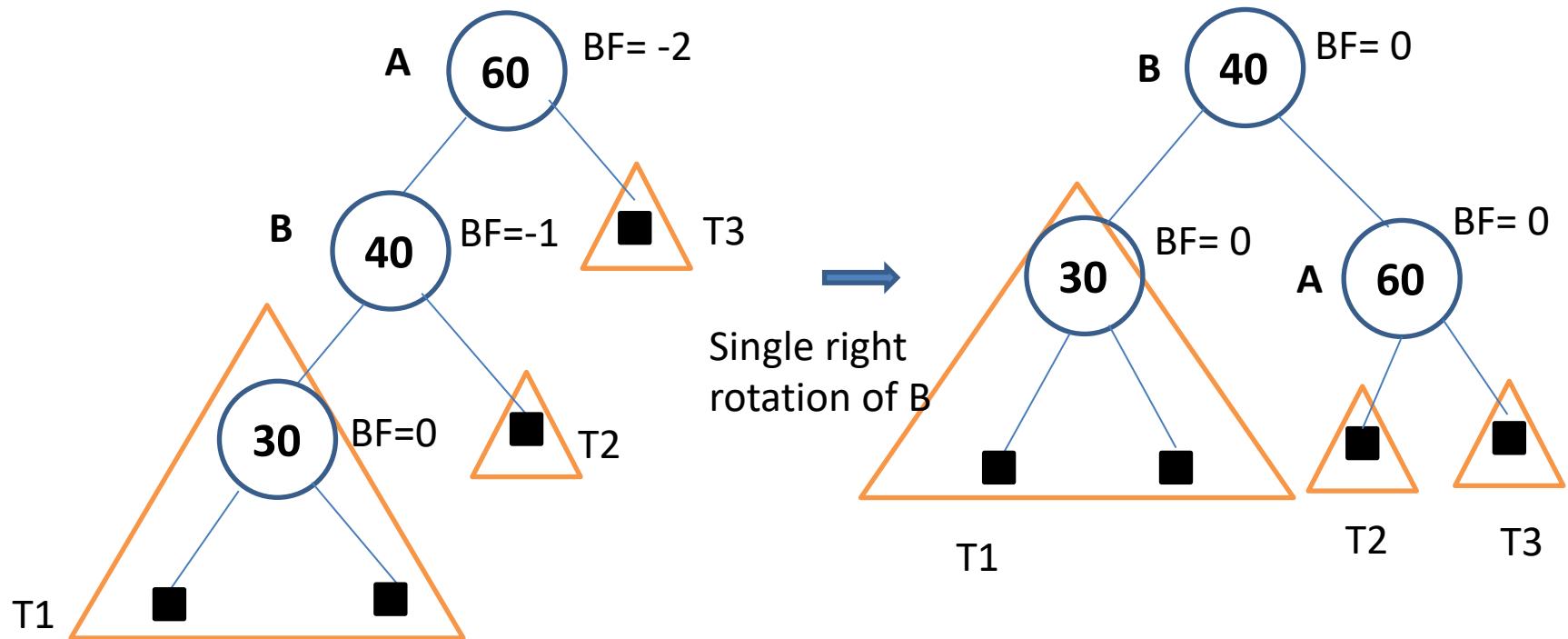


- Insert 40



AVL Insertion examples (contd.)

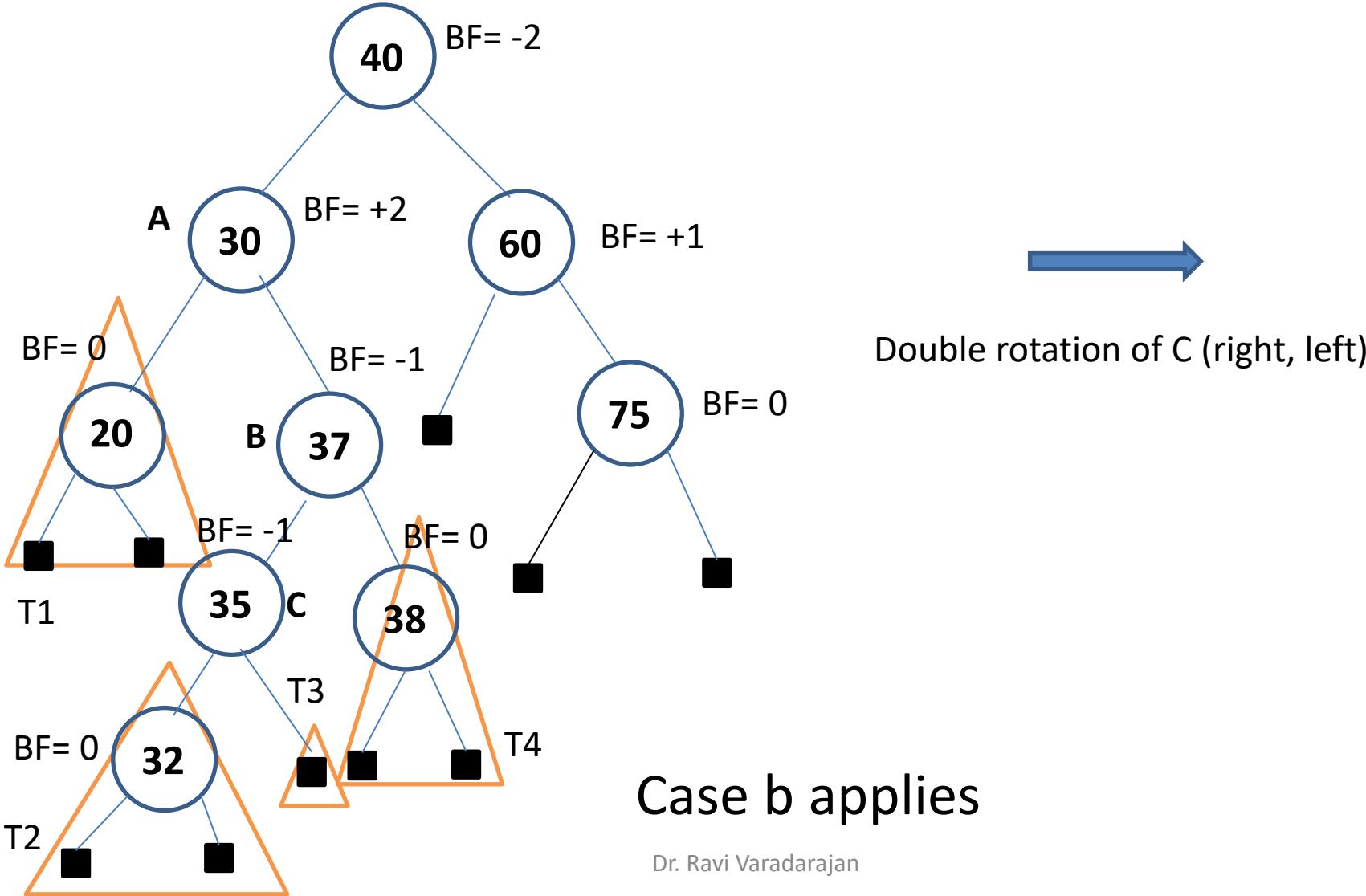
- Insert 30



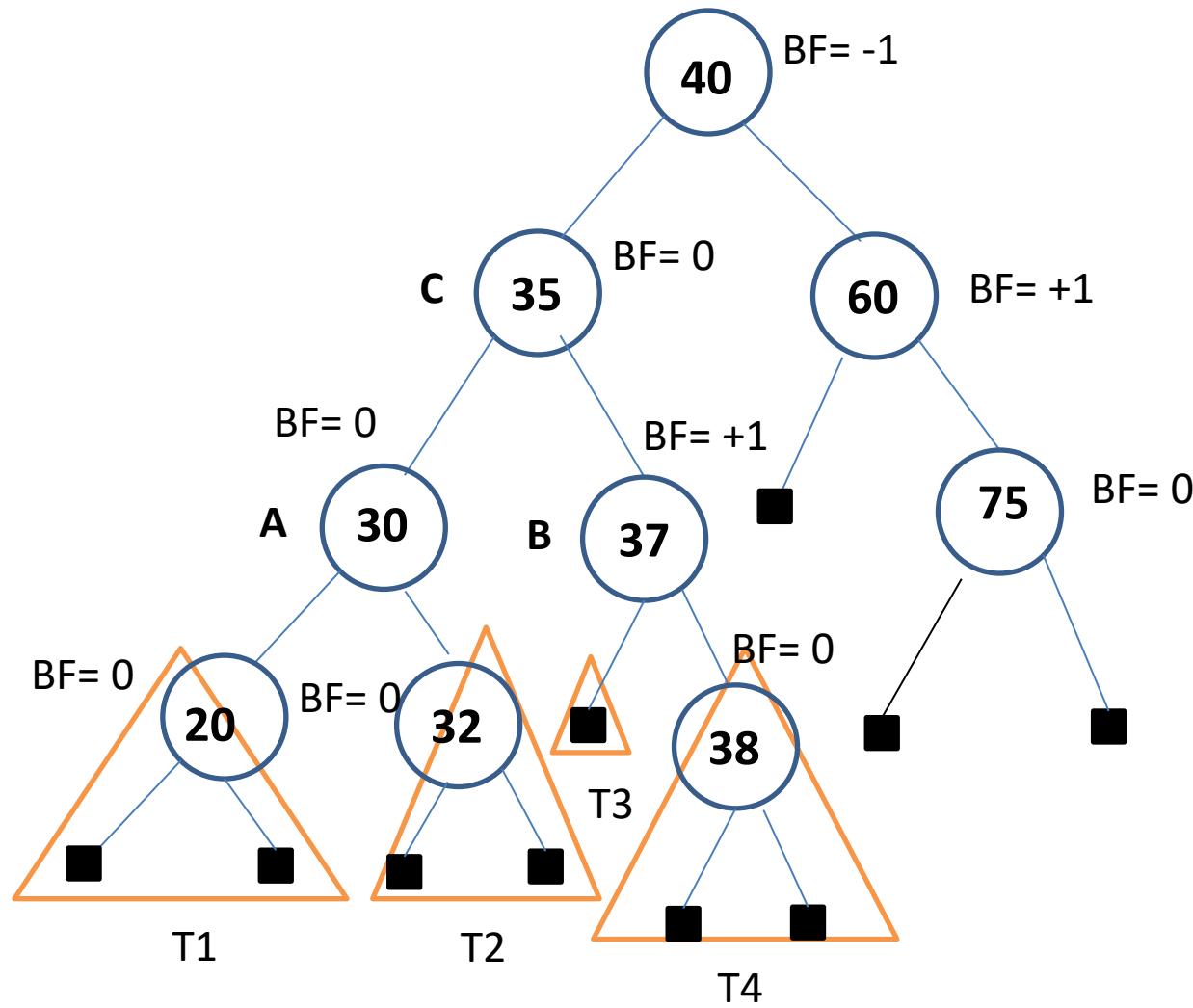
Case c applies

AVL Insertion examples (contd.)

- After inserting 75, 37, 20, 35, 38, when 32 inserted

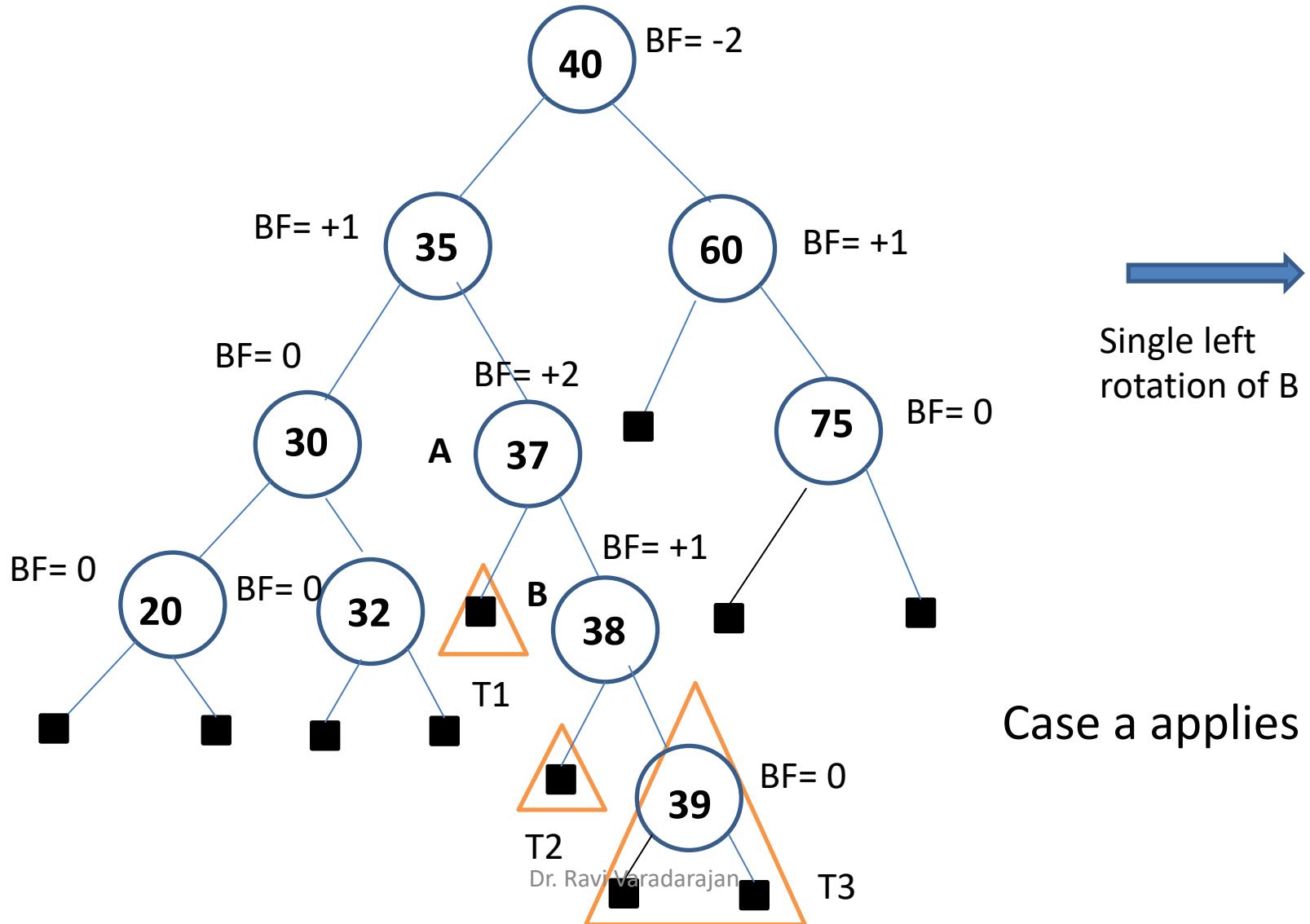


AVL Insertion examples (contd.)

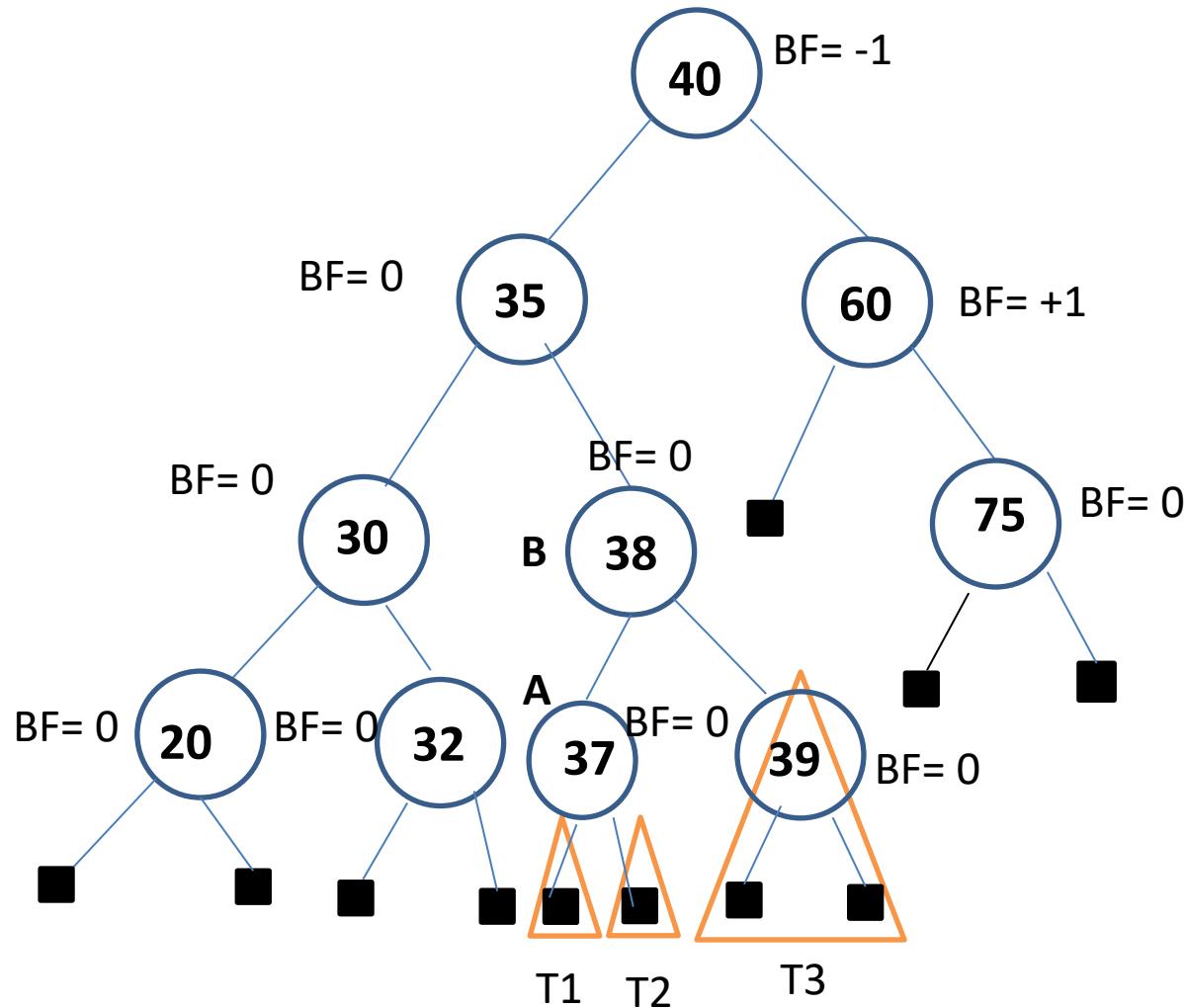


AVL Insertion examples (contd.)

- Insert 39

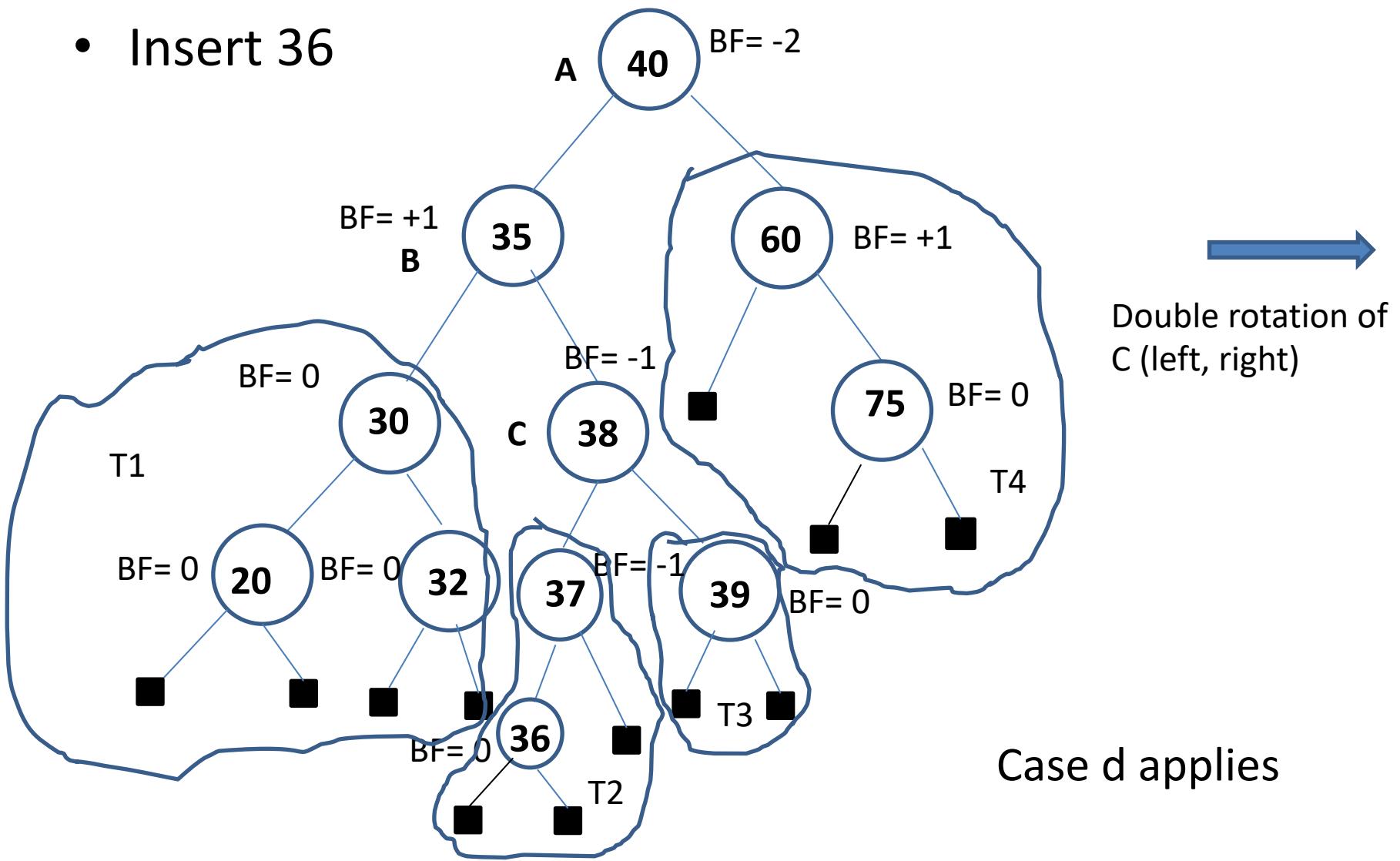


AVL insertion examples (contd.)

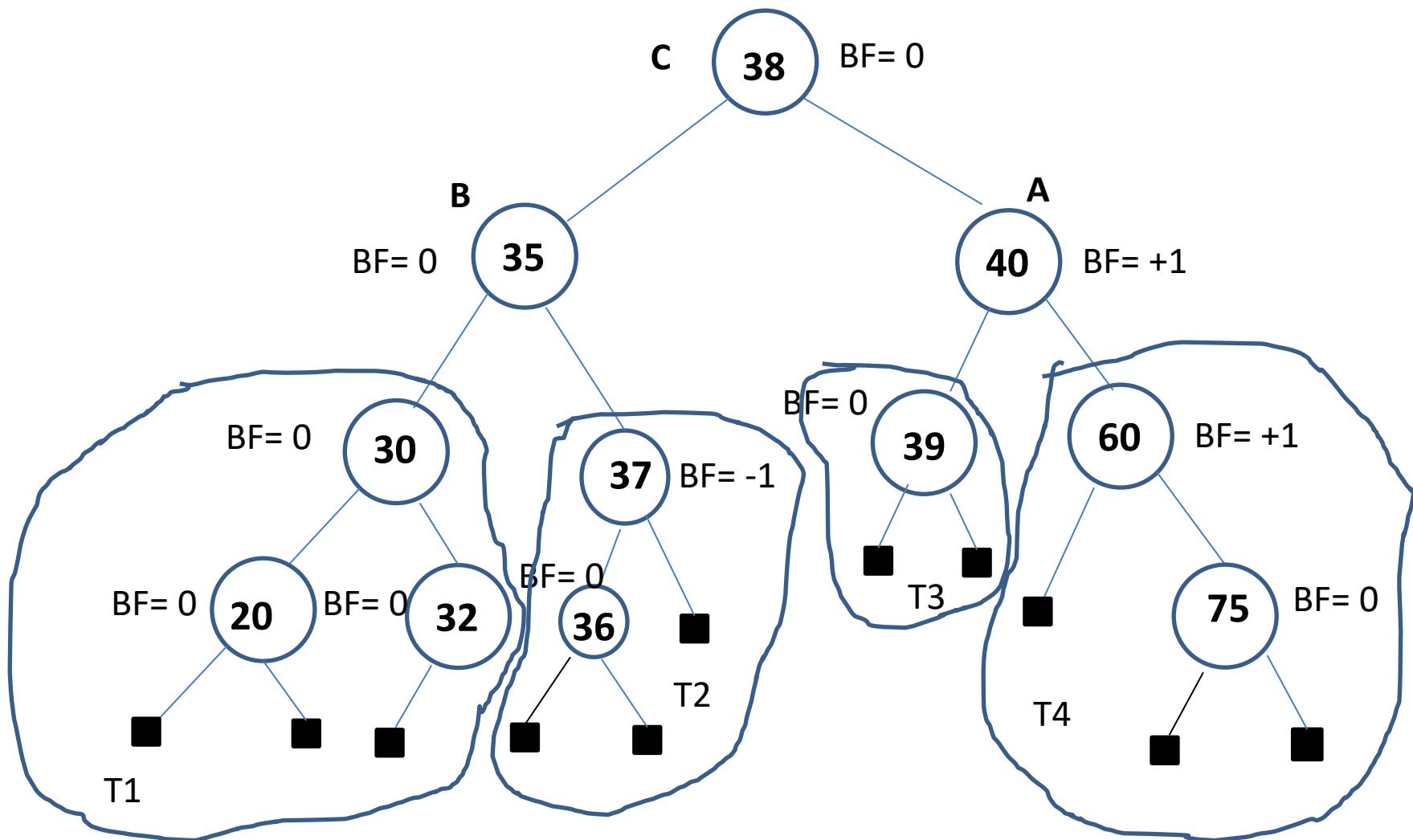


AVL insertion examples (contd.)

- Insert 36



AVL insertion examples (contd.)



Search Trees

Ordered Dictionary ADT

- Keys are assumed to be from a totally ordered set
- Additional operations:
 - closestKeyBefore(k) – return largest key smaller than k
 - closestKeyAfter(k) – return smallest key greater than k
- Implementation choices:
 - (a) Sorted array of keys
 - insert and remove operations take $O(n)$ time in worst case
 - findElement(k) – use binary search that takes $O(\log n)$ time
 - closestKeyBefore, closestKeyAfter take $O(\log n)$ time.

Binary search tree

- Binary tree implementation with keys stored only in internal nodes.
- Have ordering property:

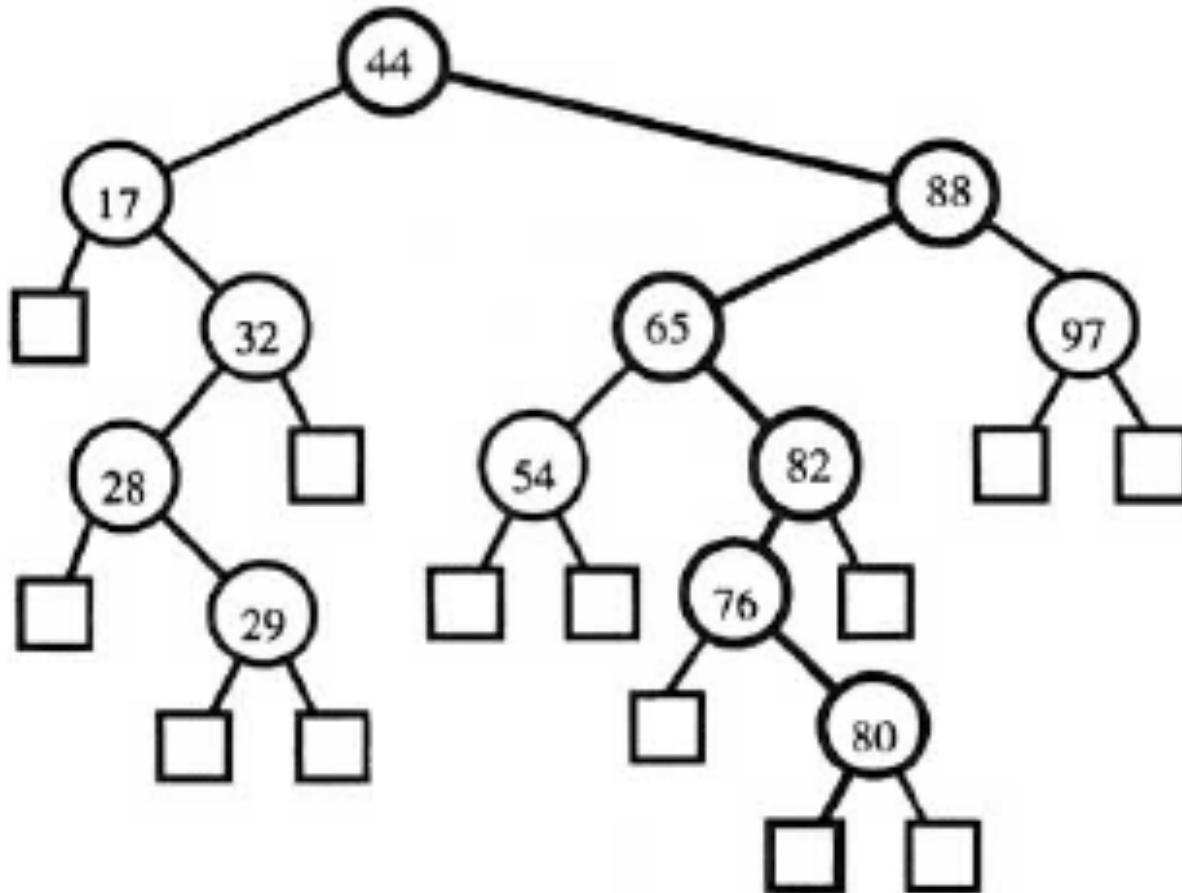
For any subtree rooted at internal node v, all keys in left subtree of node have items smaller than the item at v and all keys in right subtree of node have items larger than the item at v

- Recursive structure
- Binary search tree time complexities:
 - findElement, insertElement, removeElement – $O(h)$
 - findClosestBefore, findClosestAfter – $O(h)$ time

Where h is height of the tree which can be “n” in the worst case n being number of keys in the tree.

- Balanced binary search trees keep height as $O(\log n)$ in the worst-case while ensuring the same order of complexity for the different operations.

BST example



Binary Search Tree

insertElement(v,k,e):

Input : v root node of binary search tree, k key and e element

Output: node where key is inserted or exists

if isExternal(v)

 make v internal with key k and element e with 2 external children
nodes

 return v

if k = key(v)

 elem(v) \leftarrow e; return v

if k < key(v)

 childNode \leftarrow leftChild(v)

else

 childNode \leftarrow rightChild(v)

return insertElement(childNode,k,e)

insertElement(T,k,e):

 return insertElement(T.root,k,e)

Search Trees

Binary Search Tree

insertElement(v,k,e):

Input : v root node of binary search tree, k key and e element

Output: node where key is inserted or exists

if isExternal(v)

 make v internal with key k and element e with 2 external children
nodes

 return v

if k = key(v)

 elem(v) \leftarrow e; return v

if k < key(v)

 childNode \leftarrow leftChild(v)

else

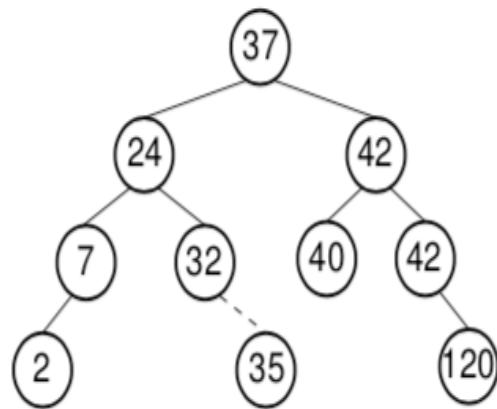
 childNode \leftarrow rightChild(v)

return insertElement(childNode,k,e)

insertElement(T,k,e):

 return insertElement(T.root,k,e)

BST key insertion example



Binary Search Tree (contd.)

removeElement(v,k,e):

Input : v root node of binary search tree, k key and e element

Output: element with key k if it exists or error

if isExternal(v)

 return "NO SUCH KEY" error

if k < key(v)

 return removeElement(leftChild(v),k,e)

else if k > key(v)

 return removeElement(rightChild(v),k,e)

else

 elem ← elem(v)

 if v has both external child nodes

 make v external node removing key and elem

 return elem

 else if isExternal(leftChild(v))

 key(v) ← key(rightChild(v))

 elem(v) ← elem(rightChild(v))

 leftChild(v) ← leftChild(rightChild(v))

 rightChild(v) ← rightChild(rightChild(v))

 return elem

 else if isExternal(rightChild(v))

 // do similar to previous case with rightChild(v) replaced by leftChild(v) in rhs of assignment statements

Binary Search Tree (contd.)

else

 successor \leftarrow inOrderSuccessor(T, k)

 key(v) \leftarrow key(successor)

 elem(v) \leftarrow elem(successor)

 removeElement(successor, key(successor),
 elem(successor))

Time Complexity:

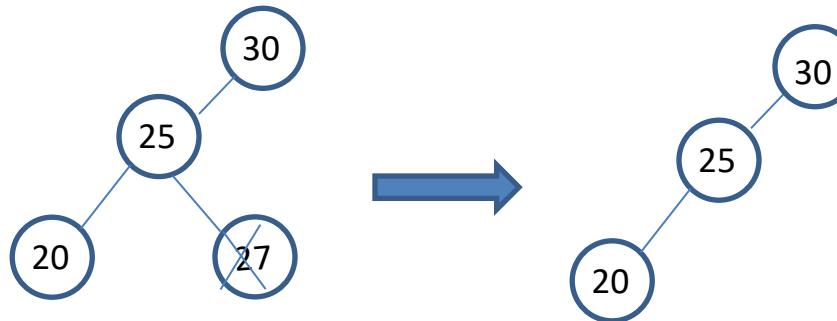
$O(h)$ to find node to remove, $O(h)$ to find in order successor,
 $O(1)$ to remove successor node

 Total time complexity : $O(h)$

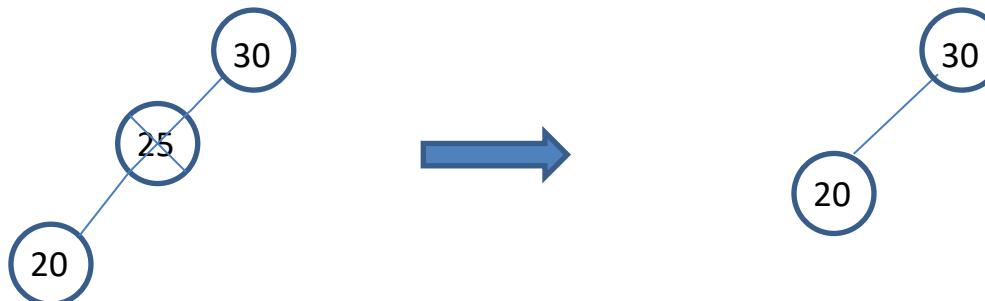
 Total additional space complexity : $O(h)$

Dictionary operations for BST - delete

(a) If node to be removed is a leaf, we can just remove it from the tree

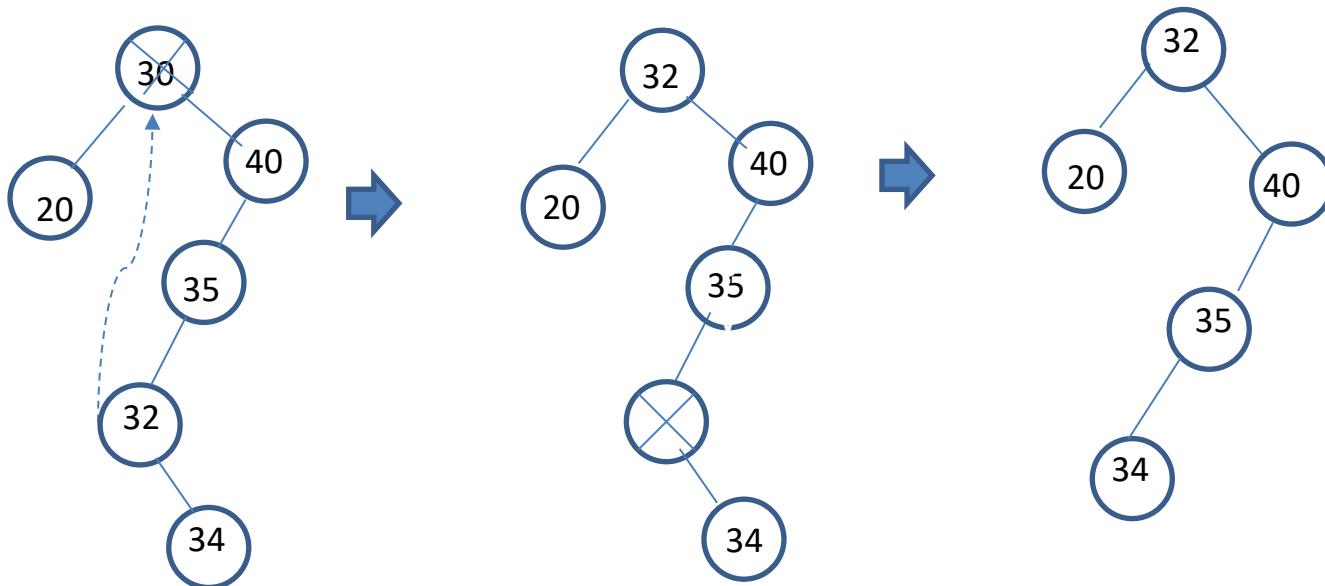


(b) If a node has only one child, then make the child the child of its parent



Dictionary operations for BST - delete

(c) when node to be removed has 2 children (replace it with in-order successor or minimum key in right subtree i.e. left most node)



Build a BST

- To build a binary search tree with n keys
- If we randomly choose a sequence a_1, a_2, \dots, a_n from the given keys and build using `insertElement()` n times, worst-case complexity is $O(n^2)$ as tree may be skewed and height can be $O(j)$ for a j -element tree
- But average time complexity can be shown to be $O(n \log n)$
- Also using DP we can build a optimal binary search tree for a set of n keys with p_i probability of executing a `find()` with key equal to a_i and q_i probability of executing a `find()` with $a_i < \text{key} < a_{i+1}$ and q_0 probability of executing a `find()` with $\text{key} < a_1$

Balanced Binary Search Trees

- Satisfies some **height balancing property** at every node of the tree
- Recursive structure
- Height balancing property typically guarantees logarithmic bounds on tree height
- Insertions and removals restructure trees to guarantee this property with minimum overhead
- They involve **rotation** operations.

AVL tree

- Due to inventors Adelson-Velskii and Landis
- Balance factor at a node = Height of right subtree – height of left subtree rooted at that node
- Binary search tree with height-balancing property : balance factor at each node is 0,-1 or 1
- Keys stored only in internal nodes
- Note # of external nodes = # internal nodes + 1
- An AVL tree of height “h” has at least $n(h)$ nodes, $n(0) = 0$, $n(1) = 1$ and $n(h) = 1 + n(h-1)+n(h-2)$, $h \geq 2$. Recognize $n(h)$?

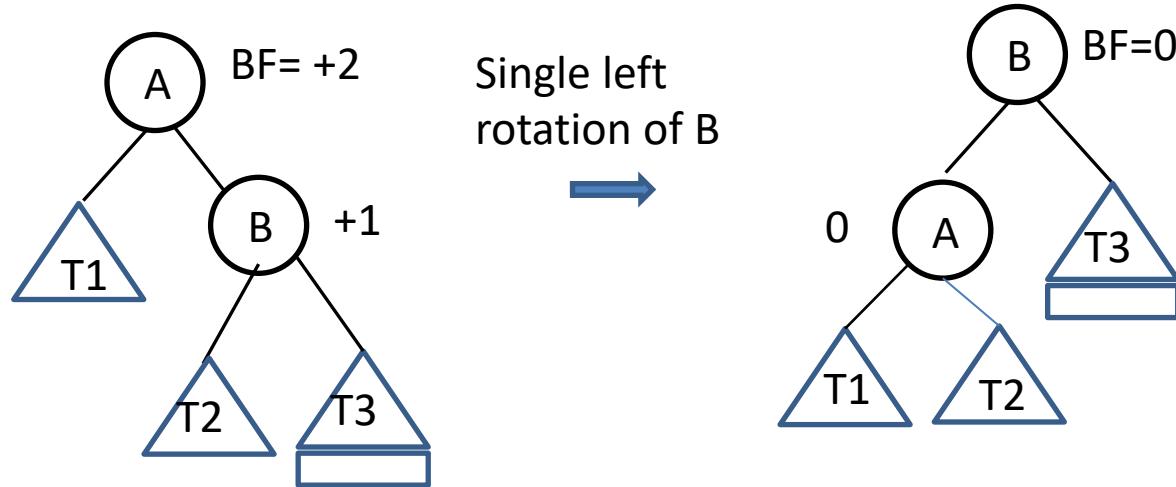
Like Fibonacci sequence

- It has at most $m(h)$ nodes, $m(h) = 1 + 2 * m(h-1)$, $h \geq 2$ and $m(0) = 0$
- Can show $n(h) > 2^{\frac{h}{2}-1} \rightarrow h < 2 \log n + 2 \rightarrow h$ is $O(\log n)$
- Also we see that $m(h) = 2^h - 1 \rightarrow n < 2^h \rightarrow h > \log n \rightarrow h$ is $\Omega(\log n)$
- FindElement() takes $O(\log n)$ time.

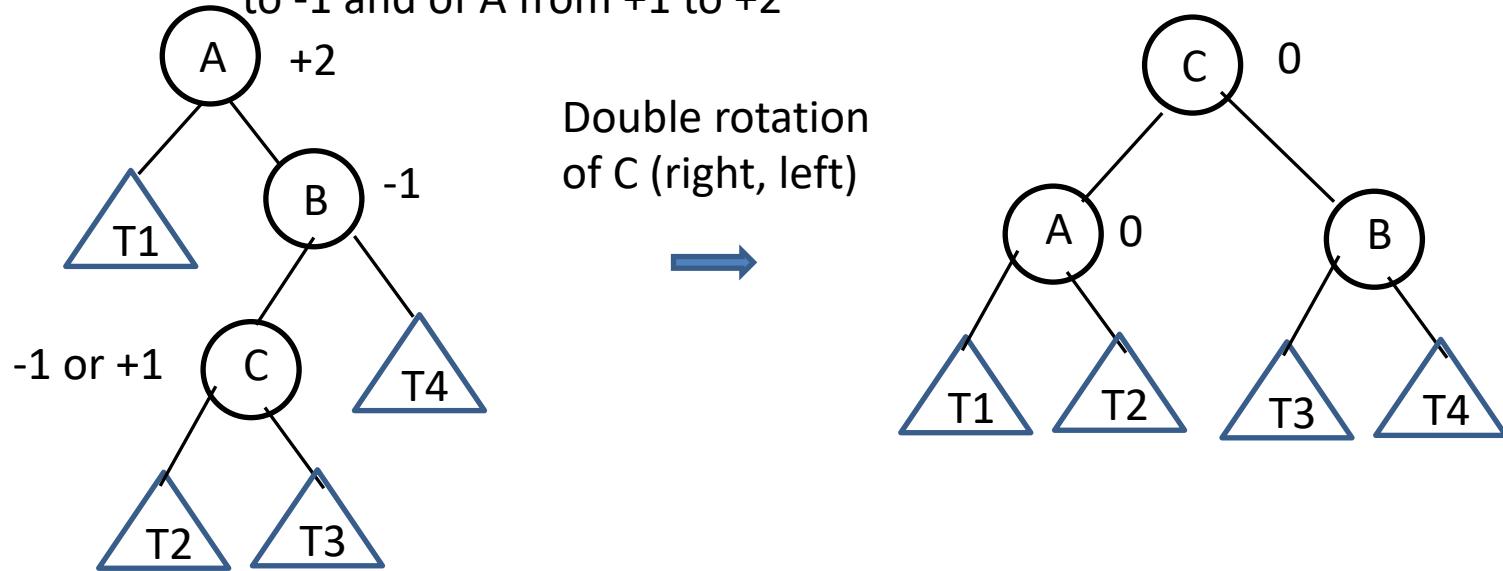
$O(\log n)$ AVL tree insertion

- May cause balance factor at an ancestor node of inserted node to change to -2 or +2.
- Fixing it requires only one “rotation” operation which takes $O(1)$ time as it requires only pointer changes
- 4 cases:
 - (a) Node A’s BF changes from +1 to +2, its right child node B’s BF changes from 0 to +1
Left rotate B to move its right subtree up one level
 - (b) Node A’s BF changes from +1 to +2, its right child node B’s BF changes from 0 to -1 as its left child C’s BF changes from 0 to +1 or -1
Double rotate C (right followed by left) to make A and B its children
 - (c) & (d) are “mirror” cases of (a) and (b)

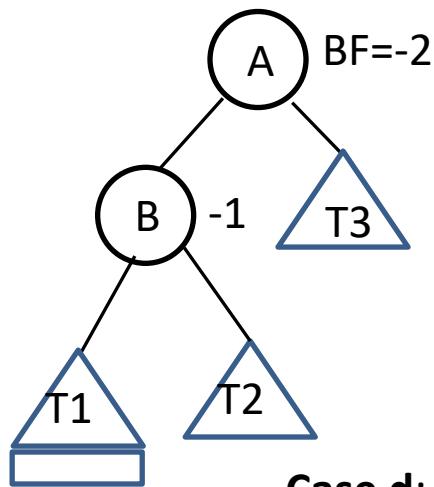
Case a: Insertion into T3 changes BF of B from 0 to +1 and of A from +1 to +2



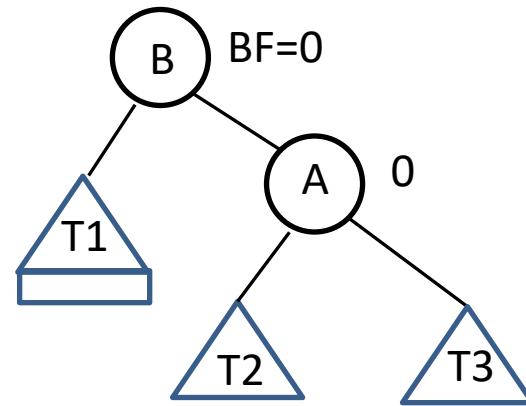
Case b: Insertion into T2 or T3 changes BF of B from 0 to -1 and of A from +1 to +2



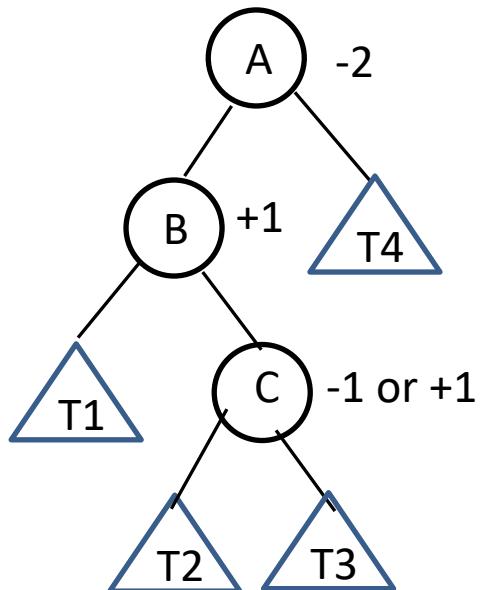
Case c: Insertion into T1 changes BF of B from 0 to -1
and of A from -1 to -2



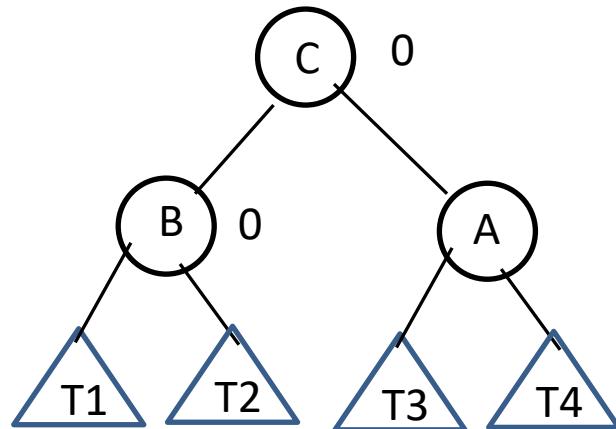
Single right rotation of B



Case d: Insertion into T2 or T3 changes BF of B from 0 to +1 and of A from -1 to -2



Double rotation
of C (left, right)



Search Trees (contd.)

$O(\log n)$ AVL tree deletion

- First do deletion as in ordinary binary search tree (discussed before)
- This may cause BF of an ancestor node to change to +2 or -2.
- Use “rotation” operations as in insertion to balance the tree
- Can be shown to be $O(\log n)$ in the worst-case.

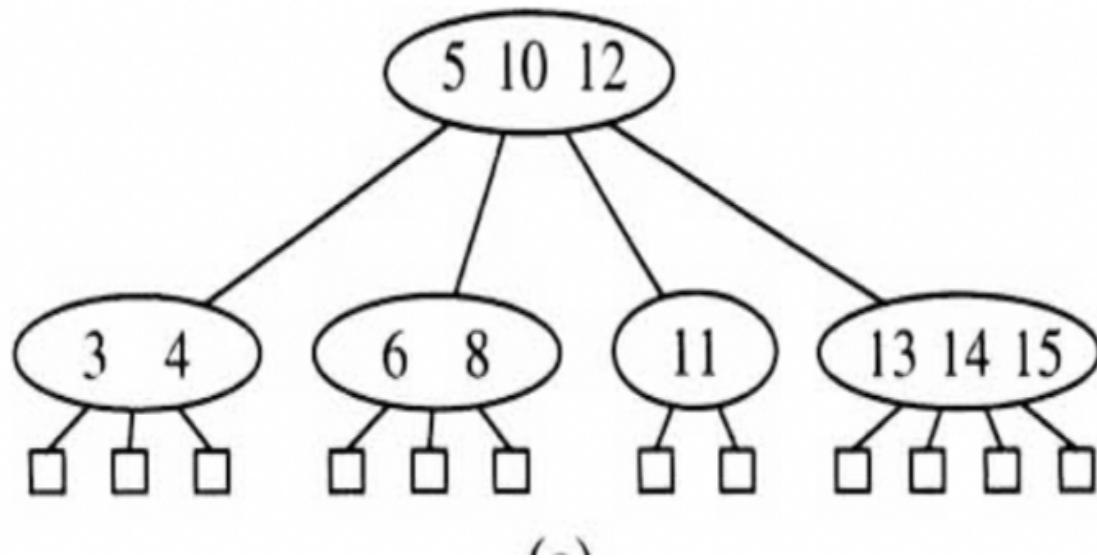
Multi-way search trees

- Stores more than one key in an internal node and if a node has at most “m” keys $k_1 \ k_2 \dots k_m$ then it has $m+1$ children
- Any key k in subtree of i-th child ($1 \leq i \leq m+1$) satisfying $k_{i-1} < k < k_i$ assuming $k_0 = -\infty$ and $k_{m+1} = \infty$
- Examples : 2-4 tree ($m=3$), B-tree (m depends on disk block size)
- Maintains height balance property by having every external node at the same height.
- Height of tree is $\Theta(\log n)$ where n is number of keys.

2-4 tree

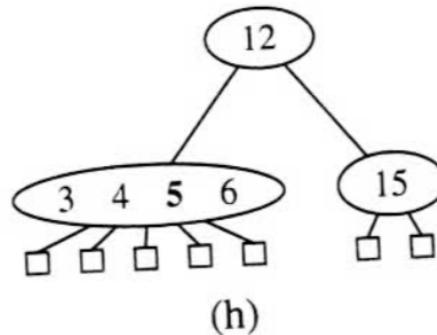
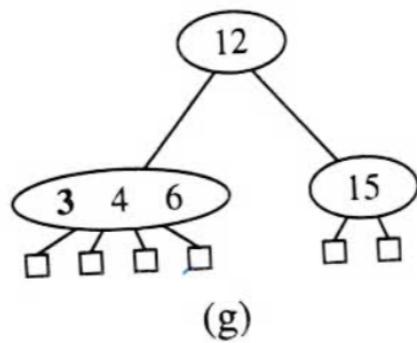
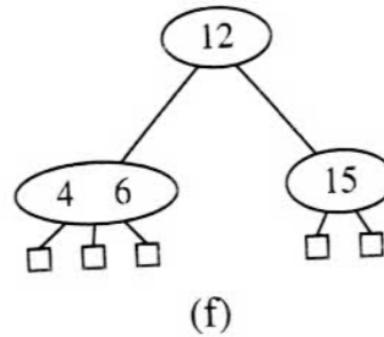
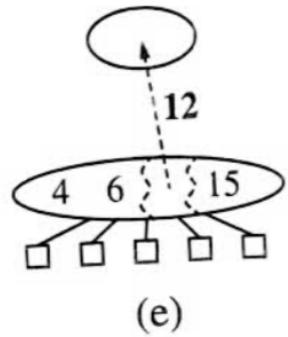
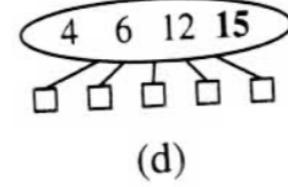
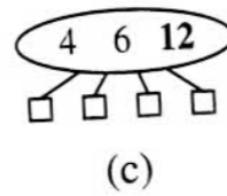
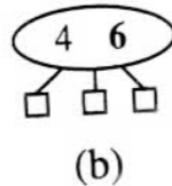
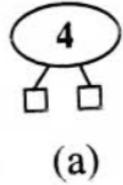
- A search tree with each node having either 2,3 or 4 children.
- Every path from root to external node has same length.
- A tree of height h must have at most 4^h external nodes and at least 2^h external nodes.
- Hence $2^h \leq n+1 \leq 4^h$ where n is number of keys
- $\rightarrow \log(n+1) / 2 \leq h \leq \log (n+1) \rightarrow h$ is $\Theta(\log n)$
- **FindElement()** takes $\Theta(\log n)$ time

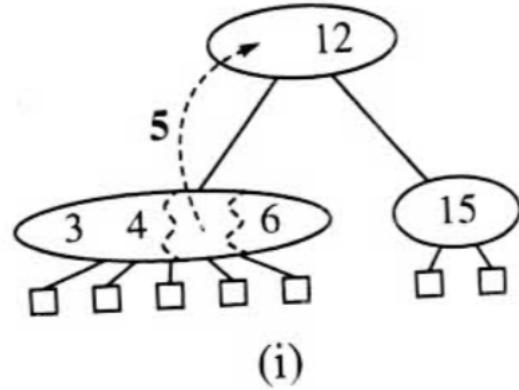
Example of 2-4 tree



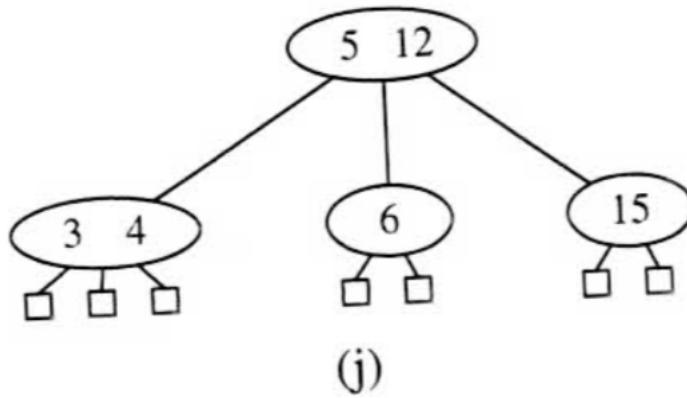
Insertion in 2-4 tree

- Use **FindElement()** to reach bottom internal node.
- **Case 1** : No overflow in internal node to be inserted (i.e. # of keys < 3 before insertion)
- **Case 2** : Overflow in node which is not root. Add key, split the node (making extra child of parent) and push the middle key of node up to be inserted into parent. (recursive case)
- **Case 3** : Overflow in node which is root. Create a new root node and make node and its split node as its children. Increases height of tree.
- **Time complexity** : Down and up phases take $\Theta(\log n)$ time (comparisons + pointer changes)

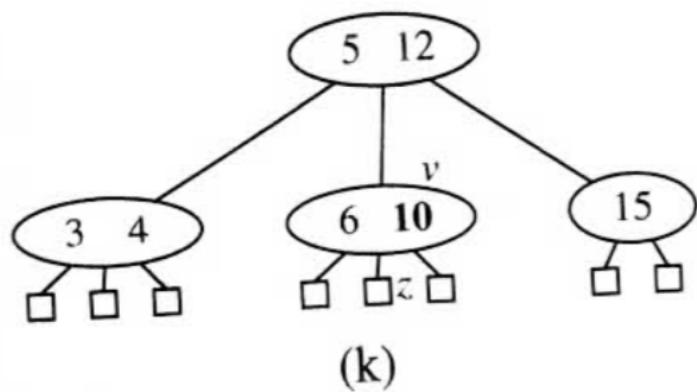




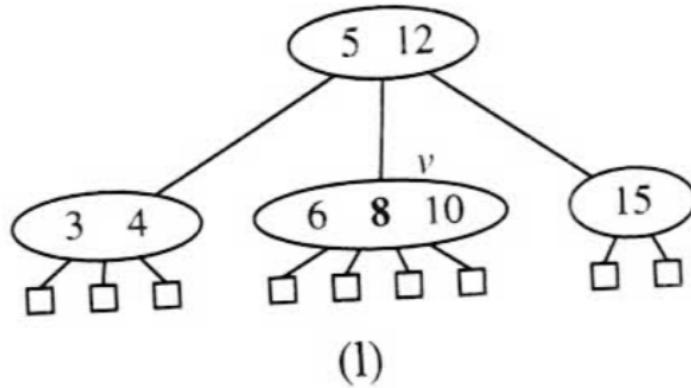
(i)



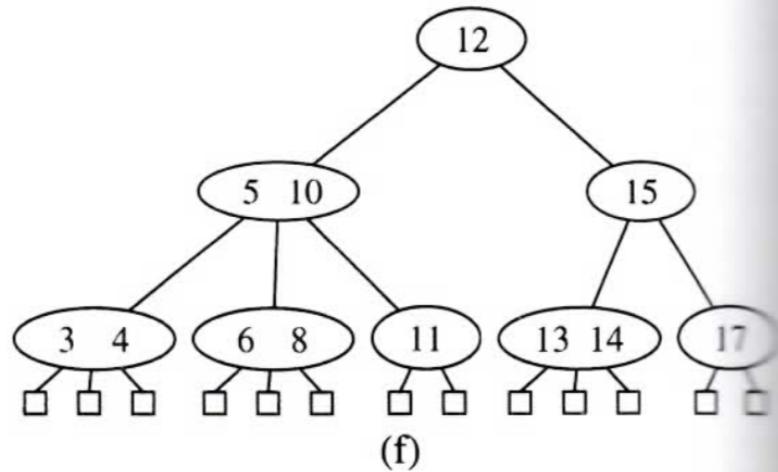
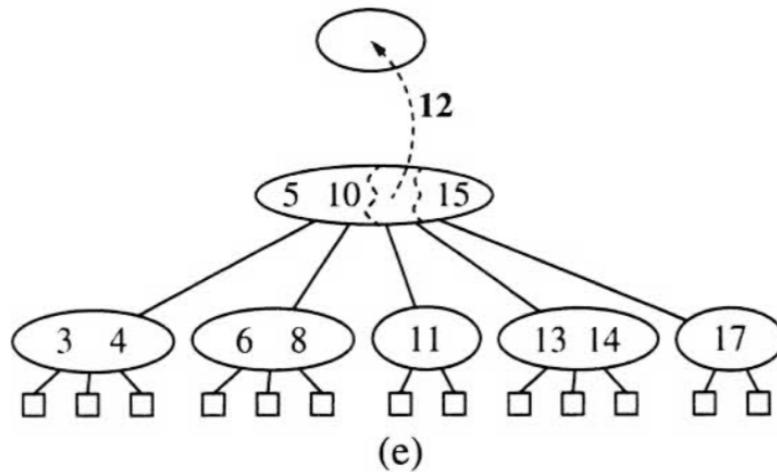
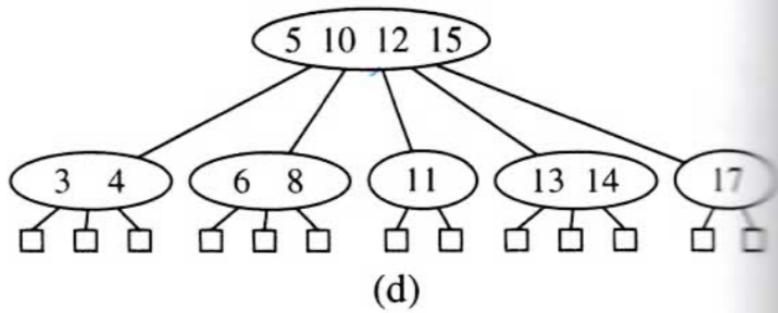
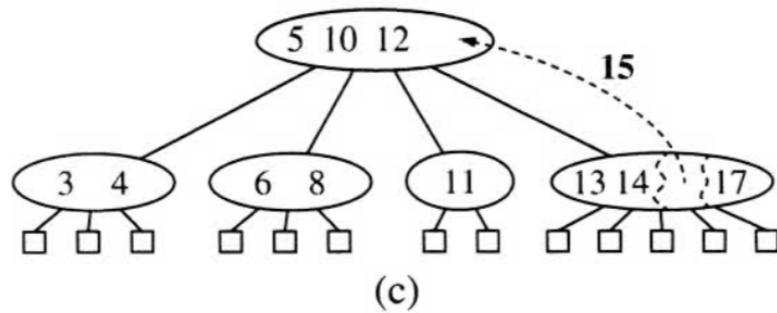
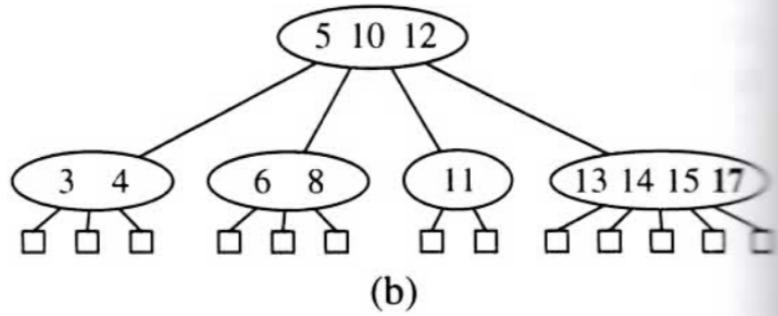
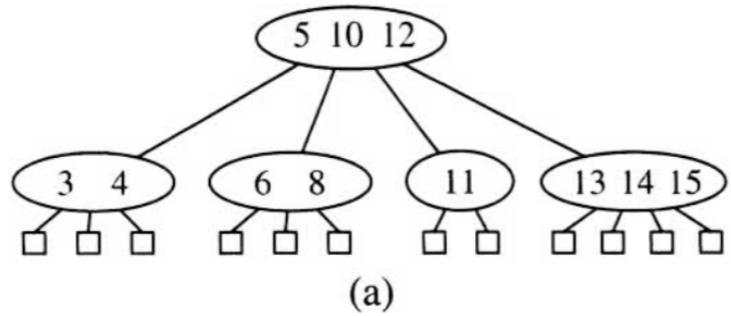
(j)



(k)



(l)



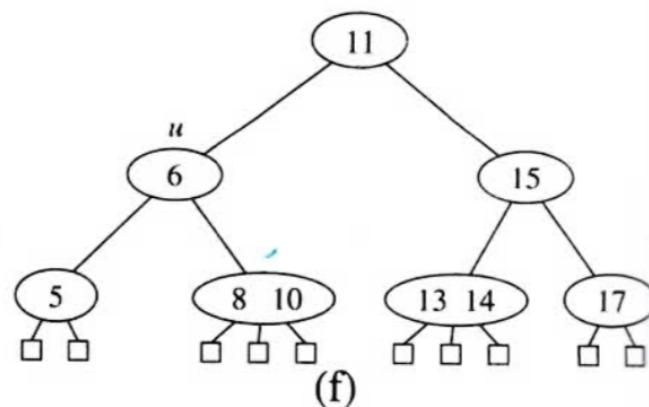
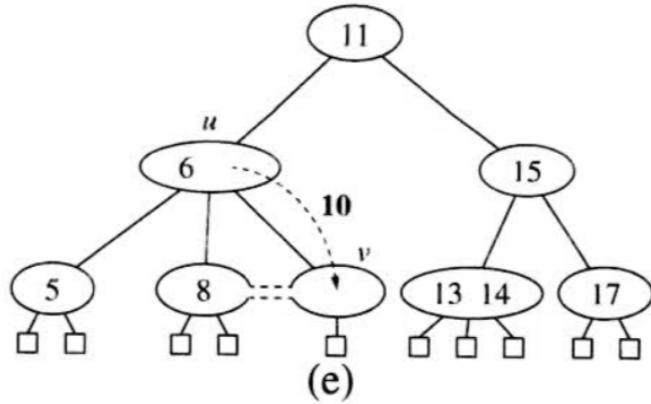
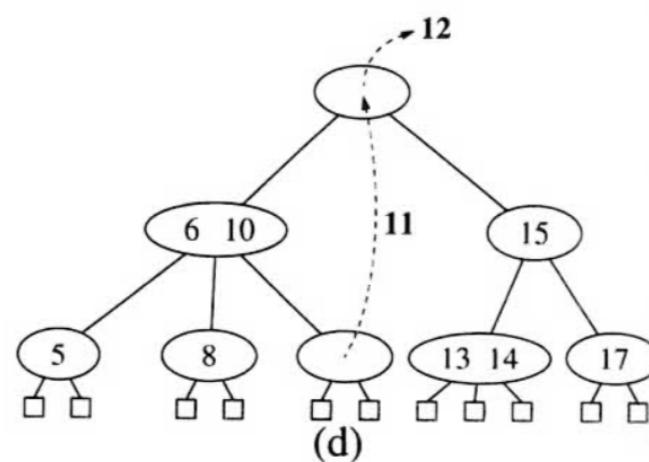
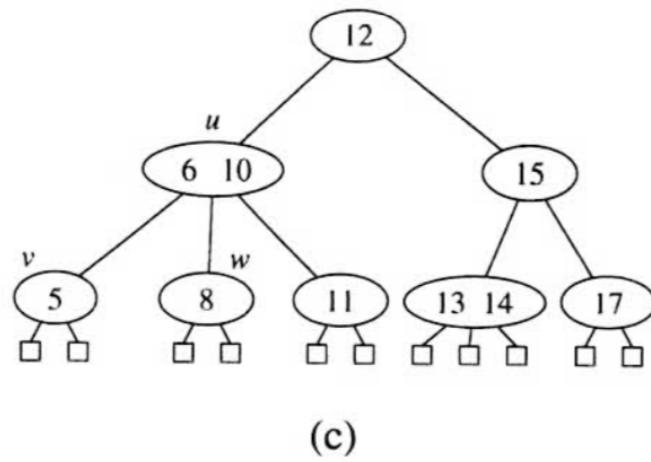
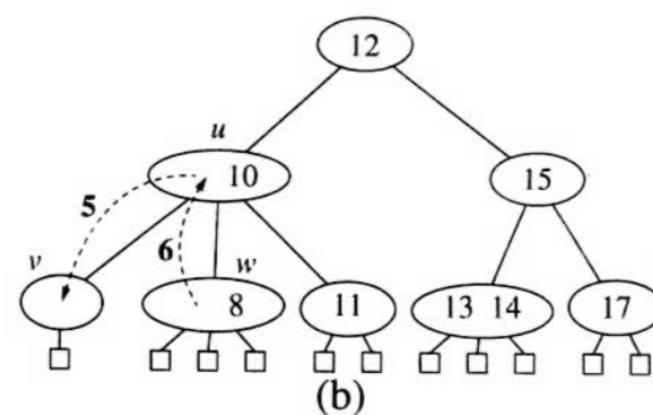
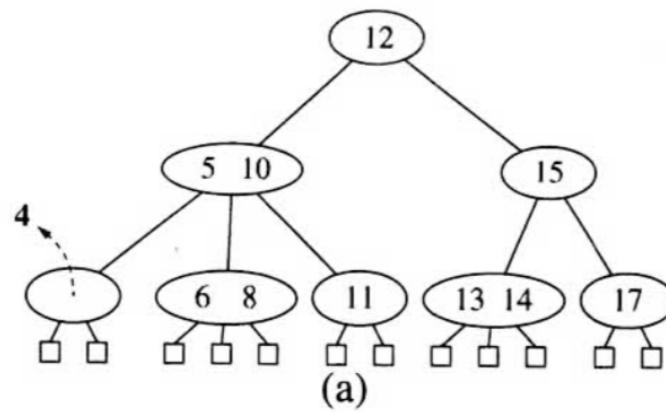
Deletion from 2-4 tree

- If it is not in a bottom internal node, replace key by the smallest item in the subtree whose keys > key.
- Problem reduces to removing key from a bottom internal node with only external children.
- **Case 1:** No underflow - # of keys in node > 1 before deletion.
- **Case 2 :** Underflow in node whose parent is not root. (recursive case)

2a : Immediate sibling has at least 2 keys from which we can transfer a key to this node or thru' a transfer of a key from parent.

2b : All siblings have just one key. Merge with a sibling node by having a new node and moving a key from parent to this new node.

- **Case 3 :** Underflow in node whose root is parent. New replacement root node causes height to decrease.
- **Time complexity :** Down and up phases take $\Theta(\log n)$ time.



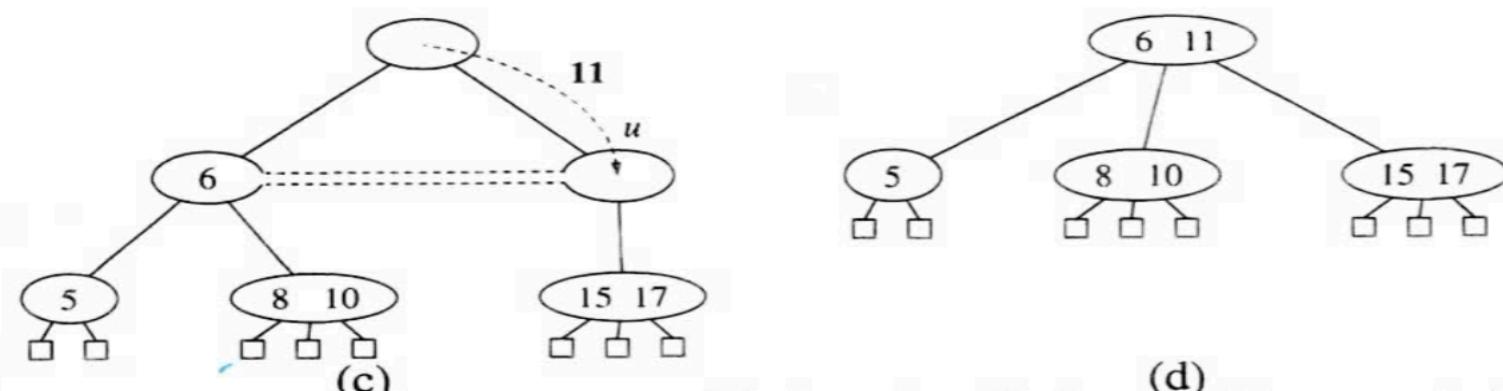
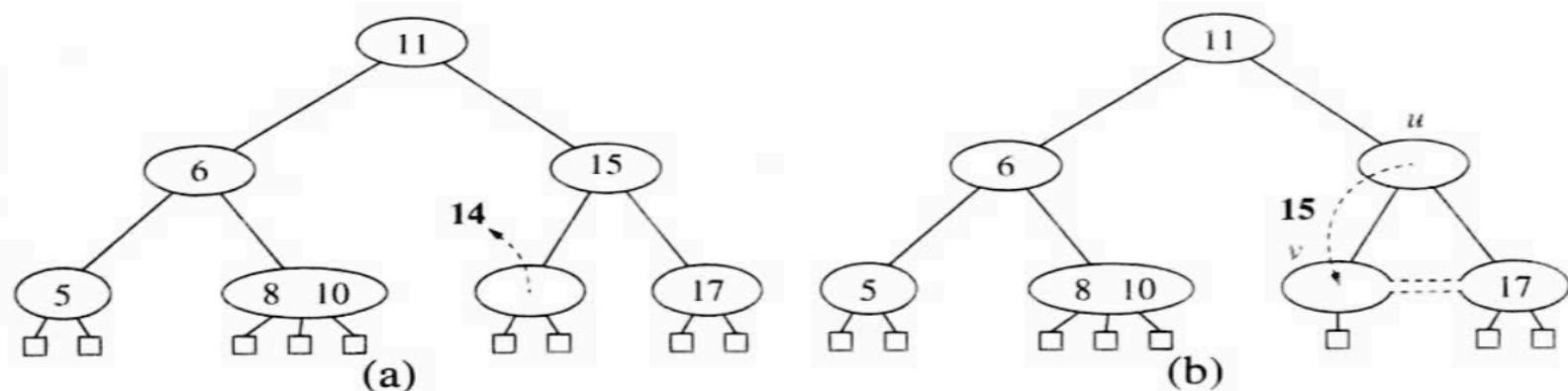
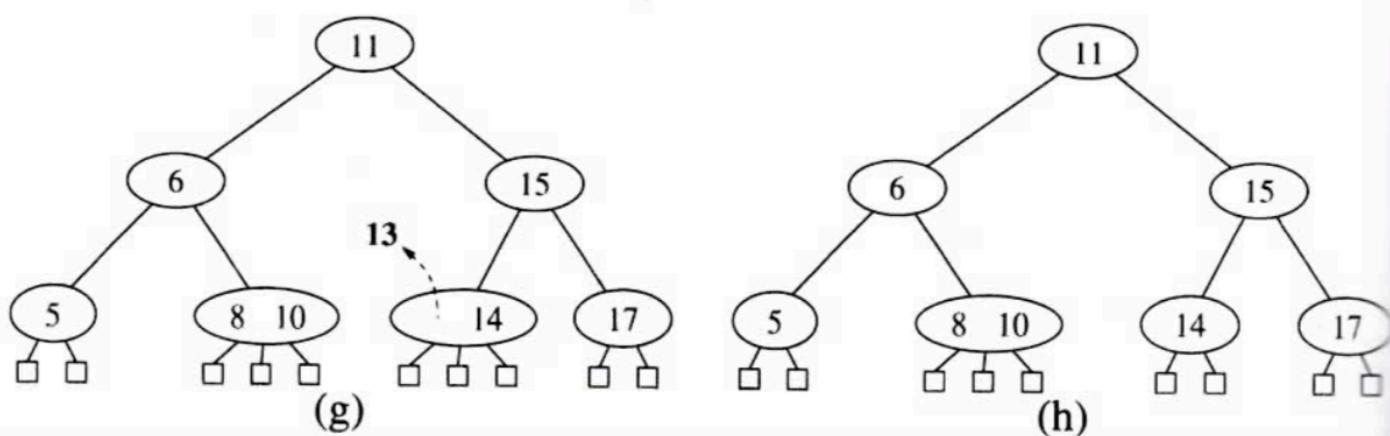
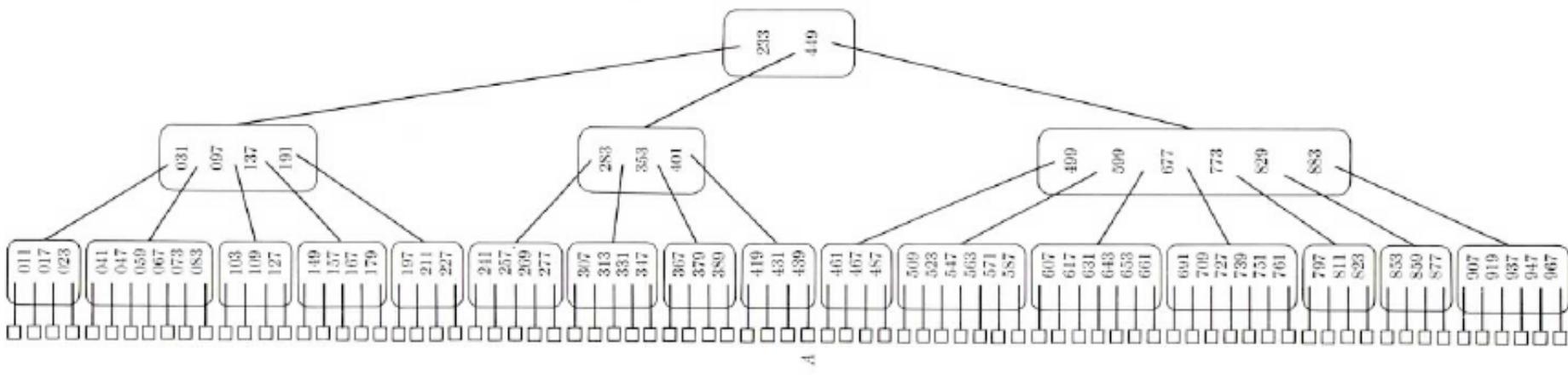


Figure 3.23: A propagating sequence of fusions in a (2,4) tree: (a) removal of 14, which causes an underflow; (b) fusion, which causes another underflow; (c) second fusion operation, which causes the root to be removed; (d) final tree.

B-tree

- m-way search tree used for external searching of records as in a database (B-tree index)
- # of keys in internal node allowed to vary between “d” and “2d”. Branching factor - # of keys + 1
- “d” depends on size of disk block
- Insertion/deletion work the same way as in 2-4 tree.
- Typically only when a node is accessed, it is brought into main memory from disk; entire tree not kept in main memory.
- # of levels $\leq 1 + \log_d ((n+1)/2)$ where n is number of keys
- For $n \approx 2$ million and $d \approx 100$, # of levels is at most 3 requiring at most 3-4 disk accesses.
- In B+ tree, records with keys kept in leaf nodes
- In B* tree, non-root nodes have at least 2/3 full capacity. Instead of splitting/merging, sometimes keys may be transferred from/to sibling nodes.

B- tree (order 7) example



Basic data structures-2

Amortization

- Many dynamic data structures do well for a sequence of operations though the worst-case time complexity based on a single operation may be high
- But we amortize the restructuring cost of a data structure over the sequence of n operations – restructure for future benefit
- Two methods for time complexity analysis :
 - (a) Accounting method
 - (b) Potential function method

Accounting method

- Assign an amortized cost to each operation which may be less than or greater than the actual cost of the operation.
- Operations whose actual cost is less than amortized cost will help pay for operations whose actual cost is more than amortized cost using “credits” which are associated with data structure.
- We require that sum of amortized costs for n operations must be at least the sum of actual costs so that total credit is always positive.
- For dynamic array example, set amortized cost to be 3 for each operation: 1 unit for inserting element itself, one for copying it in the future when array is doubled and 1 unit for an item already copied during array doubling before the insertion of this item
- 2 credits will help pay for actual cost when array is doubled.
- $\text{Total actual cost} \leq \text{Total amortized cost} \leq 3n$

Potential Function

- Associate a “potential” with each state of the data structure that represents prepaid work for future operations.
- Assume D_0 is initial state and let D_i be the state after the i -th operation.
- Define potential function $\phi : D_i \rightarrow \mathbb{R}$ s.t.
 - amortized cost for i -th operation $e_i =$
actual cost $c_i + \phi(D_i) - \phi(D_{i-1})$
- Sum of amortized cost $\sum_{i=1}^n e_i =$ sum of actual cost $\sum_{i=1}^n c_i + \phi(D_n) - \phi(D_0)$ (due to telescopic sum)
- If we choose potential function s.t. $\phi(D_n) \geq \phi(D_0)$, we get an upper bound on total actual cost using total amortized cost.

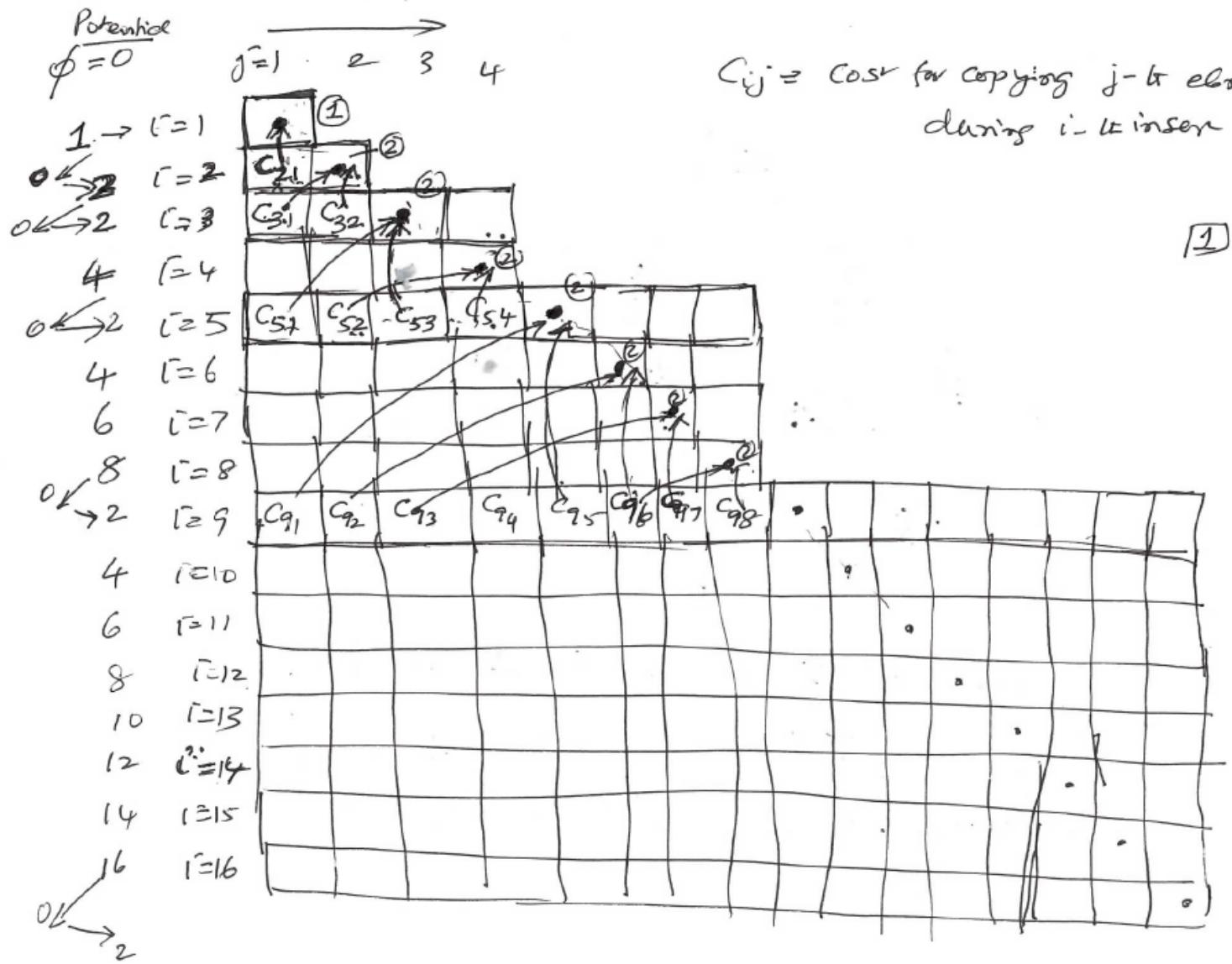
Potential function (contd.)

- For dynamic arrays, we can set $\phi(T) = 2*T.\text{num} - T.\text{size}$.
- Initial value of ϕ is 0 .
- Immediately before an expansion, $T.\text{num} = T.\text{size} \Rightarrow \phi = T.\text{num}$
- Immediately after expansion, $T.\text{size} = 2*T.\text{num} \Rightarrow \phi = 0$.
- Since array is at least half full always, $T.\text{num} \geq T.\text{size}/2$, hence $\phi(T) \geq 0$
- Let num_i - number of elements after i -th operation, note $\text{num}_i - \text{num}_{i-1} = 1$.
- Two cases:
 - a. i -th operation does not cause expansion (

$$e_i = c_i + \phi_i - \phi_{i-1} = 1 + (2 * \text{num}_i - \text{size}) - (2 * \text{num}_{i-1} - \text{size}) = 1 + 2 = 3$$

- b. i -th operation triggers expansion: (1 insert and $\text{num}_i - 1$ copying operations)

$$e_i = \text{num}_i + (2 * \text{num}_i - 2 * (\text{num}_i - 1)) - (2 * (\text{num}_i - 1) - (\text{num}_i - 1)) = 3$$



Linked list implementation

- Each node has prev and next links (doubly linked)
- “count” for number of items in the list

insertAfter(p,e): // p can be null if need to insert as first item in the list

v ← newly allocated node

v.item ← e

v.prev ← p

if p = null

 v.next ← head

else

 v.next ← p.next

if v.next = null

 tail ← v

else

 v.next.prev ← v

if v.prev = null

 head ← v

else

 v.prev.next ← v

count ← count + 1

return v

Linked List (contd.)

remove(p):

 elem \leftarrow p.item

 if p.prev = null

 head \leftarrow p.next

 else

 (p.prev).next \leftarrow p.next

 if p.next = null

 tail \leftarrow p.prev

 else

 (p.next).prev \leftarrow p.prev

 p.prev \leftarrow null; p.next \leftarrow null;

 count \leftarrow count - 1

 return elem

Time complexity comparison

Array vs Linked List

Operations	Array	Linked List
size, isEmpty	O(1)	O(1)
atRank,rankOf,elemAtRank	O(1)	O(n)
first, last, before, after	O(1)	O(1)
insertAtRank,removeAtRank	O(n)	O(n)
insertFirst, insertLast	O(1)	O(1)
insertAfter, insertBefore	O(n)	O(1)
remove	O(n)	O(1)

Binary Tree

- A recursive data structure with a root node and left and right children being roots of binary trees themselves
- We use convention that each internal node has exactly two children and an external node (leaf) has no children
- Operations:
 - leftChild(v) – left child of node v ; error for external node v
 - rightChild(v) – right child of node v , error for external node v
 - isInternal(v) – true iff v is an internal node

Binary tree properties

- Depth of a node is number of internal nodes in the path from root to the node. Root has depth 0
- Height h of a tree is the maximum depth of an external node in the tree.
- Height of tree = $1 + \max(\text{height of left sub tree}, \text{height of right sub tree})$
- # of external nodes = $1 + \# \text{ of internal nodes}$ (by induction on tree height)
- $h \leq \# \text{ of internal nodes} \leq 2^h - 1$
- $h+1 \leq \# \text{ of external nodes} \leq 2^h$
- $2h+1 \leq \# \text{ of nodes} \leq 2^{h+1} - 1$
- For a tree with n nodes, $\log(n+1) - 1 \leq h \leq (n-1)/2$

Binary tree traversals

- Preorder – root, left subtree, right subtree
- Inorder – left subtree, root, right subtree
- Postorder – left subtree, right subtree, root

binaryPreorder(T,v):

Input : Binary tree T, a node v

Output: Perform action on each node of subtree rooted at node v

 performAction(v)

 if T.isInternal(v)

 binaryPreorder(T, T.leftChild(v))

 binaryPreorder(T, T.rightChild(v))

Time complexity – O(n) as each node is visited only once

BT array implementation

- A tree with height h needs an array $A[0..n]$ where $n = 2^{h+1} - 1$
- $\text{Rank}(v)$ determines **index** of node v in A
- $\text{Rank}(v) = 1$ for root v
 $\text{Rank}(\text{leftChild}(v)) = 2 * \text{Rank}(v)$
 $\text{Rank}(\text{rightChild}(v)) = 2 * \text{Rank}(v) + 1$
- For a sparse tree, many cells in A will be unused
space complexity is $O(2^n)$ in the worst-case
- Simple and fast for many tree operations.
- Can use dynamic arrays for expanding trees.
- Revisit this implementation for heap ADTs.

BT Linked structure

- Similar to linked list, each node has item as well links to left and child children nodes (null for external nodes) and optionally a link to its parent (null for root)
- Easy to extend to non-binary trees
- Space-efficient as complexity is $O(n)$ where n is number of nodes
- Time complexity for many operations is comparable to array implementation but small overhead for dereferencing the links.
- Revisit this implementation for binary search trees.

Basic data structures-3

Min-Priority Queue ADT

- Allows a totally ordered set of elements to be stored in such a way that “minimum” element can be extracted efficiently – self-reorganizing structure
- Useful for task scheduling, efficient sorting
- Operations:
 - insertElement(e)** – insert element in queue
 - removeMin()** – remove and return smallest
 - minElement()** – return minimum element
- In a Max-heap, maximum elements are of interest.
- In Java PriorityQueue<E> is a class based on unbounded heap

PQ – simple array implementation

- Two approaches:
 - (a) Keep array unordered
 - (b) Keep array sorted at all times
- Unordered array
 - (a) insertElement(E) – add element to end of array – $O(1)$ time
 - (b) minElement() – $O(n)$ time even in best-case
 - (c) removeMin() – $O(n)$ time even in best-case
basis of **Selection Sort** $O(n^2)$ even in best-case
- Sorted array (max element in position 0)
 - (a) insertElement(E) - find position to insert and shift elements to right – $O(1)$ time best-case and $O(n)$ time in worst-case
 - (b) minElement() – $O(1)$ time
 - (c) removeMin() – $O(1)$ time (basis of **Insertion Sort** $O(n^2)$ in worst-case and $O(n)$ time in best-case)

Min PQ Binary heap

- It is a complete binary tree with array implementation. In the tree we fill every level from left to right before proceeding to next level
- All elements stored in internal nodes only. Ignore external nodes in discussion.
- The element at a node (except root) is always \geq element stored in its parent node – **heap property**
- **Theorem 1** : Root element of a subtree is the minimum element in that subtree (use induction)
- Recursive structure
- **Theorem 2** : The height of an n-element heap is at most $\lceil \log n \rceil$. Note # of nodes of a heap of height h is at least 2^h and at most $2^{h+1} - 1$
- Easy to see that **minElement()** takes O(1) time.

Heap operations

insertElement(E):

 size \leftarrow size + 1

 A[size] \leftarrow E

 elemPos \leftarrow size

 while elemPos > 1

 parentPos \leftarrow \lfloor elemPos/2 \rfloor

 if A[elemPos] < A[parentPos]

 // swap parent with child

 temp \leftarrow A[elemPos]

 A[elemPos] \leftarrow A[parentPos]

 A[parentPos] \leftarrow temp

 elemPos \leftarrow parentPos // (1)

 else // (2)

 break // (3)

Heap operations (contd.)

- **Correctness proof:**

Code equivalent to removing (2), (3) and moving (1) outside IF statement and complexity bounded by this modified code's complexity.

Loop invariance : Subtree rooted at elemPos is a heap

We prove this using induction on loop index i.

Basis i=0: Before loop begins, subtree rooted at elemPos is just one-element subtree and is a heap

Induction Step : Assume true for $i=m-1$, $m \geq 1$. For m-th iteration, after swap of child with parent, we see that new parent is still smaller than the sibling (as old parent was smaller than the sibling). So must be a heap for iteration m also.

- **Time complexity:** Initialization and each iteration takes $O(1)$ time and # of iterations bounded by height of the heap which is at most $\lceil \log n \rceil$.
- Time complexity is $O(\log n)$.

Heap operations (contd.)

removeMin():

```
minVal ← A[1]
A[1] ← A[size] // move last element to root
size ← size - 1
elemPos ← 1
bubbleDown(A,elemPos)
return minVal
```

bubbleDown(A,elemPos): // Takes **O(log n)** time

```
while elemPos < [size/2]
    childPos ← 2 * elemPos
    if childPos + 1 <= size && A[childPos+1] < A[childPos]
        childPos ← childPos + 1
    if A[childPos] < A[elemPos]
        // swap parent with child
        temp ← A[childPos]
        A[childPos] ← A[elemPos]
        A[elemPos] ← temp
        elemPos ← childPos
    else
        break
```

Heap construction

- Given n elements, using **insertElement()** to construct a heap takes $O(n \log n)$ time. Can be done faster.

Heapify(A[0..n]):

```
for k ← ⌊n/2⌋ to 1 by -1  
    bubbleDown(A,k)
```

- Start with heaps of height 0, then construct heaps of height 1, then height 2 and so on.
- Correctness works by correctness of **bubbleDown()** and the recursive nature of heap
- Time complexity:**

Claim : There are at most $\lceil n/2^{i+1} \rceil$ heaps of height i for $0 \leq i \leq h$ where h is height of the heap. (Proof by induction on i).

bubbleDown() for a heap of height i takes at most $c*i$ for some constant c .

- Total time $\leq c/2 * \sum_{i=1}^{\lceil \log n \rceil} \lceil n/2^i \rceil * i \leq c1 * n * \sum_{i=1}^{\log n} i/2^i$
- Total time is thus $O(n)$ since $\sum_{i=1}^{\infty} i/2^i$ converges and hence is $O(1)$

Probabilistic data structures for dictionaries and sets

Ravi Varadarajan

Bloom filters

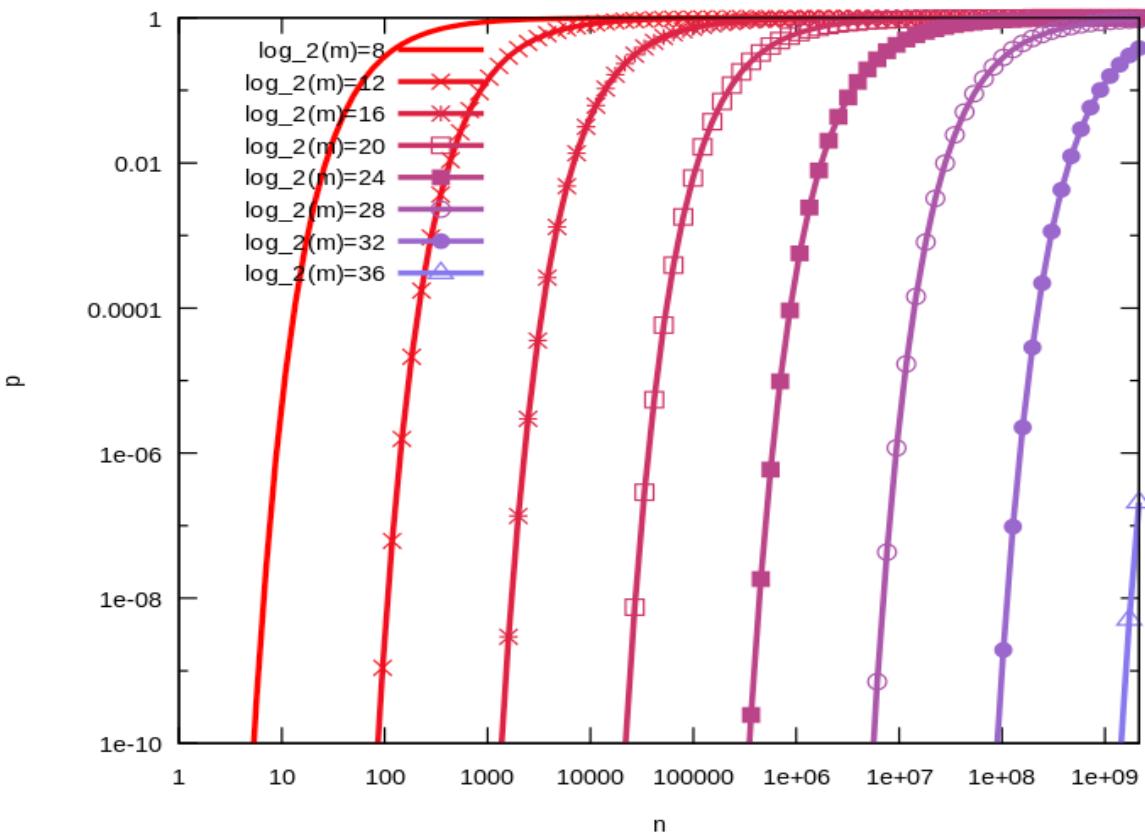
- **isMember(S,e)** – check if e belongs to S
- Can use dictionary implementations (hash table, search trees)
- Alternatively we can use a bit-vector of size n where i-th bit indicates if i-th element belongs to the set
- **Advantage** : Easy to perform union, intersection operations.
- **Disadvantage** : Space requirements very high for large number of elements.
- If we relax restriction that error-free hash is required, we can use probabilistic data structures using hashes.
- Here we use a bit vector of m-bits (m smaller than n)
- We use “k” ($\ll m$) hash functions to map an element to “k” positions in the bit vector

False positives in Bloom Filters

- To add an element, need to set bits to 1 in all positions indexed by “k” hash functions.
- To query an element, need to check all “k” positions indexed by hash functions to see if they are all 1's.
- If an element is removed from the set, we cannot set any of the “k” bits to 0 as other elements may be hashed into these positions. This causes “false” positives in set membership test.
- Assuming independence of bits being set to 1, we can show that the probability of a “false” positive (assuming “n” elements inserted) is given by :

$$[1 - (1 - 1/m)^{kn}]^k \approx (1 - e^{-kn/m})^k$$

- Bloom filters very fast and have less space requirements; typically we require disk accesses to get items only after we find the key exists.



The false positive probability p as a function of number of elements n in the filter and the filter size m . An optimal number of hash functions $k = (m/n) \ln 2$ has been assumed.

Chinese Remainder Theorem for integers

- Two integers a and b are relatively prime if $\gcd(a,b) = 1$
- Let p_0, p_1, \dots, p_{k-1} be k relatively prime numbers. Then any integer r between 0 and $p_0 * p_1 * \dots * p_{k-1} - 1$ can be represented as $(r_0, r_1, \dots, r_{k-1})$ (called “residues” or “moduli”) where $r_j = q \bmod p_j$. We write it as $r \leftrightarrow (r_0, r_1, \dots, r_{k-1})$
- This representation is unique due to Chinese Remainder Theorem (stated below without proof):
- **Given** : residues $(u_0, u_1, \dots, u_{k-1})$ w.r.t to p_0, p_1, \dots, p_{k-1}
Then $u = \sum_{i=0}^{k-1} c_i d_i u_i$ modulo p where
$$p = p_0 * p_1 * \dots * p_{k-1}$$
 and $c_i = p / p_i$, and $d_i = c_i^{-1} \bmod p_i$
- **Example:** $p_0 = 2, p_1 = 3, p_2 = 5$ and $p_3 = 7$ and $(u_0, u_1, u_2, u_3) = (1, 2, 4, 3)$. What is u such that $u \leftrightarrow (1, 2, 4, 3)$?

Chinese Remainder Algorithm For Integers

Example : when $k = 4$

$$\begin{aligned}\sum_{i=0}^3 c_i d_i u_i &= \sum_{i=0}^3 (p / p_i) d_i u_i = \\d_0 u_0 p_1 p_2 p_3 + d_1 u_1 p_0 p_2 p_3 + d_2 u_2 p_0 p_1 p_3 + \\d_3 u_3 p_0 p_1 p_2 \\&= p_0 p_1 (d_2 u_2 p_3 + d_3 u_3 p_2) + p_2 p_3 (d_0 u_0 p_1 + d_1 u_1 p_0) \\&= q_{11} (s_{00} q_{01} + s_{01} q_{00}) + q_{10} (s_{02} q_{03} + s_{03} q_{02})\end{aligned}$$

Where at level 0 we have $q_{00} = p_0$, $q_{01} = p_1$, $q_{02} = p_2$ and $q_{03} = p_3$;
 $s_{00} = d_0 u_0$, $s_{01} = d_1 u_1$, $s_{02} = d_2 u_2$ and $s_{03} = d_3 u_3$

And at level 1, if we define $q_{10} = q_{00} * q_{01} = p_0 p_1$ and
 $q_{11} = q_{02} * q_{03} = p_2 p_3$; $s_{10} = s_{00} q_{01} + s_{01} q_{00}$ and $s_{11} = s_{02} q_{03} + s_{03} q_{02}$,

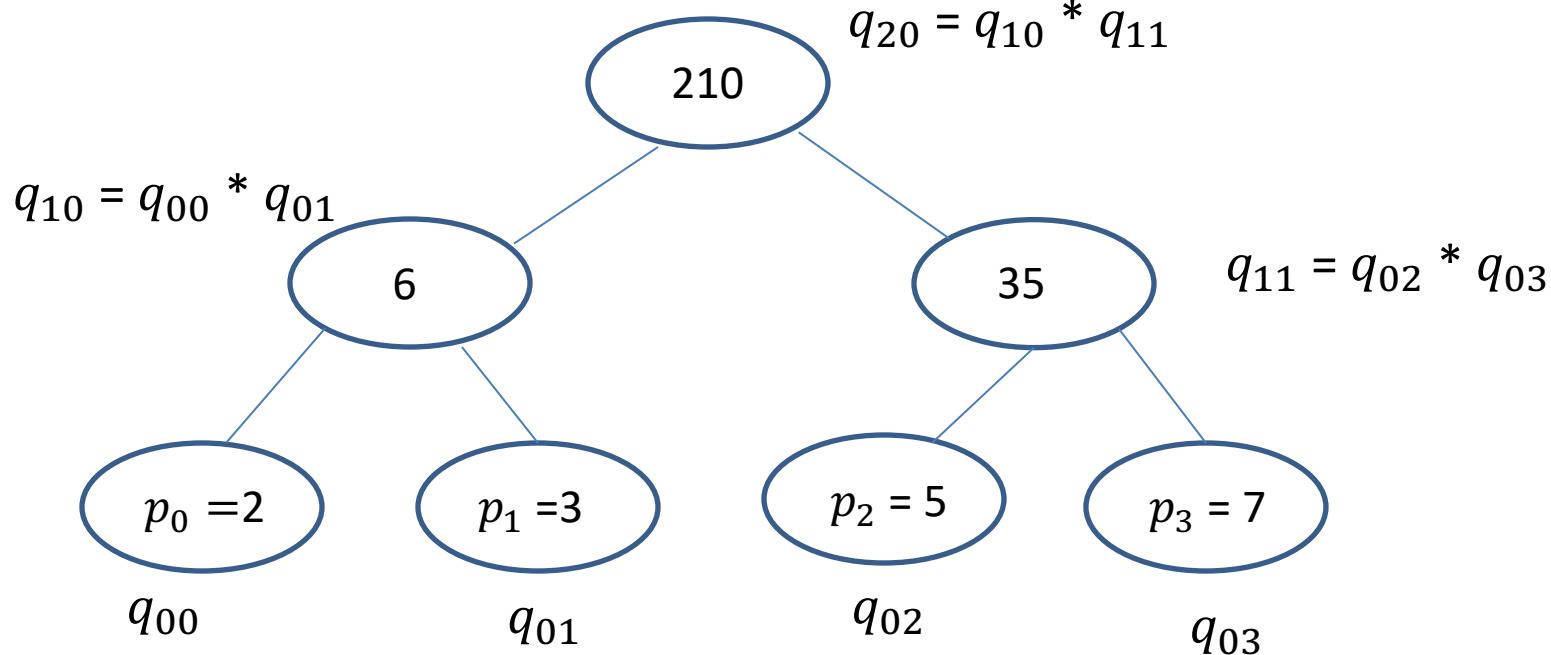
We get at level 2

$$\sum_{i=0}^3 c_i d_i u_i = s_{20} = s_{10} q_{11} + s_{11} q_{10}$$

We can continue this pattern to more than 3 levels for larger values of k .

Chinese Remainder Algorithm For Integers

Pre-conditioning step:

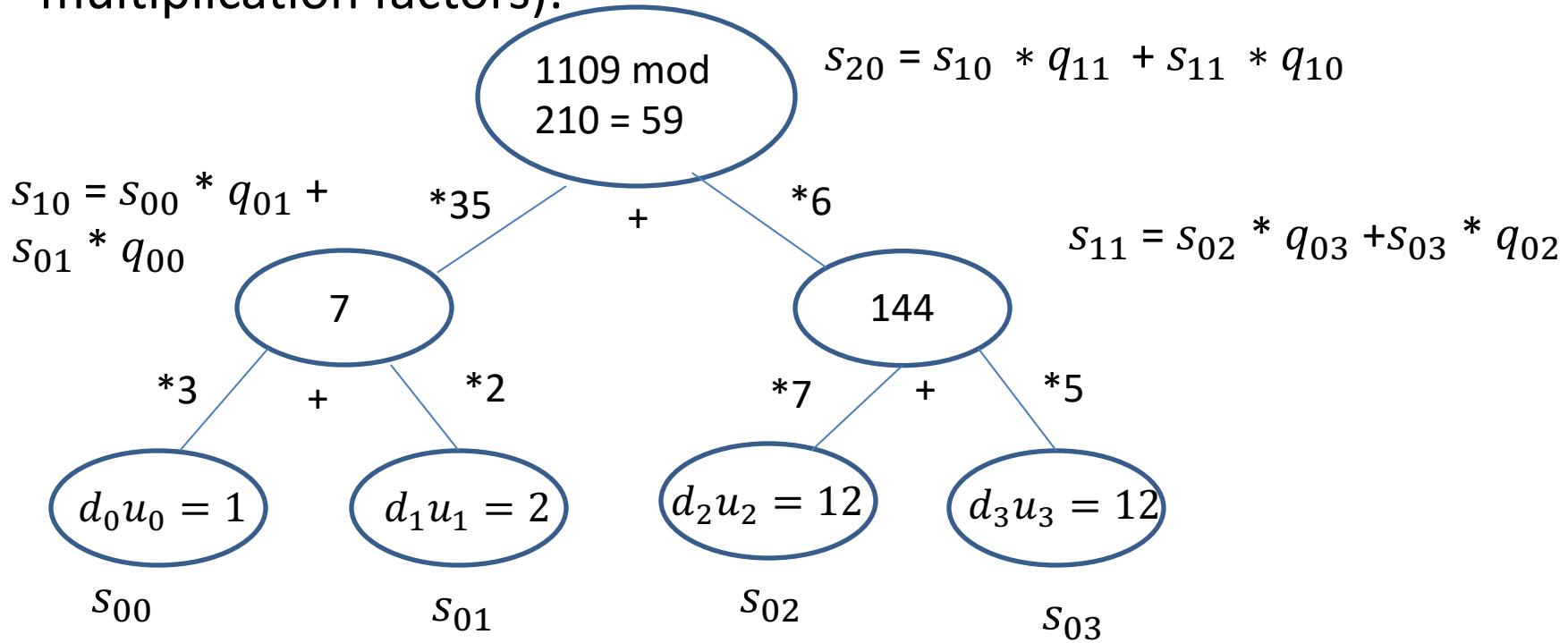


Complexity = $O(k)$ integer multiplications

Done only once for the given k relatively prime integers

Chinese Remainder Algorithm For Integers

Computing step (use tree constructed in pre-conditioning for multiplication factors):



$$d_0 = (3*5*7)^{-1} \bmod 2 = 105^{-1} \bmod 2 = 1 \text{ (i.e. } 1*105 \bmod 2 = 1\text{)}; u_0 = 1$$

$$d_1 = (2*5*7)^{-1} \bmod 3 = 70^{-1} \bmod 3 = 1 \text{ (i.e. } 1*70 \bmod 3 = 1\text{)}; u_1 = 2$$

$$d_2 = (2*3*7)^{-1} \bmod 5 = 42^{-1} \bmod 5 = 3 \text{ (i.e. } 3*42 \bmod 5 = 1\text{)}; u_2 = 4$$

$$d_3 = (2*3*5)^{-1} \bmod 7 = 30^{-1} \bmod 7 = 4 \text{ (i.e. } 4*30 \bmod 7 = 1\text{)}; u_3 = 3$$

Complexity = O(k) integer multiplications + O(k) inverse operations

Complexity Theory

Ravi Varadarajan

k -tape Turing Machine

Definition: A k -tape Turing machine

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

has k different tapes and k different read/write heads:

- Q is finite set of states
- Σ is input alphabet (where $\sqcup \notin \Sigma$)
- Γ is tape alphabet with $(\{\sqcup\} \cup \Sigma) \subseteq \Gamma$
- q_0 is start state $\in Q$
- q_{accept} is accept state $\in Q$
- q_{reject} is reject state $\in Q$
- δ is transition function

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

where $\Gamma^k = \underbrace{\Gamma \times \Gamma \times \cdots \times \Gamma}_{k \text{ times}}$.

Nondeterministic TM

Definition: A **nondeterministic Turing machine** (NTM) M can have several options at every step. It is defined by the 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where

- Q is finite set of states
- Σ is *input alphabet* (without blank \sqcup)
- Γ is *tape alphabet* with $\{\sqcup\} \cup \Sigma \subseteq \Gamma$
- q_0 is *start state* $\in Q$
- q_{accept} is *accept state* $\in Q$
- q_{reject} is *reject state* $\in Q$
- δ is *transition function*

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Encoding of Problems

- Recall: TM running time defined as fcn of length of encoding $\langle x \rangle$ of input x .
- But for given problem, many ways to encode input x as $\langle x \rangle$.
- For integers
 - binary is good
 - unary is bad (exponentially worse)
 - **Example:** Suppose input to TM is the number 18 in decimal.
 - ▲ if encoding in binary, $\langle 18 \rangle = 10010$
 - ▲ if encoding in unary, $\langle 18 \rangle = 11111111111111111111$
- For graphs
 - list of nodes and edges (good)
 - adjacency matrix (good)

The Class P

Because of polynomial equivalence of DTM models,

- group languages solvable in $O(n^2)$, $O(n \log n)$, $O(n)$, etc., together in the **polynomial-time class**.

Definition: The class of languages that can be decided by a single-tape DTM in polynomial time is denoted by P, where

$$P = \bigcup_{k \geq 0} \text{TIME}(n^k).$$

Remarks:

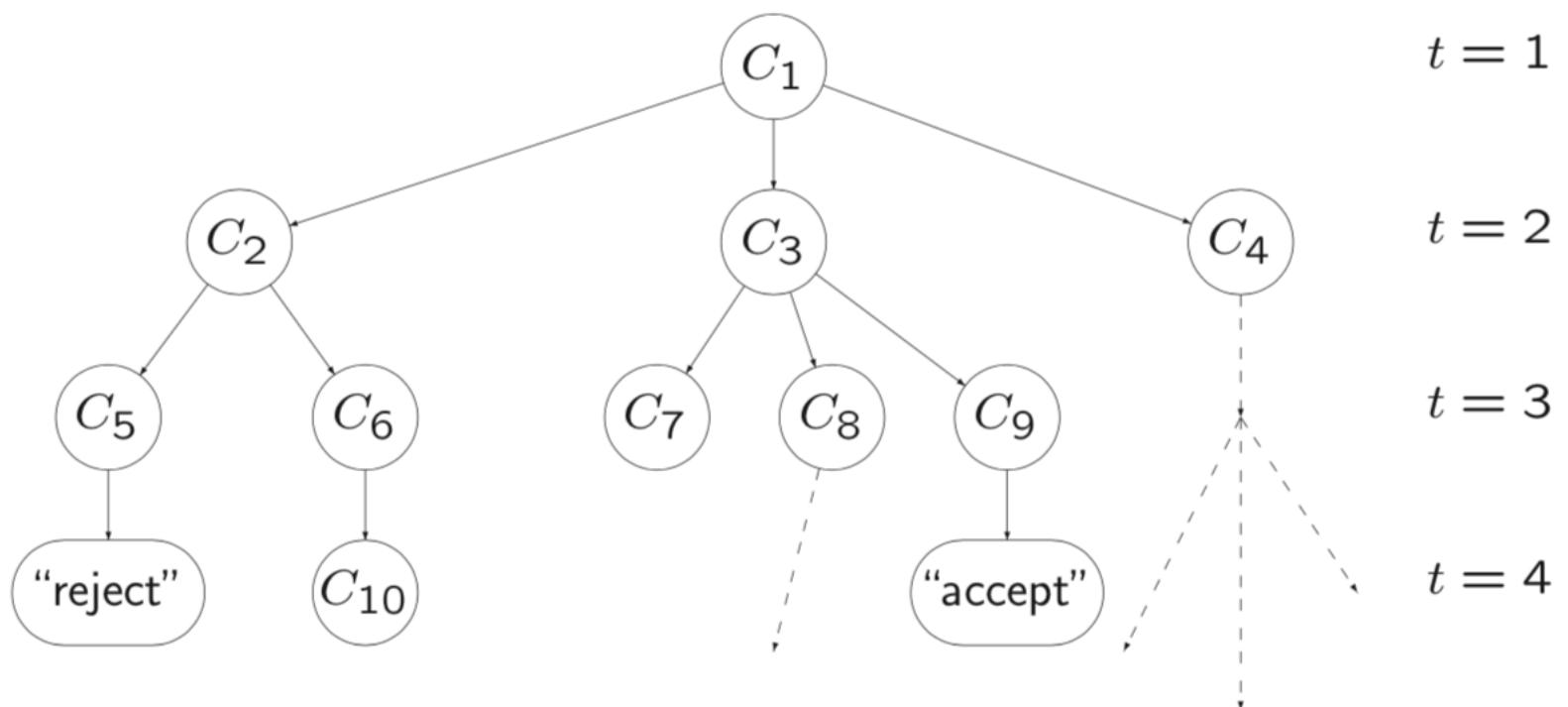
- If we ask if a particular problem A is solvable in polynomial time (i.e., is $A \in P$?),
 - answer is independent of deterministic computational model used.
- Class P roughly corresponds to *tractable* (i.e., realistically solvable) problems.

RAM Model vs TM model

- Recall RAM model has infinite memory and unlimited word size
- RAM model assumes unit times for primitive operations such as arithmetic operations, comparisons etc. Time is independent of word size (uniform cost model)
- TM uses logarithmic complexity model where input size for numeric data uses logarithmic space under “reasonable encoding” schemes
- In TM, arithmetic operations take time that are dependent on the logarithmic size of the numerical inputs.
- RAM model can be simulated by a TM with polynomial time complexity overhead.

Computing With NTMs

- On any input w , evolution of NTM represented by a **tree of configurations** (rather than a single path).
- If \exists (at least) one accepting leaf, then NTM **accepts**.



Class NP

Definition: NP is the class of languages that have polynomial-time verifiers.

Remarks:

- Class NP important because it contains many problems of practical interest
 - *HAMPATH*
 - Travelling salesman
 - All of P
- The term NP comes from **nondeterministic polynomial time**.
 - Can define NP in terms of nondeterministic polynomial-time TMs.
- Recall: a nondeterministic TM (NTM) makes “lucky guesses” in computation.

Equivalent Definition of NP

Theorem 7.20

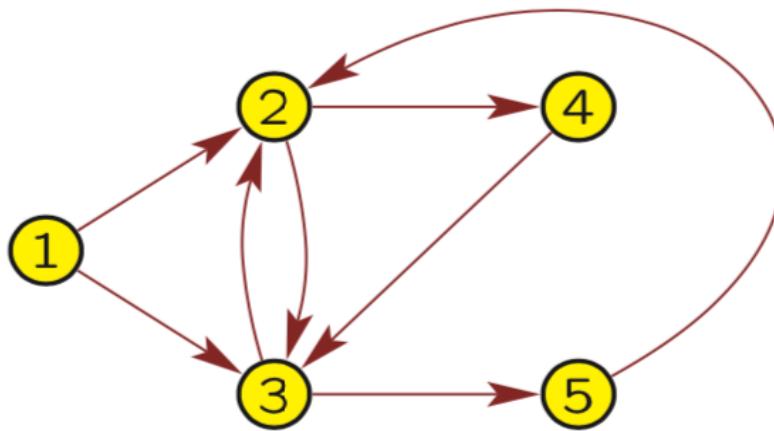
A language is in NP if and only if it is decided by some nondeterministic polynomial-time TM.

Proof Idea:

- Recall language in NP has (deterministic) poly-time verifier.
- Given a poly-time verifier, build NTM that guesses the certificate c and then runs verifier using c .
 - NTM runs in nondeterministic polynomial time.
- Given a poly-time NTM, build verifier with certificate c that tells NTM which is accepting branch.
 - Verifier runs in deterministic polynomial time.

Example of Problem in P: PATH

- **Decision problem:** Given directed graph G with nodes s and t , does G have a path from s to t ?



- $\langle G, s, t \rangle$ is an **instance** of the decision problem (for a particular encoding scheme).
- **Language** of decision problem comprises YES instances:
$$PATH = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with path from } s \text{ to } t \}.$$
- For graph G above, $\langle G, 1, 5 \rangle \in PATH$, but $\langle G, 2, 1 \rangle \notin PATH$.

NTIME($t(n)$) and NP

Definition:

$\text{NTIME}(t(n)) = \{ L \mid L \text{ is a language decided by an } O(t(n))\text{-time NTM} \}$

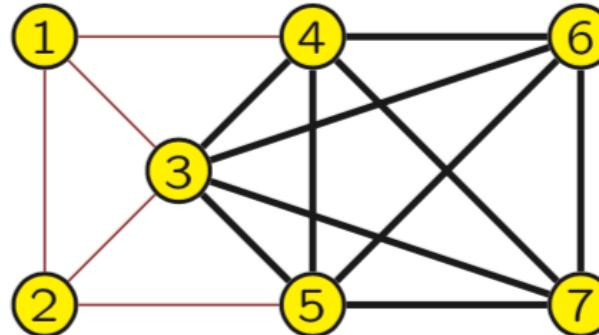
Corollary 7.22

$$\text{NP} = \bigcup_{k \geq 0} \text{NTIME}(n^k).$$

Remark:

- NP is insensitive to choice of “reasonable” nondeterministic computational model.
 - This is because all such models are polynomially equivalent.

Example: CLIQUE



- **Definition:** A **clique** in a graph is a subgraph in which every two nodes are connected by an edge, i.e., clique is a complete subgraph.

- **Definition:** A **k -clique** is a clique of size k .

- **Decision problem:** Given graph G and integer k , does G have k -clique?

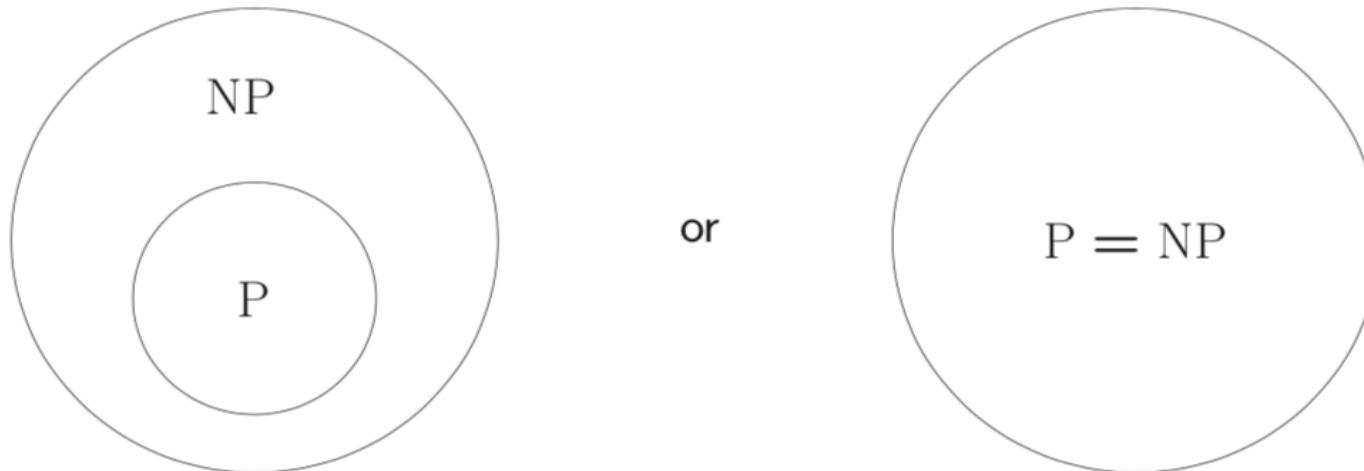
- **Instance** is $\langle G, k \rangle$.
- **Language** of decision problem

$$\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}.$$

- For graph G above, $\langle G, 5 \rangle \in \text{CLIQUE}$, but $\langle G, 6 \rangle \notin \text{CLIQUE}$.

P vs. NP Question

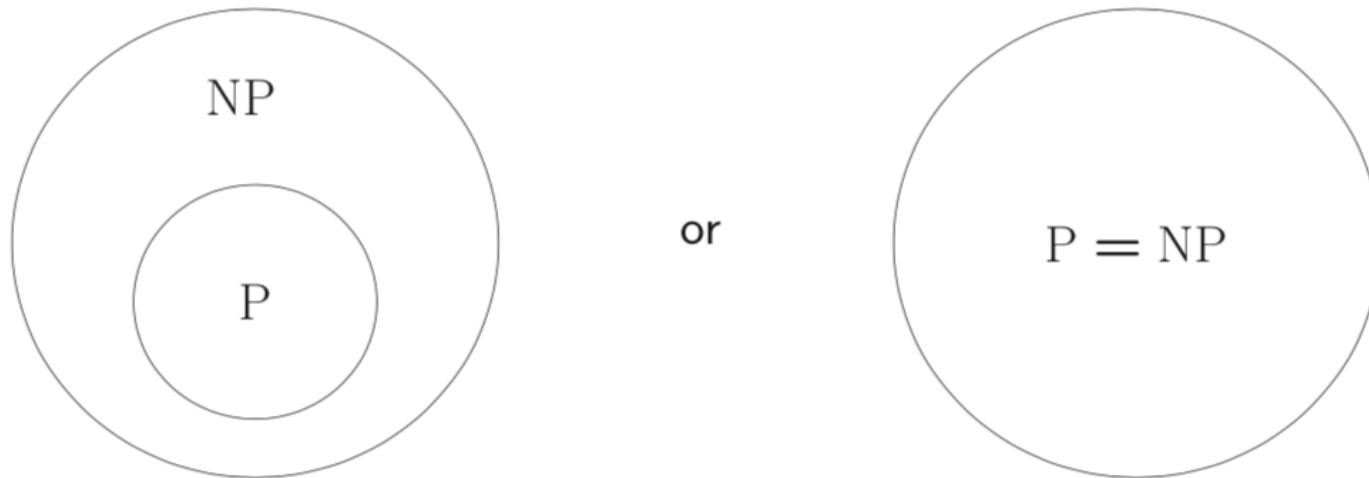
- Languages in P have polynomial-time **deciders**.
- Languages in NP have polynomial-time **verifiers**.



- Answering question whether $P = NP$ or not is one of the great unsolved mysteries in computer science and mathematics.
 - Most computer scientists believe $P \neq NP$; e.g., jigsaw puzzle.
 - Clay Math Institute (www.claymath.org) has \$1,000,000 prize to anyone who can prove either $P = NP$ or $P \neq NP$.

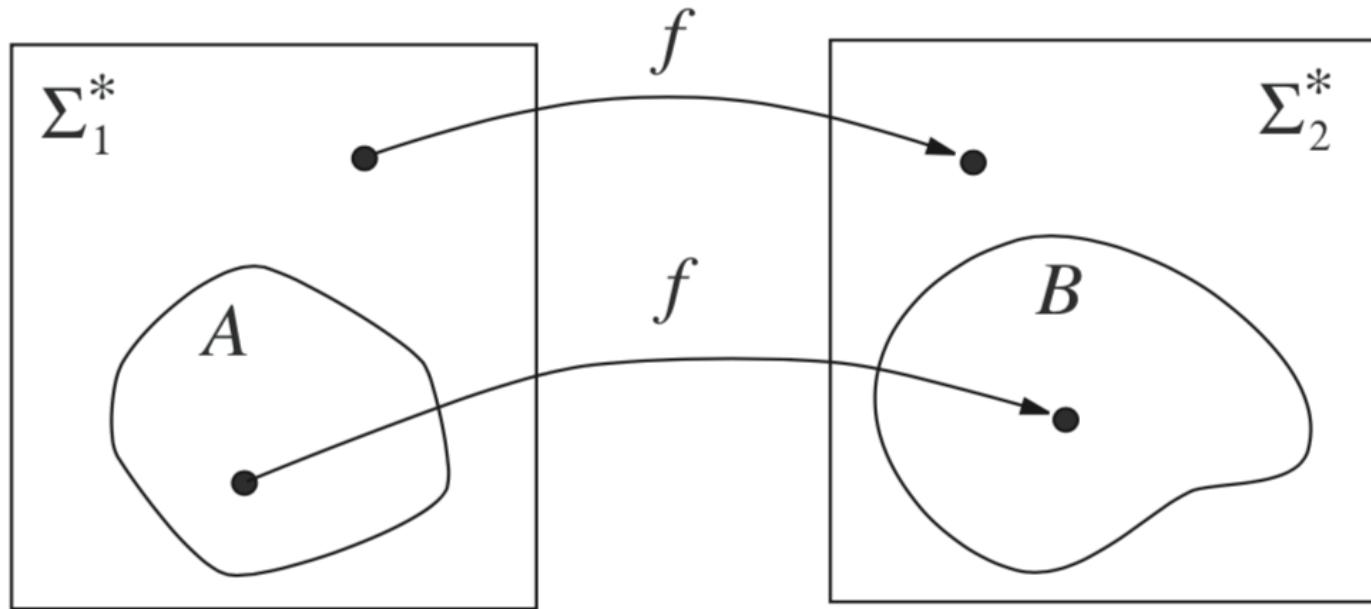
Remarks on P vs. NP Question

- If $P \neq NP$, then
 - languages in P are **tractable** (i.e., solvable in polynomial time)
 - languages in $NP - P$ are **intractable** (i.e., polynomial-time solution doesn't exist).



- If any NP language $A \notin P$, then $P \neq NP$.
 - Nobody has been able to prove \exists language $\in NP - P$.

Polynomial-Time Mapping Reducible

 $w \in A$ \iff $f(w) \in B$ YES instance for problem A \iff YES instance for problem B

- converts questions about membership in A to membership in B
- conversion is done **efficiently** (i.e., in polynomial time).

Polynomial-Time Mapping Reducible

Theorem 7.31

If $A \leq_P B$ and $B \in P$, then $A \in P$.

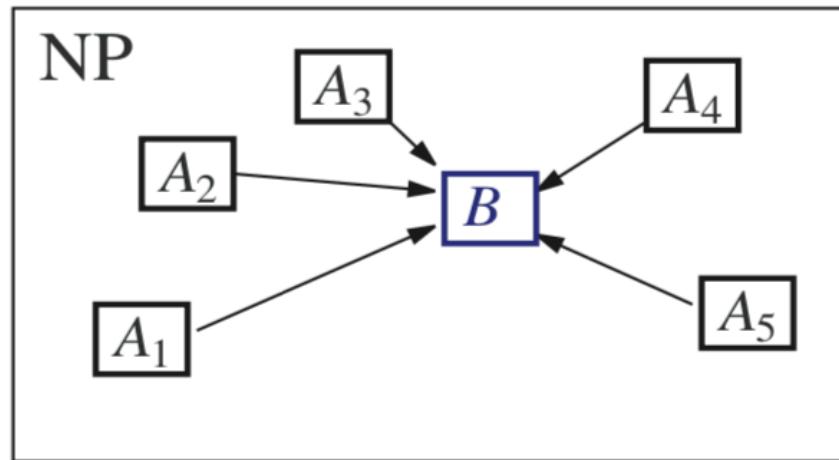
Proof.

- $B \in P \Rightarrow \exists$ TM M that is polynomial-time decider for B .
- $A \leq_P B \Rightarrow \exists$ function f that reduces A to B in polynomial time.
- Define TM N that decides A as follows:
 N = “On input w ,
 1. Compute $f(w)$.
 2. Run M on input $f(w)$ and output whatever M outputs.”
- **Analysis of Time Complexity of TM N :**
 - Each stage runs once.
 - Stage 1 is polynomial because f is polynomial-time function.
 - Stage 2 is polynomial because M is polynomial-time decider for B .

NP-Complete

Definition: Language B is **NP-Complete** if

1. $B \in \text{NP}$, and
2. For every language $A \in \text{NP}$, we have $A \leq_P B$.



Remarks:

- NP-Complete problems are the most difficult problems in NP.
- If we omit first requirement, then we say that B is **NP-Hard**.

Cook-Levin Theorem

- Once we have one NP-Complete problem, can identify others by using polynomial-time reduction (Theorem 7.36).
- But identifying the first NP-Complete problem requires some effort.
- Recall **satisfiability problem**:

$$SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean function} \}$$

Theorem 7.37

SAT is NP-Complete.

Proof Idea:

- $SAT \in NP$ because a polynomial-time NTM can guess assignment to formula ϕ and *accept* if assignment satisfies ϕ .
- Show that for every language $A \in NP$, we have $A \leq_P SAT$.

3SAT is NP-Complete

Recall

$$3SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-function} \}$$

Corollary 7.42

$3SAT$ is NP-Complete.

Proof Idea:

Can modify proof that SAT is NP-Complete (Theorem 7.37) so that resulting Boolean function is a 3cnf-function.

NP-Hard Optimization Problems

- Decision problems have YES/NO answers.
- Many decision problems have corresponding **optimization** version.
- Optimization version of NP-Complete problems are NP-Hard.

Problem	Decision Version	Optimization Version
<i>CLIQUE</i>	Does a graph G have a clique of size k ?	Find largest clique
<i>ILP</i>	Does \exists integer vector y such that $Ay \leq b$?	Find integer vector y to $\max d^\top y$ s.t. $Ay \leq b$
TSP	Does a graph G have tour of length $\leq d$?	Find min length tour
Scheduling	Given set of tasks and constraints, can we finish all tasks in time d ?	Find min time schedule

Why are NP-Complete and NP-Hard Important?

- Suppose you are faced with a problem and you can't come up with an efficient algorithm for it.
- If you can prove the problem is NP-Complete or NP-Hard, then there is no known efficient algorithm to solve it.
 - No known polynomial-time algorithms for NP-Complete and NP-Hard problems.
- How to deal with an NP-Complete or NP-Hard problem?
 - Approximation algorithm
 - Probabilistic algorithm
 - Special cases
 - Heuristic e.g. Branch and bound search

Bounded approximation algorithms

- For a given instance I of the minimization optimization problem π , let $\text{OPT}(I)$ be the optimum value.
- Given an approximation algorithm A for π , let $A(I)$ be the solution given by A for instance I .
- Define performance ratio

$$R_A(I) = \frac{A(I)}{\text{OPT}(I)}$$

For a maximization problem π , define $R_A(I) = \frac{\text{OPT}(I)}{A(I)}$

- Absolute performance ratio R_A is given by smallest value $r \geq 1$ such that $R_A(I) \leq r$ for all instances I of π .

Travelling Salesman Problem

- Triangular inequality constraint:
The graph satisfies the property that
$$w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k) \text{ for any 3 distinct vertices } v_i, v_j \text{ and } v_k.$$
- There is an approximation alg. A for TSP with triangular inequality that achieves $R_A < 2$.
- It constructs an Euler tour of the minimum cost spanning tree of the graph and applies triangular inequality to find a TSP tour.

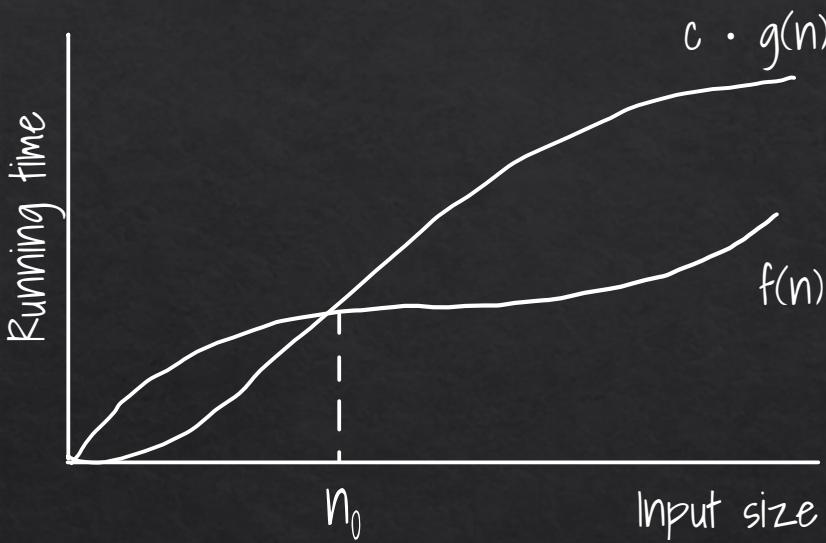
Approximation Schemes

- An approximation algorithm A which given an accuracy requirement $\varepsilon > 0$ and an instance I of the problem constructs a solution in polynomial time and achieves $R_{A_\varepsilon}(I) \leq 1 + \varepsilon$ (i.e. a range of app. algorithms one for each ε)
- A fully polynomial-time approximation scheme runs in time which is a polynomial function of the length of the input and $\frac{1}{\varepsilon}$
It has time-accuracy trade-offs.

Knapsack problem

- **Problem :** $\text{Max } \sum_{i=1}^n b_i t_i$
s.t. $\sum_{i=1}^n w_i t_i \leq W, t_i = 0 \text{ or } 1, \forall i$
- Decision version is NP-Complete (Can reduce Partition problem to Knapsack)
- Note the DP algorithm has time complexity $O(nW)$. This is NOT a polynomial-time algorithm as length of input includes $\log W$ and time complexity is $O(n 2^{\log W})$. It is called a “pseudo-polynomial time” algorithm.
- By scaling the problem depending on accuracy ε , we can design an approximation scheme for knapsack problem that works in time $O(\frac{n^2}{\varepsilon})$

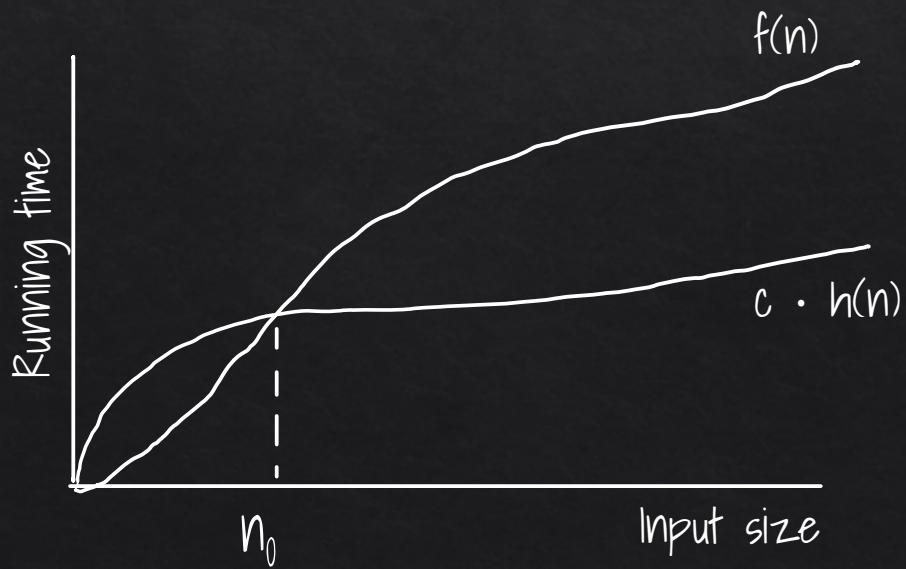
Review of Big Oh Notations



$$f(n) \leq c \cdot g(n)$$

→

$$f(n) \in O(g(n))$$

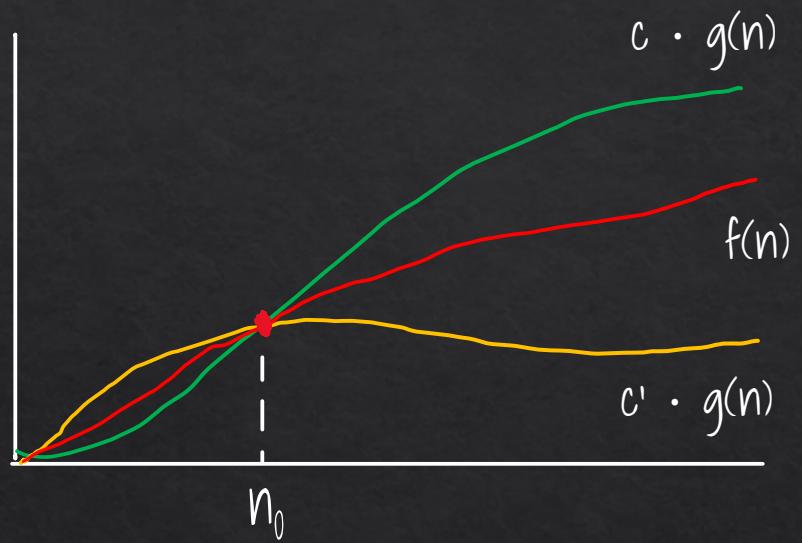


$$f(n) \geq c \cdot h(n)$$

→

$$f(n) \in \Omega(h(n))$$

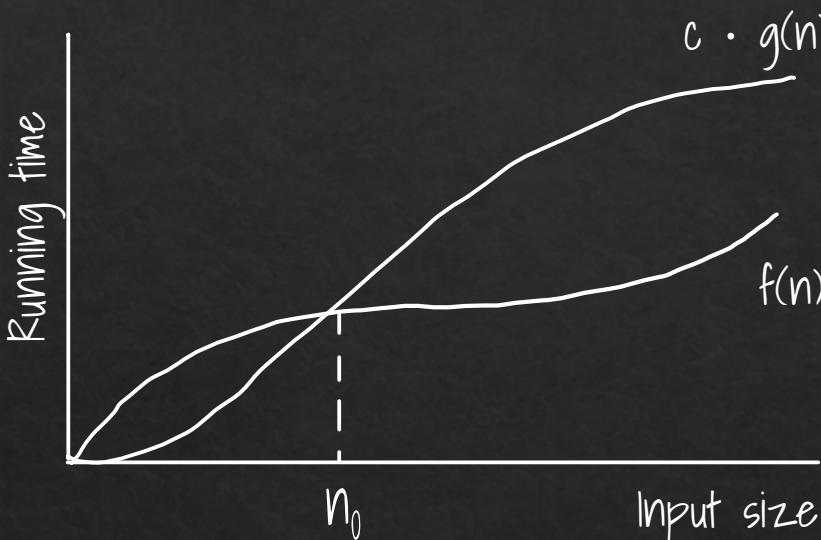
Review of Big Oh Notations



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

$f(n)$ and $g(n)$ are asymptotically equal, up to a constant factor.

Review of Big Oh Notations

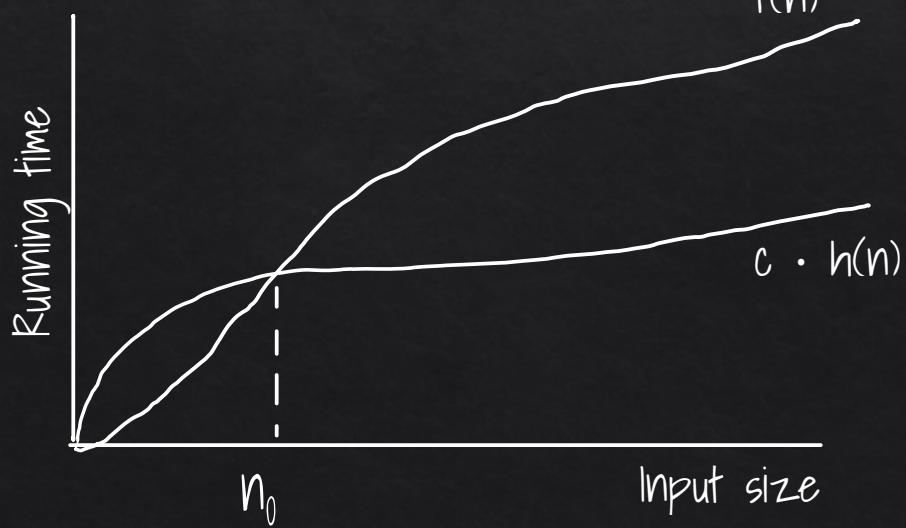


$$f(n) \leq c \cdot g(n) \rightarrow f(n) \in O(g(n)) \quad \exists c > 0 \text{ s.t. } \exists n_0 \geq 1$$

$$\forall c > 0 \text{ s.t. } \exists n_0 > 0 \rightarrow f(n) \in o(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \quad (\text{less than in asymptotic sense})$$

Approaches but never touches



$$f(n) \geq c \cdot h(n) \rightarrow f(n) \in \Omega(h(n)) \quad \exists c > 0 \text{ s.t. } \exists n_0 \geq 1$$

$$\forall c > 0 \text{ s.t. } \exists n_0 > 0 \rightarrow f(n) \in \omega(h(n))$$

Little Oh and Little Omega

Example:

Prove that $f(n) = 6n^2 + 3n$ is $o(n^3)$ and $\omega(n)$

Solution:

The ask: $f(n) \leq c \cdot n^3$ and $f(n) \geq c \cdot n$ for $\forall c > 0$ and $\exists n_0 > 0$

Show $f(n)$ is $o(n^3)$: Let $c > 0$, pick $n_0 = (6+3)/c = 9/c \rightarrow cn \geq 9$ for $n \geq n_0$

$$f(n) = 6n^2 + 3n \leq 6n^2 + 3n^2 = 9n^2 \leq cn \cdot n^2 = cn^3 \text{ for } n \geq n_0$$

Show $f(n)$ is $\omega(n)$: Let $c > 0$, pick $n_0 = c/6 \Rightarrow 6n \geq c$ for $n \geq n_0$

$$f(n) = 6n^2 + 3n \geq 6n^2 = 6n \cdot n \geq cn$$

Main Topic – Huffman Coding

Text Compression Goal:

Store as many documents as possible

16 bits (e.g. 0101010101010101) -> Same doc = 16Megabits

Fact:

ASCII & Unicode system use fixed-length binary strings to encode characters

7 bits (e.g. 0101010) -> Doc of 100M characters = $100M \times 7 = 7\text{Megabits}$

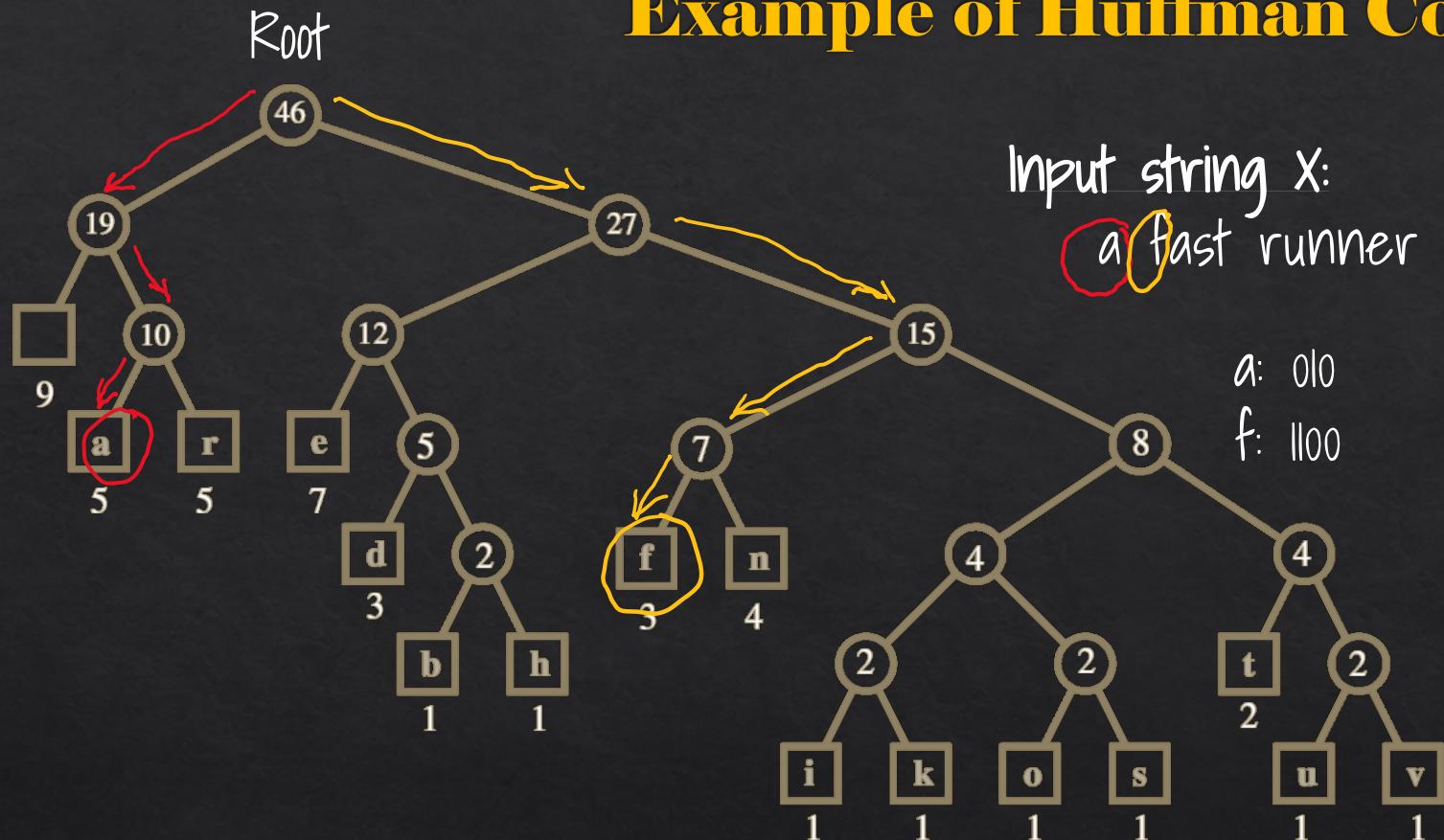
Innovate:

Achieve text compression goal using a **variable-length encoding scheme**

Core Idea:

Most-frequently used characters to use the least numbers of bits and vice versa.

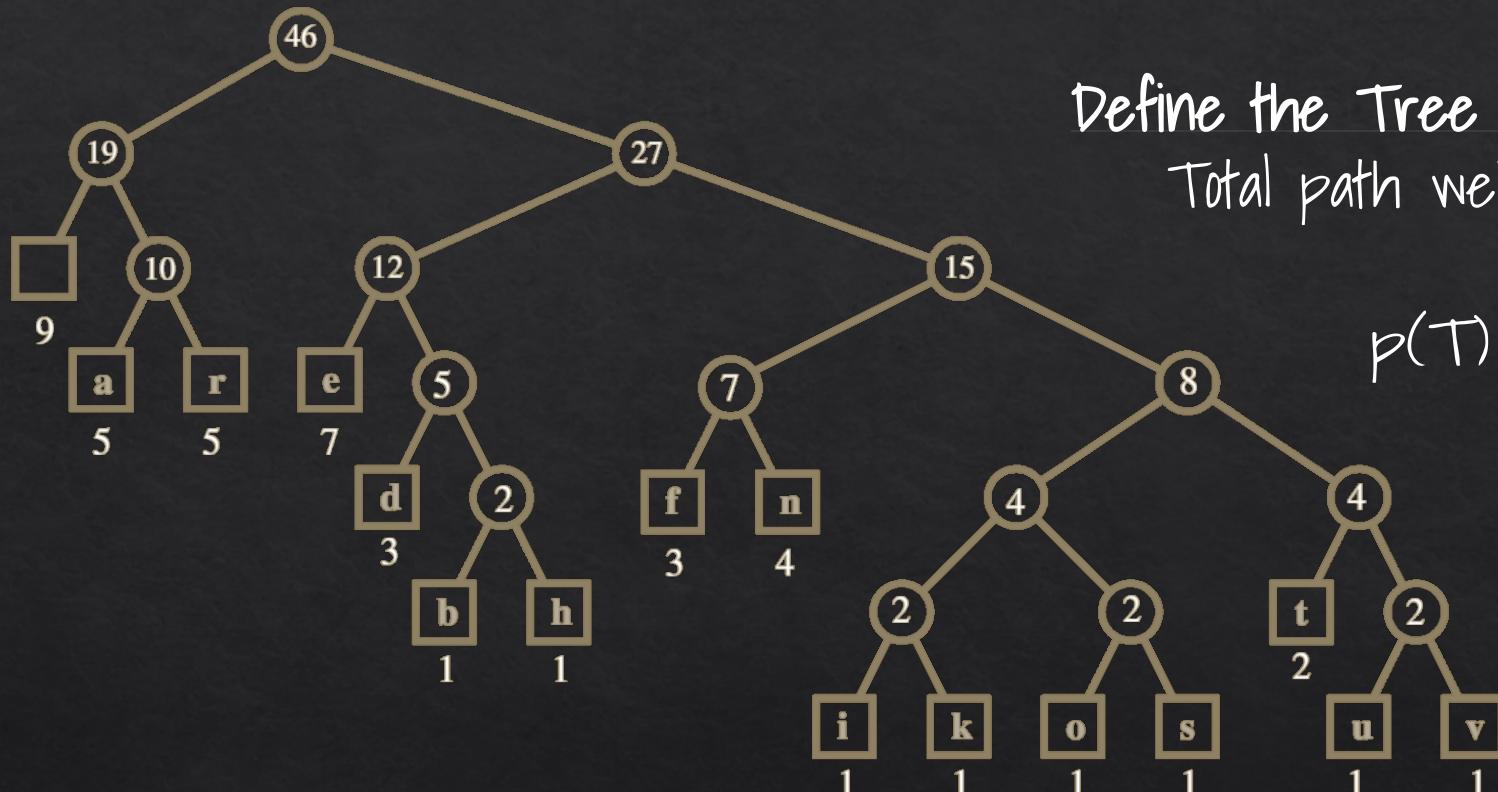
Example of Huffman Coding



- Representation Convention:
- Circle: internal
 - Square: external
 - v: a vertex/node
 - C: a character
 - C: collection of characters

<i>Character</i>		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
<i>Frequency</i>	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

Example of Huffman Coding



* Depth of a node = number of its proper ancestors.

Define the Tree T:

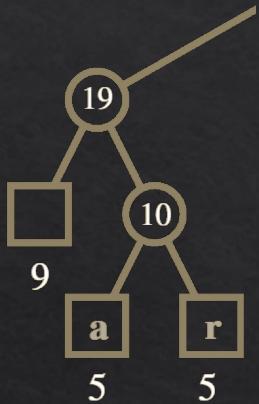
Total path weight: $p(T)$

$$p(T) = \sum_{v \in T} f(v) = \sum_{c \in C} f(c) \cdot d(v_c),$$

- v_c : external node asso. w. c
- $d_T(v_c)$ or $d(v_c)$: depth of v_c in T
- $f(c)$: frequency of c
- $f(v_c)$: sum of the $f(c)$ for all v_c who are its descendants.

c	a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
$f(c)$	9	5	1	3	7	3	1	1	4	1	5	1	2	1	1
$d(v_c)$	2	3	5	4	3	4	5	6	6	4	6	3	6	5	6

Example of Huffman Coding



Define the Tree T:

Total path weight: $p(T)$

$$p(T) = \sum_{v \in T} f(v) = \sum_{c \in C} f(c) \cdot d(v_c),$$

$$19 + 10 + 9 + 5 + 5 = 9 \times 2 + 5 \times 3 + 5 \times 3 = 48$$

- v_c : external node asso. w. c
- $d_T(v_c)$ or $d(v_c)$: depth of v_c in T
- $f(c)$: frequency of c
- $f(v_c)$: sum of the $f(c)$ for all v_c who are its descendants.

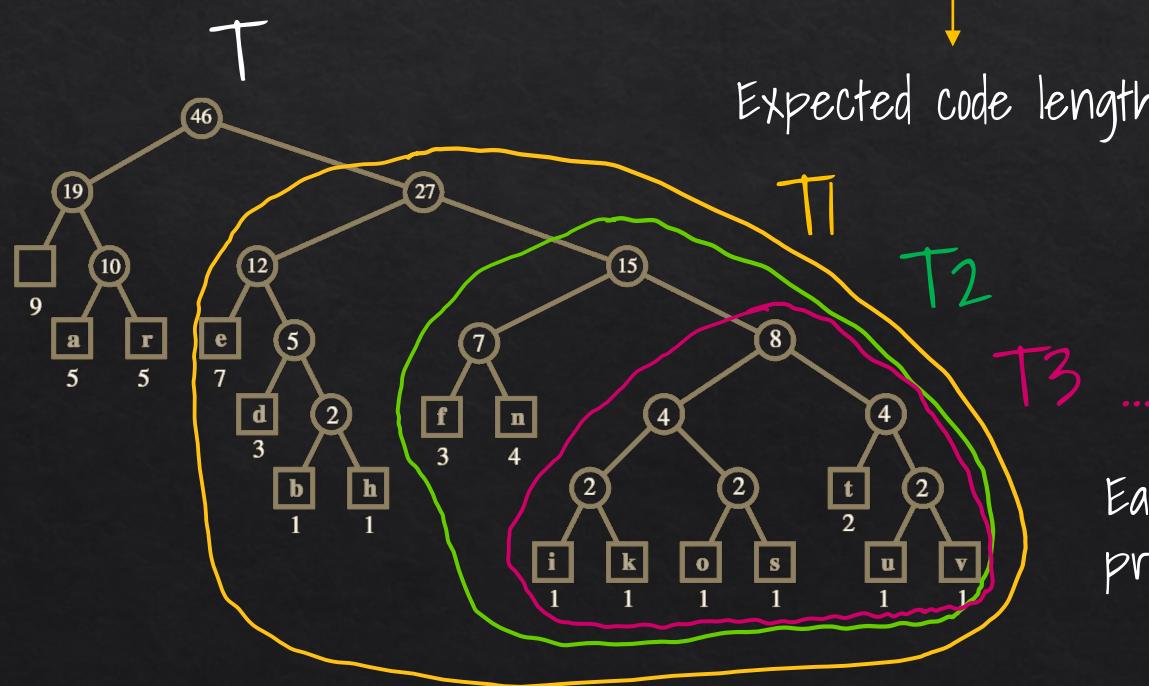
* Depth of a node = number of its proper ancestors.

c	a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
$f(c)$	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1
$d(v_c)$	2	3	5	4	3	4	5	6	6	4	6	3	6	5	6

Construct Optimal Tree - Huffman Coding

Optimal Tree Structure T:

$$p(T) = \sum_{v \in T} f(v) = \sum_{c \in C} f(c) \cdot d(v_c),$$



Expected code length. Want to minimize this

Each of the subset is the optimal for the subproblem of coding the subset

Example of Huffman Coding

Optimal Tree Structure T starts with its subtree:

$$\sum_{c \in S_{\pi}} f(c) \cdot d_T(v_c) = \sum_{c \in S_{\pi}} f(c) \cdot [d_T(v_{r_{T_1}}) + d_{T_1}(v_c)] = \sum_{c \in S_{\pi}} f(c) \cdot d_T(v_{r_{T_1}}) + \sum_{c \in S_{\pi}} f(c) \cdot d_{T_1}(v_c),$$

Depth of root node of T_1 (as in T) + depth all T_1 external nodes in T



only worry about this subtree T_1 (independent of T)

- $v_{r_{T_1}}$: Root node of T_1
- $f(c)$: weight of c (proportionate w/
 $f(v)$)
- $d(v_c)$: depth of v_c

Review of Priority Q

Abstract Structure Definition

A container of elements, each is assigned a key (at insertion time). This key will determine the "priority" used to pick elements to be removed (tends to be the min)

Priority Queue Implementations:

1. Heap (A binary tree whose elements stored at internal nodes only.)
2. Unsorted list (the key = value of elements)
3. Sorted list (the key = order in the list)
 - Linked list can be used to construct sorted list

All 1,2,3 can be array-based

Method	Unsorted List	Sorted List	Heap
Insert	O(1)	O(n)	O(logn)
Remove(min priority)	O(n)	O(1)	O(logn)
Access(key)	O(1)	O(1)	O(1)

Array Index	1	2	3	4	5	6	7	8	9	10	11
Unsorted List	2	3	13	5	9	15	16	11	17	18	
Sorted list	2	3	5	9	11	13	15	16	17	18	
Min Heap	2	3	13	5	9	15	16	11	17	18	
Max Heap	18	17	15	11	16	3	13	2	9	5	

Insertion order: 2,3,13,5,9,15,16,11,17,18

Algorithm Huffman

Algorithm Huffman(C):

Input: A set, C , of d characters, each with a given weight, $f(c)$

Output: A coding tree, T , for C , with minimum total path weight

Initialize a priority queue Q .

for each character c in C do

 Create a single-node binary tree T storing c .

 Insert T into Q with key $f(c)$.

$O(n)$

Complexity: $O(n + d \log d)$

- d : number of distinct c in X (size of C)

while $Q.size() > 1$ do

$f_1 \leftarrow Q.\text{minKey}()$

$T_1 \leftarrow Q.\text{removeMin}()$

$f_2 \leftarrow Q.\text{minKey}()$

$T_2 \leftarrow Q.\text{removeMin}()$

 Create a new binary tree T with left subtree T_1 and right subtree T_2 .

 Insert T into Q with key $f_1 + f_2$.

$O(\log d)$

$O(d \log d)$

return tree $Q.\text{removeMin}()$

Algorithm 10.9: Huffman coding algorithm.

Greedy Choice Property – Huffman Coding Algorithm

1-level tree containing 2 lowest frequency characters is part of an optimal tree

Consider a set $C = \{c_1, c_2, c_3, c_4\}$ of 4 characters with frequencies

$$f_1 \leq f_2 \leq f_3 \leq f_4.$$

Need to consider only 2 complete tree structures.

Case 1 : 2 nodes (say x and y with $f(x) \leq f(y)$) at level 3 and other nodes (w and z) at level 1 and 2

expected code length $e = f(w) + 2 f(z) + 3(f(x)+f(y))$

If $w=c_2$ & $f_2 \leq f(y) \rightarrow$ can get a new tree T' by switching w and y with expected frequency

$$e' = f(y) + 2 f(w) + 3(f(x)+f(z))$$

$e' - e = 2(f(w)-f(y)) \leq 0$ as $f(y) \geq f(w) \rightarrow$ Change to T' w. smaller expected code length

Same is true for $z \rightarrow$ In optimal tree, $\text{freq}(y) = f_2$. Can also show $f(x) = f_1$

Greedy Choice Property – Huffman Coding Algorithm

l-level tree containing 2 lowest frequency characters is part of an optimal tree

Consider a set $C = \{c_1, c_2, c_3, c_4\}$ of 4 characters with frequencies

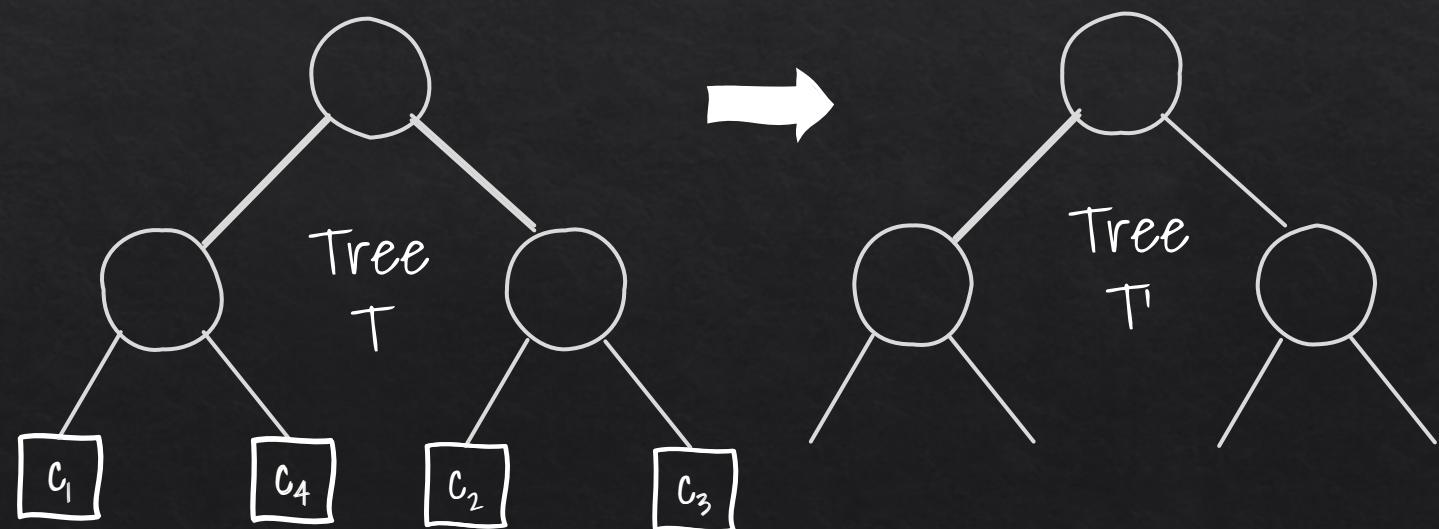
$$f_1 \leq f_2 \leq f_3 \leq f_4.$$

Need to consider only 2 complete tree structures.

Case 2 : All leaves at same level

Show easily we can switch leaves so that c_1, c_2 are in same subtree.

\Rightarrow l-level subtree containing c_1, c_2 , is part of an optimal tree.



Knapsack Problem Example

NP-complete problem. Solved using Greedy Algorithm via dynamic programming: Solution is not the optimal solution

Example:

$C = 11$, 5 objects with weights & values as follows.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$												
$w_2 = 2 v_2 = 6$												
$w_3 = 5 v_3 = 18$												
$w_4 = 6 v_4 = 22$												
$w_5 = 7 v_5 = 28$												

$$\max(F_{i+1}(C), v_i + F_{i+1}(C - w_i))$$

Previous Round Result @
the same capacity c

Current (new item's) value +
Previous Round Result @
capacity $c - w_i$

Dynamic Programming

Sorted item by weight

Round 1: only consider object 1 with w_1 and v_1
 Total weight of all objects = 1

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0											
$w_3 = 5 v_3 = 18$	0											
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

$$\max(F_{i+1}(C), V_i + F_{i+1}(C - w_i))$$

Previous Round
Result @ the
same capacity c

Current (new item's)
value + Previous Round
Result @ capacity $c - w_i$

Round 2: only consider object 1 & 2
 Total weight of all objects = 3

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5 v_3 = 18$	0											
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

$$\max(F_3(2), V_2 + F_3(2-2))$$

$$= \max(1, 6 + 0) = 6$$

Fill out the rest same as case
where max capacity = 3 has
reached

$$\max(F_3(3), V_2 + F_3(3-1))$$

$$= \max(1, 6 + 1) = 7$$

Dynamic Programming

Sorted item by weight

Round 2: only consider object 1,2

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5 v_3 = 18$	0											
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

$$\max(F_{i+1}(C), V_i + F_{i+1}(C - w_i))$$

Previous Round Result @ the same capacity C

Current (new item's) value + Previous Round Result @ capacity C - w_i

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5 v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

Round 3: only consider object 1,2,3

Keep them the same
Since $w_3 = 5 > \text{current capacity}$

$$C = 7, i = 3, w_3 = 5$$

$$\max(F_4(7), V_3 + F_4(7-5))$$

$$= \max(7, 18 + 6) = 24$$

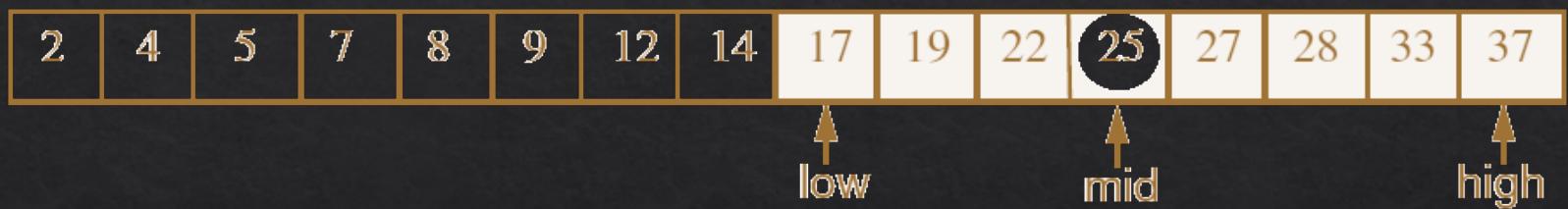
$$C = 5, i = 3, w_3 = 5$$

$$\max(F_4(5), V_3 + F_4(5-5))$$

$$= \max(7, 18 + 0) = 18$$

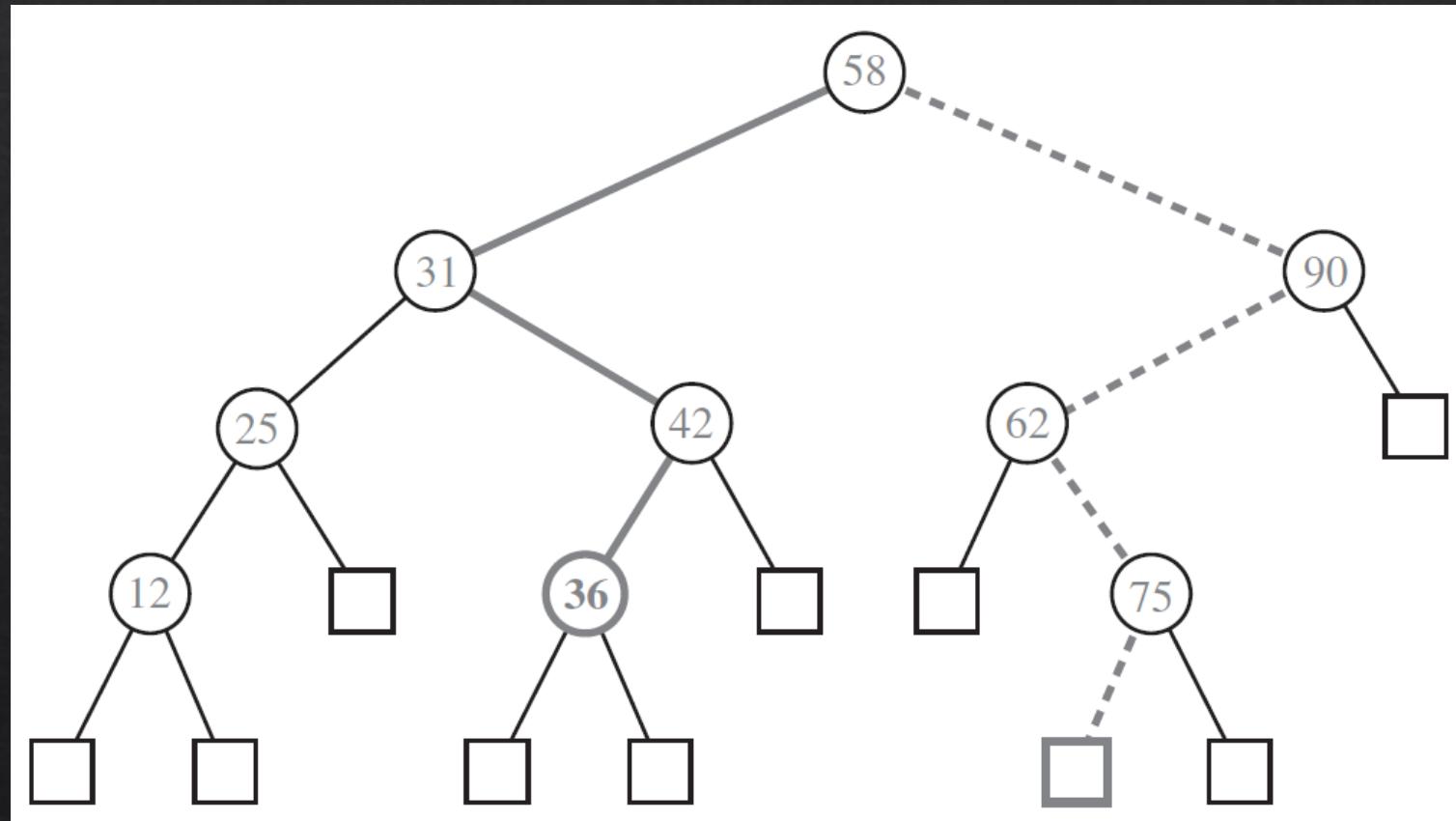
Binary Search

❖ Sorted array



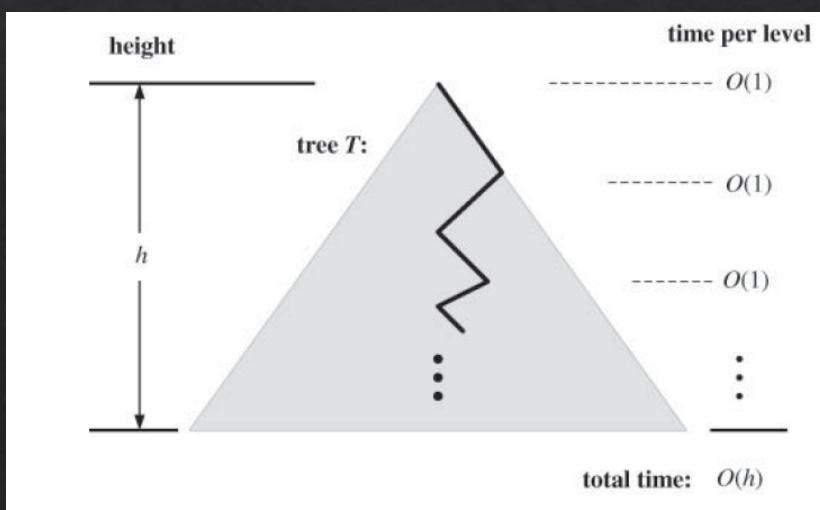
$$T(n) \leq \begin{cases} b & \text{if } n < 2 \\ T\left(\frac{n}{2} + b\right) & \text{else,} \end{cases}$$

Review of Binary Search Tree



❖ Tree-based structure

Height of Binary Search Tree

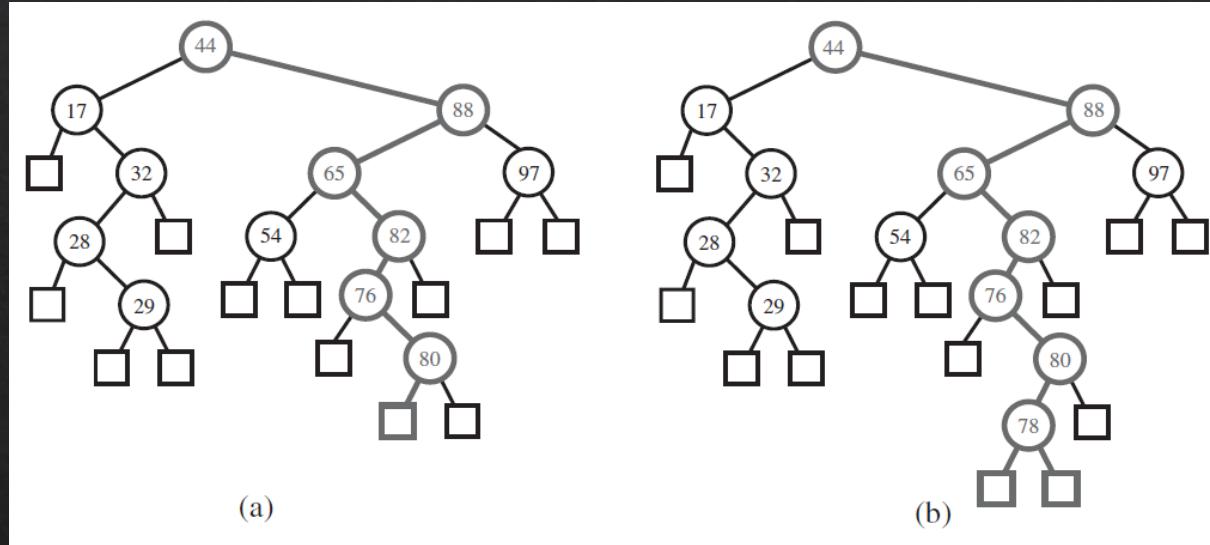


- ◆ Height of $O(\log n)$ High probability to build such tree.
- ◆ Children stores $3A$ keys (and items) \rightarrow Balanced bt

$$H(n) \leq \begin{cases} 1 & \text{if } n < 2 \\ H\left(\frac{3}{4}n + 1\right) & \text{else,} \end{cases}$$

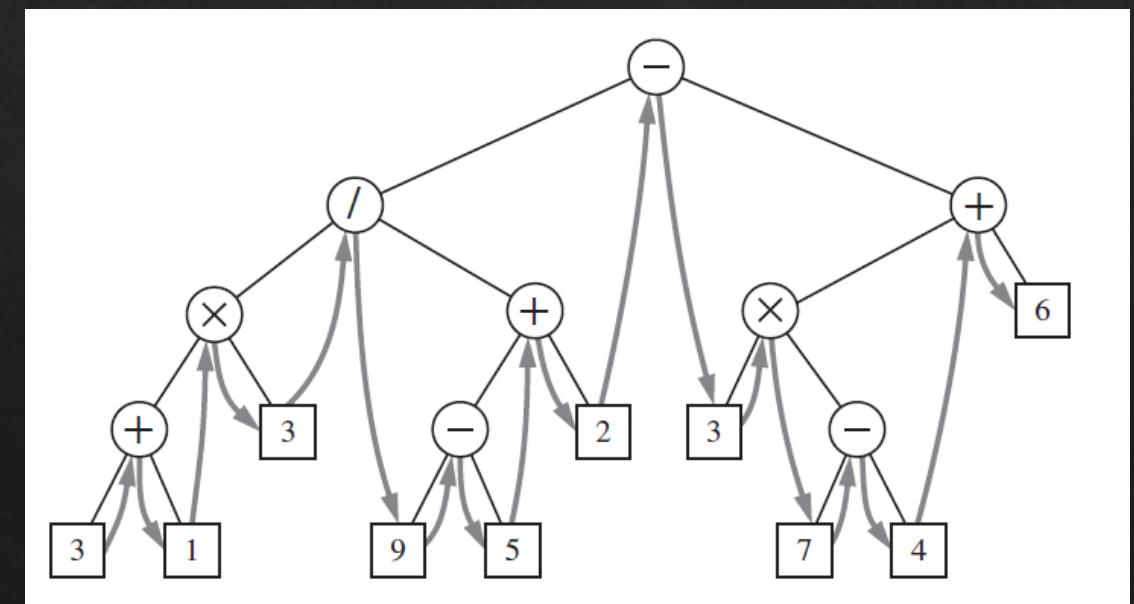
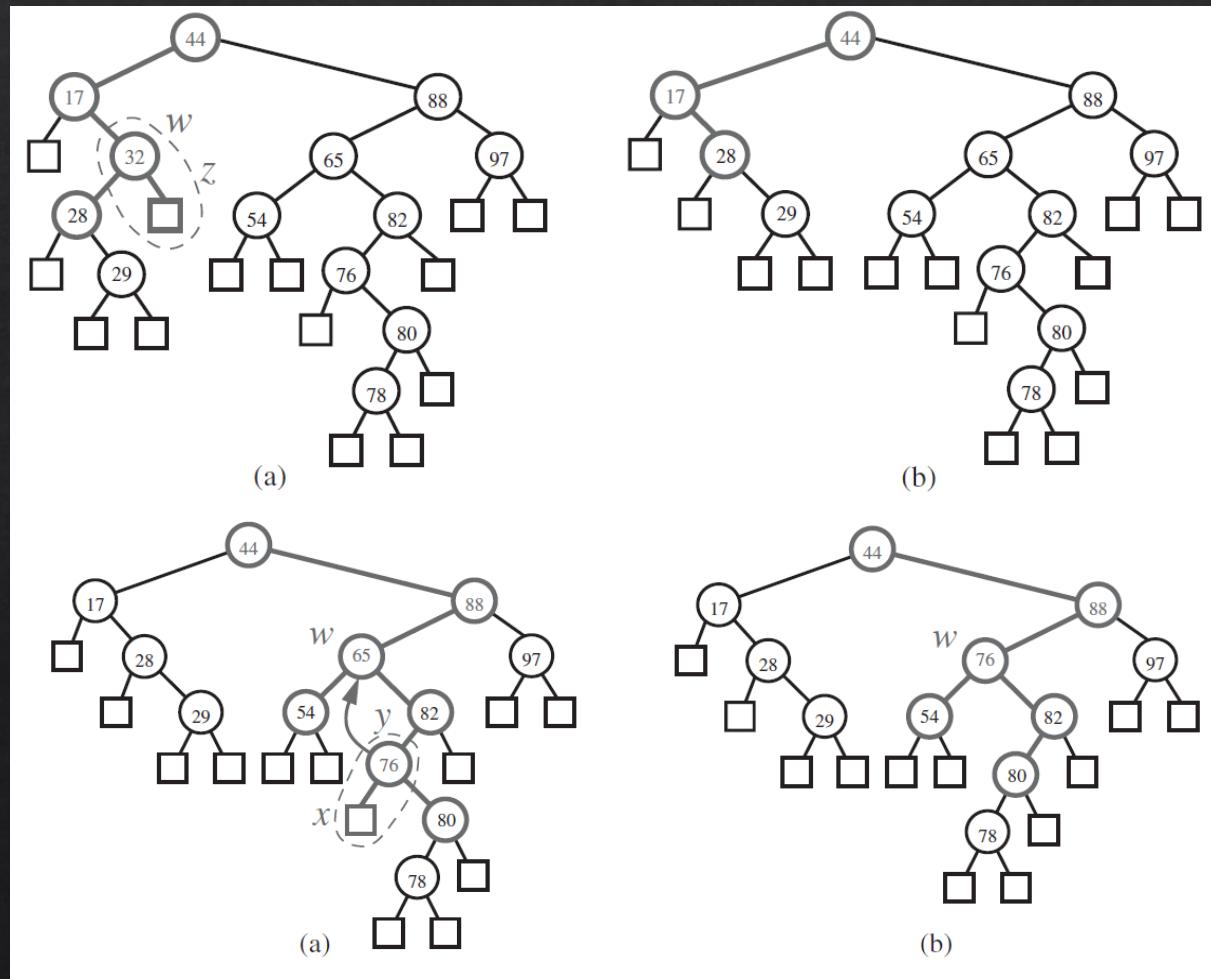
- ◆ $H(n)$ is ceiling of $(\log_{4/3} n)$

Insertion - Binary Search Tree

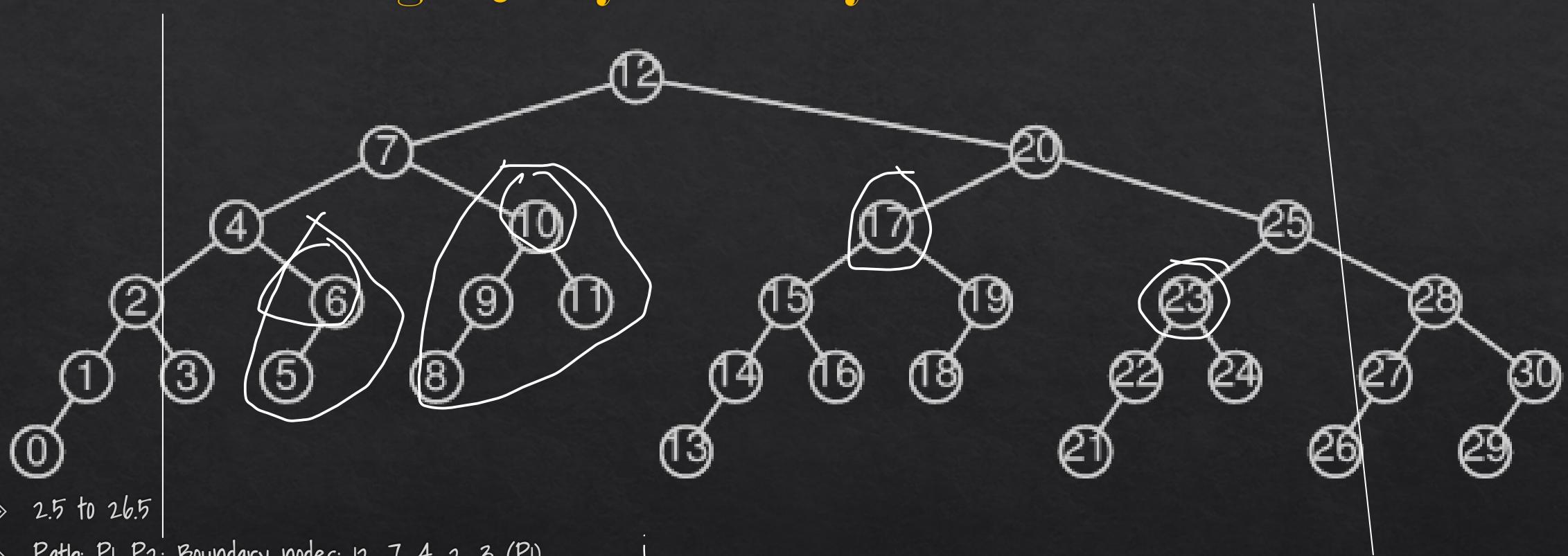


❖ $O(\log h)$

Deletion in Binary Search Tree



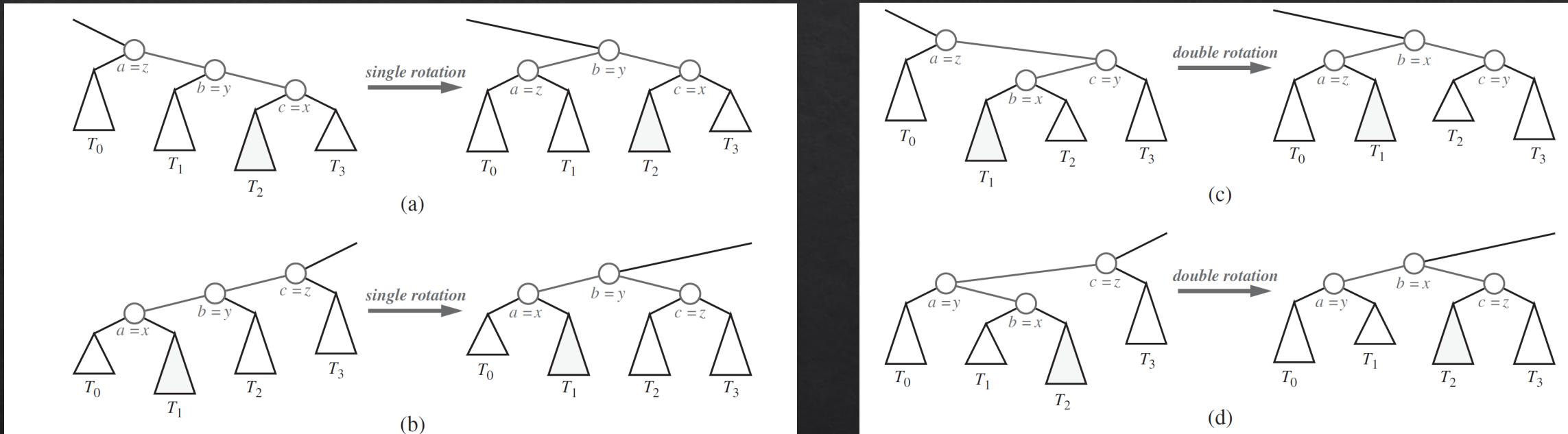
Range Query in Binary Search Tree



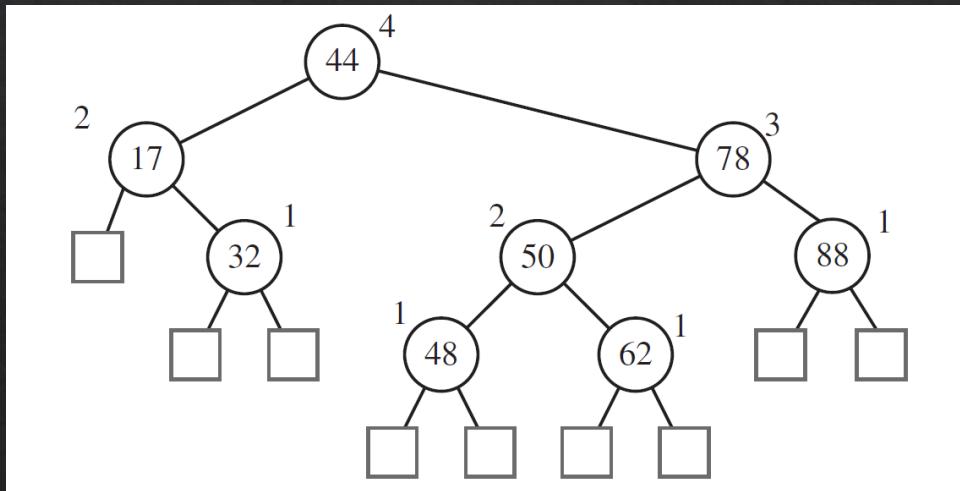
- ◊ 2.5 to 26.5
- ◊ Path: P₁, P₂: Boundary nodes: 12, 7, 4, 2, 3 (P₁)
- ◊ 20, 25, 28, 27, 26 (P₂)
- ◊ Inside nodes: > 2.5, < 26.5
- ◊ Outside nodes

$$\sum_{i=1}^j (2s_i + 1) = 2s + j \leq 2s + 2h$$

Trinode restructure(Balanced Binary Tree)

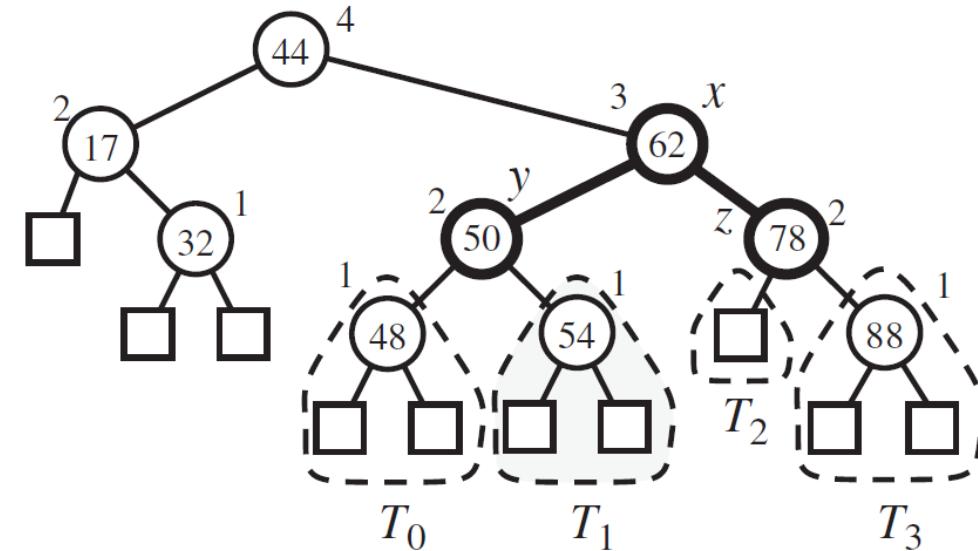
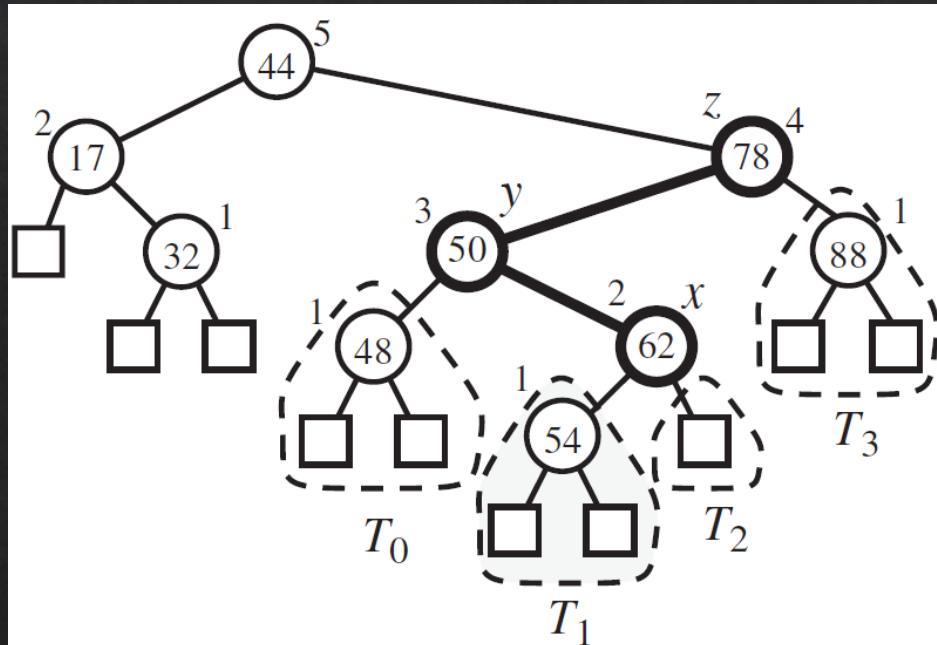


AVL Tree

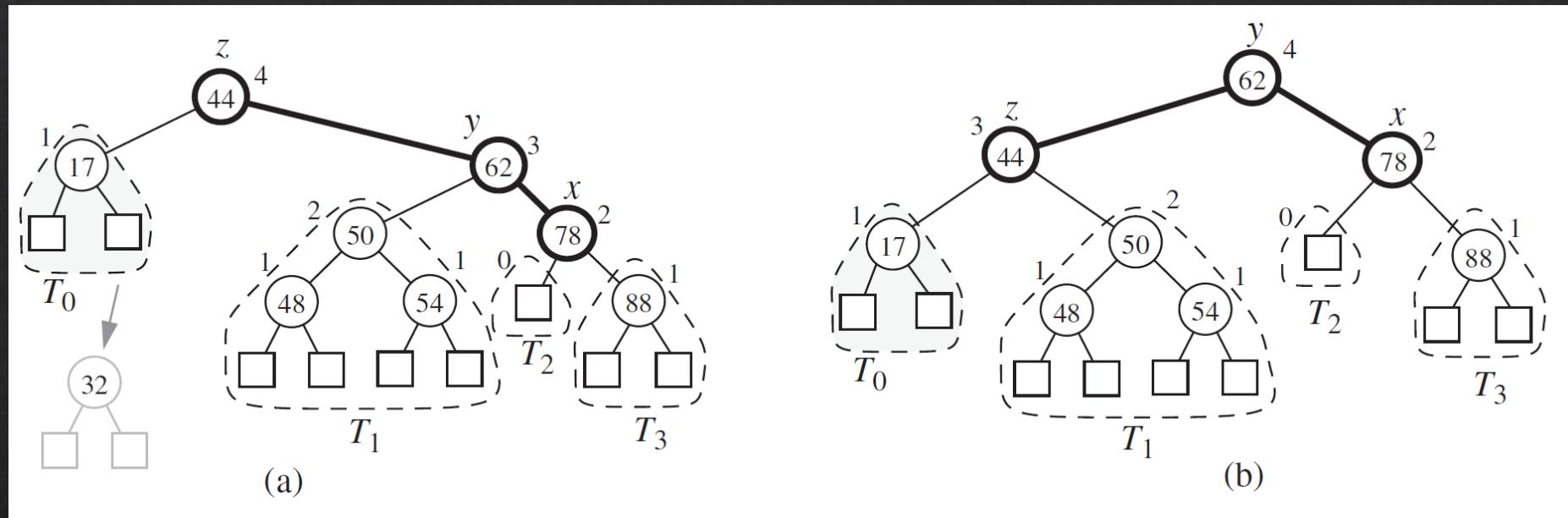


- ◆ Named after its inventors: Adel'son-Vel'ski and Landis (1962)
- ◆ x, y are siblings: $|r(x) - r(y)| \leq 1$
- ◆ Rank of a node: bottom up (excluding external node)

Insertion in AVL Tree



Removal in AVL Trees



Knapsack Problem Example

NP-complete problem. Solved using Greedy Algorithm via dynamic programming: Solution is not the optimal solution

Example:

$C = 11$, 5 objects with weights & values as follows.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$												
$w_2 = 2 v_2 = 6$												
$w_3 = 5 v_3 = 18$												
$w_4 = 6 v_4 = 22$												
$w_5 = 7 v_5 = 28$												

$$\max(F_{i+1}(C), V_i + F_{i+1}(C - w_i))$$

Previous Round Result @
the same capacity c

Current (new item's) value +
Previous Round Result @
capacity $c - w_i$

Dynamic Programming

Sorted item by weight

Round 1: only consider object 1 with w_1 and v_1
 Total weight of all objects = 1

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0											
$w_3 = 5 v_3 = 18$	0											
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

$$\max(F_{i+1}(C), V_i + F_{i+1}(C - w_i))$$

Previous Round
Result @ the
same capacity c

Current (new item's)
value + Previous Round
Result @ capacity $c - w_i$

Round 2: only consider object 1 & 2
 Total weight of all objects = 3

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5 v_3 = 18$	0											
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

$$\max(F_3(2), V_2 + F_3(2-2))$$

$$= \max(1, 6 + 0) = 6$$

Fill out the rest same as case
where max capacity = 3 has
reached

$$\max(F_3(3), V_2 + F_3(3-1))$$

$$= \max(1, 6 + 1) = 7$$

Dynamic Programming

Sorted item by weight

Round 2: only consider object 1,2

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5 v_3 = 18$	0											
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

$$\max(F_{i+1}(C), V_i + F_{i+1}(C - w_i))$$

Previous Round Result @ the same capacity C

Current (new item's) value + Previous Round Result @ capacity C - w_i

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1 v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2 v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5 v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6 v_4 = 22$	0											
$w_5 = 7 v_5 = 28$	0											

Round 3: only consider object 1,2,3

Keep them the same
Since $w_3 = 5 > \text{current capacity}$

$$C = 7, i = 3, w_3 = 5$$

$$\max(F_4(7), V_3 + F_4(7-5))$$

$$= \max(7, 18 + 6) = 24$$

$$C = 5, i = 3, w_3 = 5$$

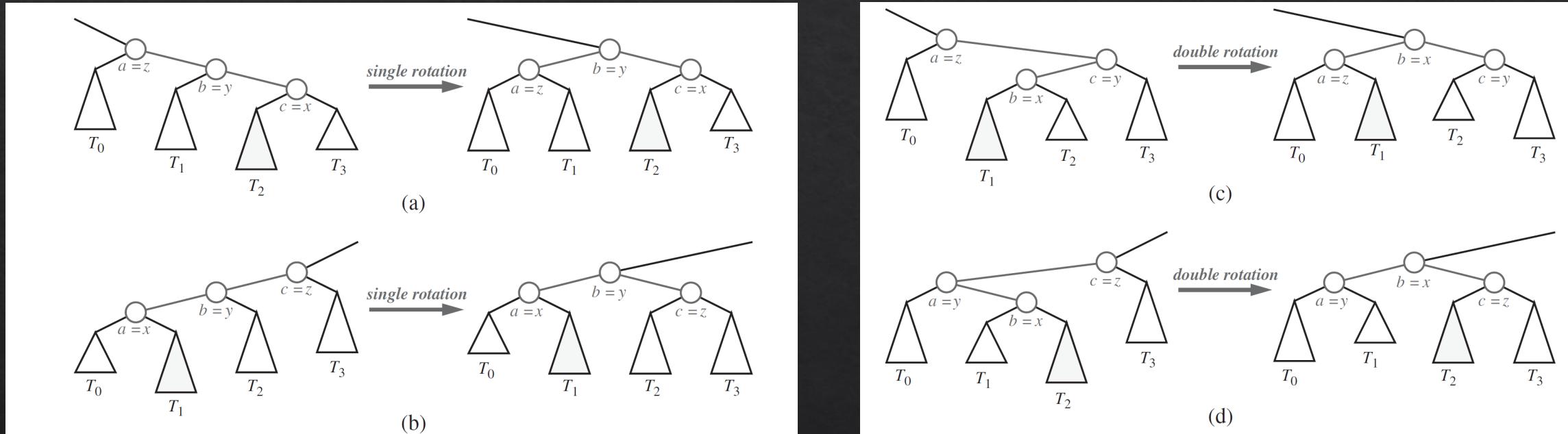
$$\max(F_4(5), V_3 + F_4(5-5))$$

$$= \max(7, 18 + 0) = 18$$

Quiz Quick Review

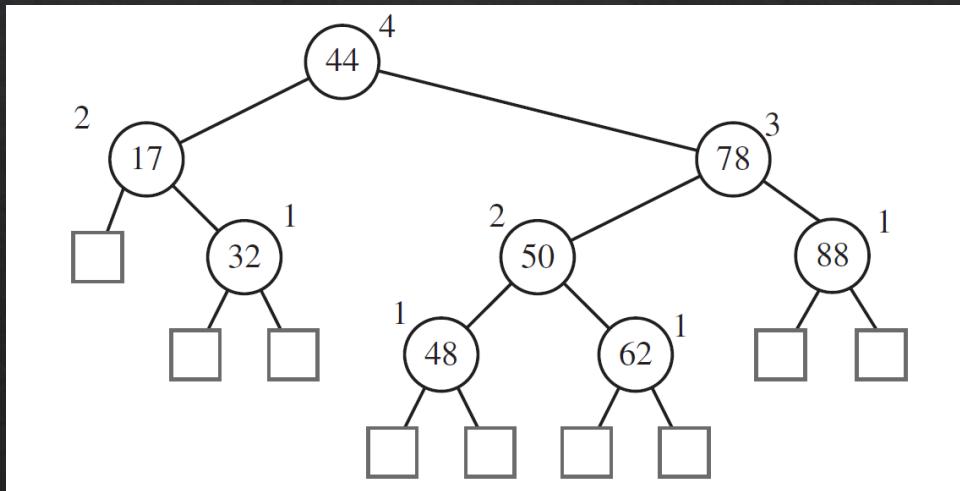
Quick Walkthrough - Does not cover all operations

Trinode restructure(Balanced Binary Tree)



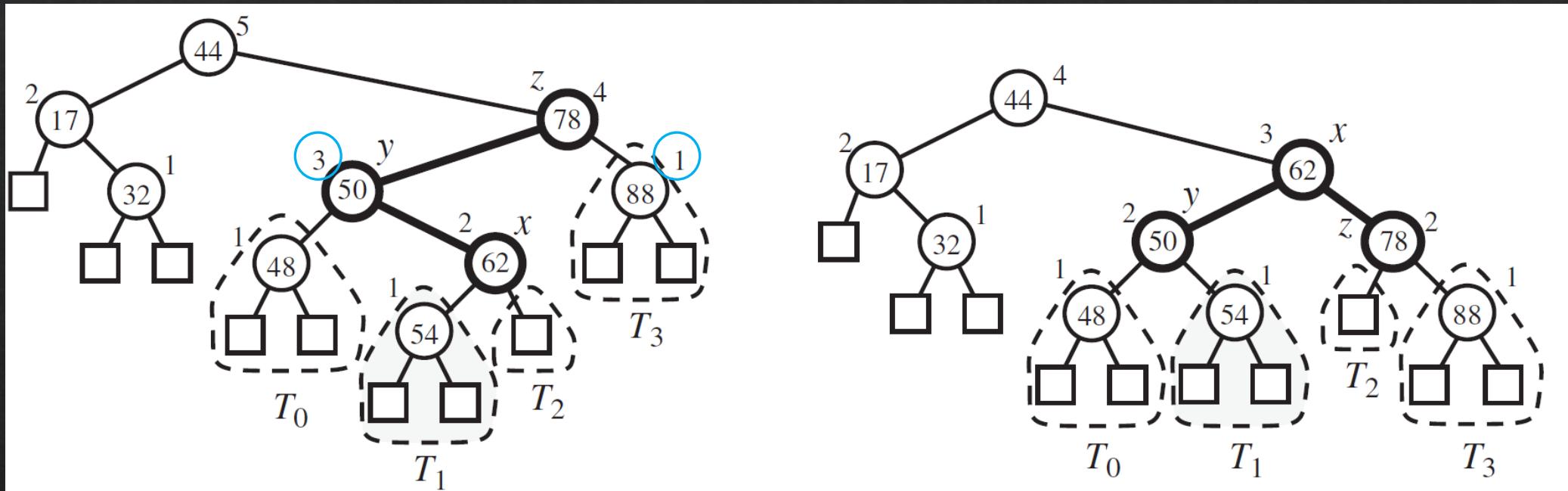
- ❖ Procedure: b to become the parent, gives it children to either a and c . a adopts its left subtree, c adopts its right subtree
- ❖ Count the "shift" of b to become the parent of a & c , if $= 1 \rightarrow$ single rotation, $= 2 \rightarrow$ double rotation

AVL Tree



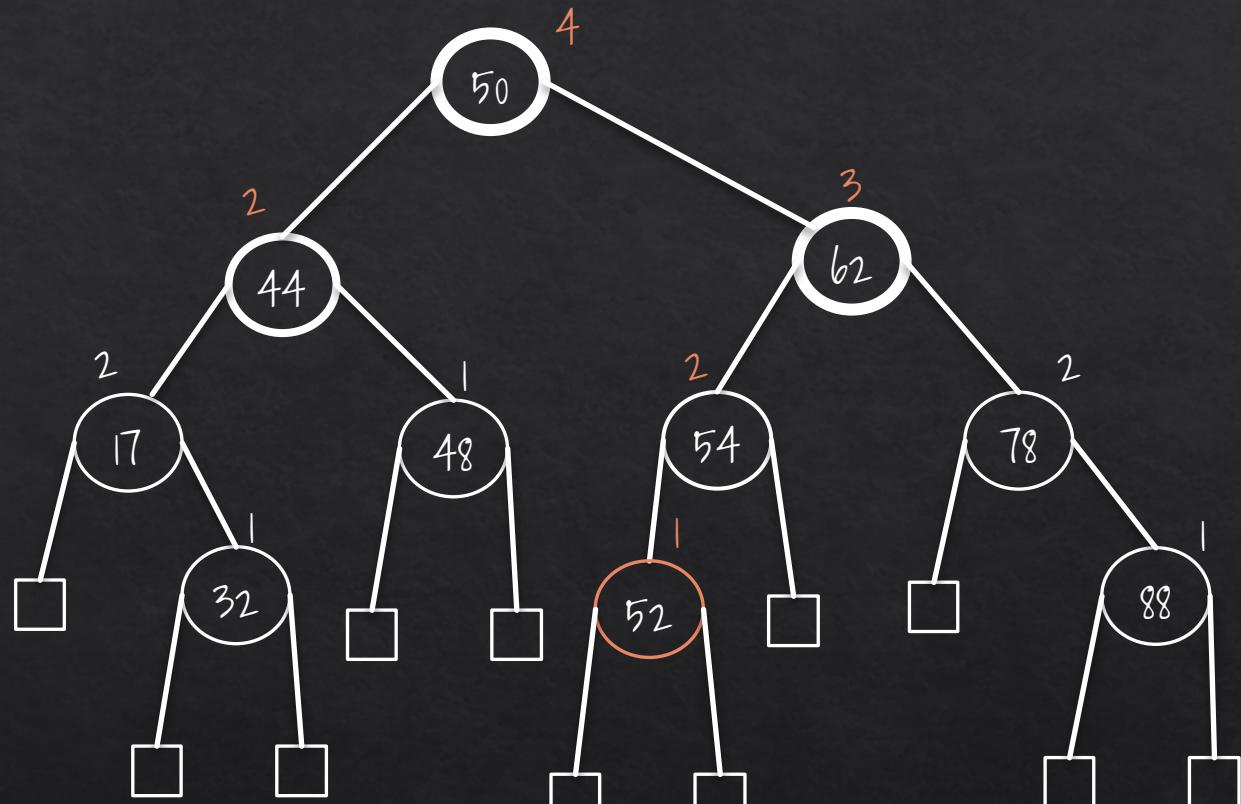
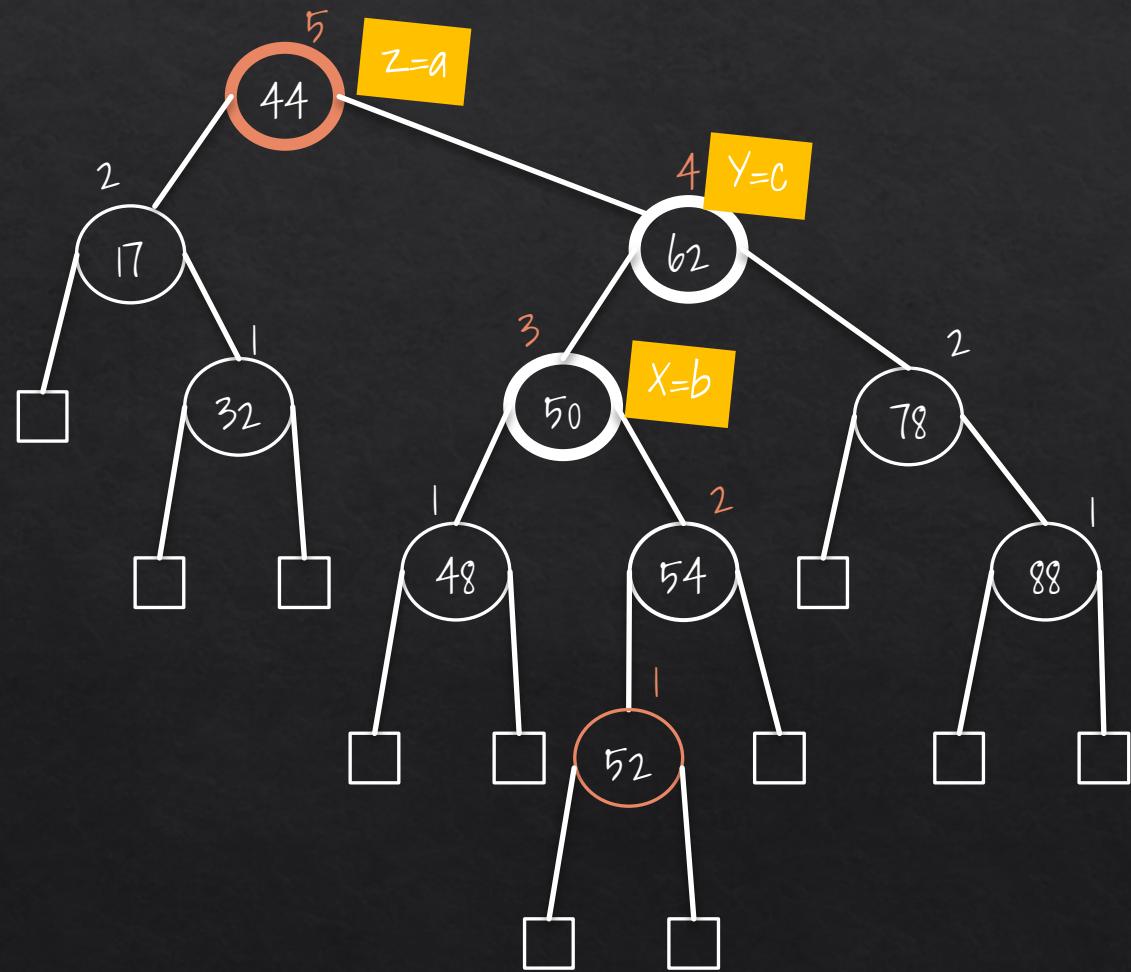
- ◆ Named after its inventors: Adel'son-Vel'ski and Landis (1962)
- ◆ x, y are siblings: $|r(x) - r(y)| \leq 1$
- ◆ Rank of a node: bottom up (excluding external node)

Insertion in AVL Tree



- ❖ Add node key 54 → update ranks → root become unbalanced (left child of rank 2, right child of rank 4)
- ❖ "Search & Repair" strategy: Rebalance subtree rooted at z by calling trinode restructuring:
 - ❖ z is the 1st node we encounter in going up from newly inserted node toward root of T such that z is unbalanced.

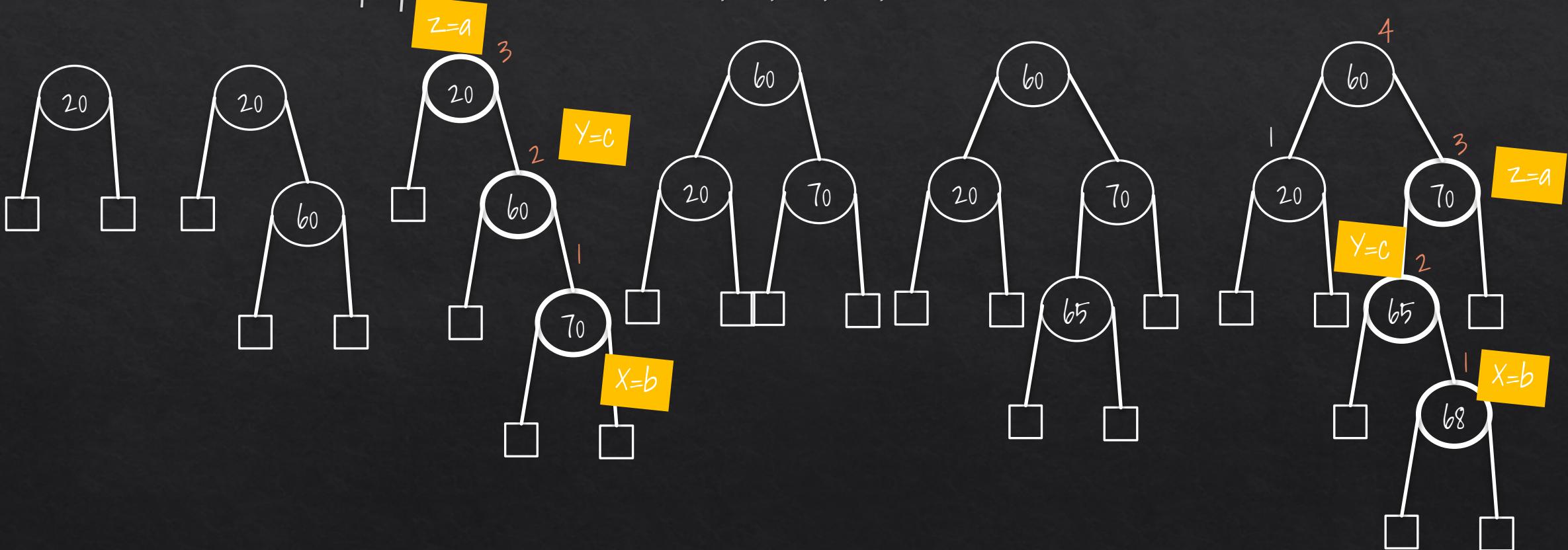
Insertion in AVL Tree



◆ Insertion only needs 1 trinode restructure to rebalance the AVL tree.

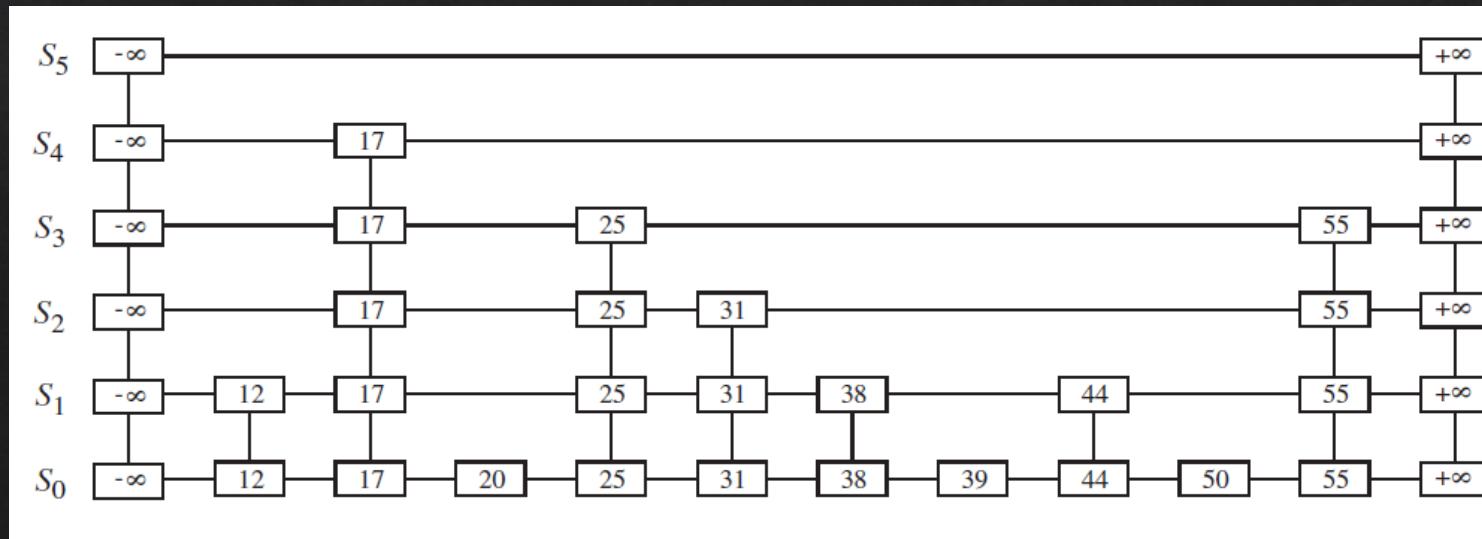
Quiz

◊ Start with an empty AVL tree: Insert 20, 60, 70, 65, 68

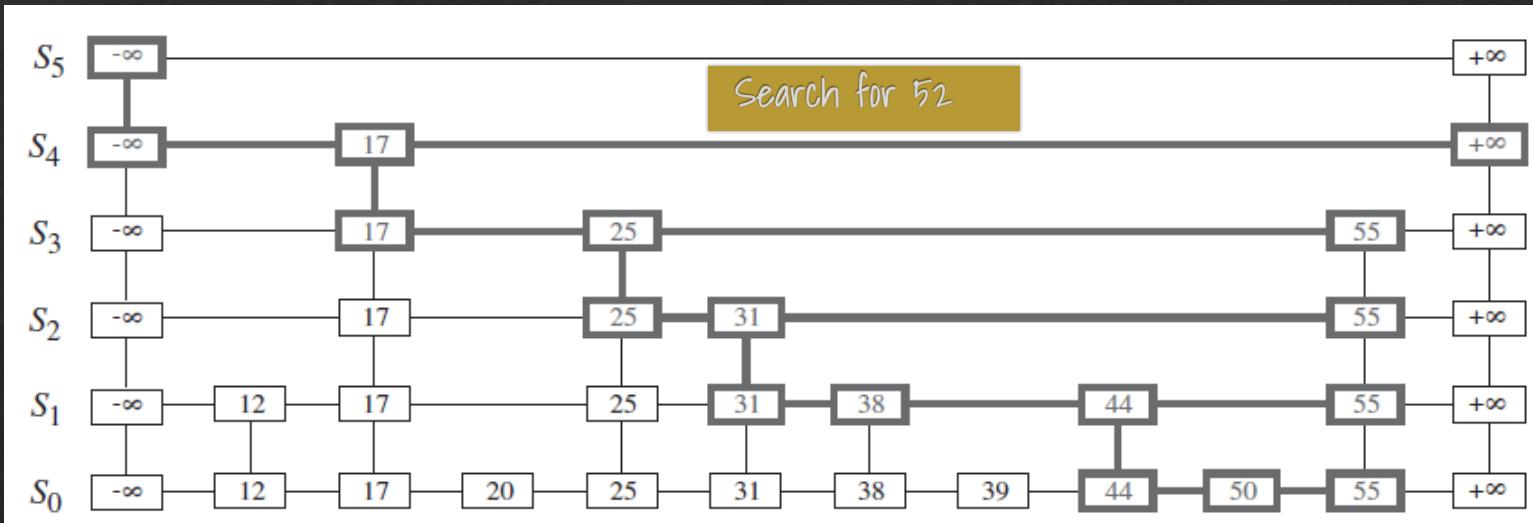


Skip Lists

- ❖ Given an ordered dictionary D of $k-v$ pairs
 - ❖ Skip list S : A series of lists $\{S_0, S_1, \dots, S_h\}$. $h = \text{height of } S$
 - ❖ S_0 contains every item in D (already sorted in non-decreasing order) + special keys
 - ❖ S_i for $i = 1 \dots h-1$: a randomly generated subset of D (already sorted in non-decreasing order) + special keys
 - ❖ S_h only contains special keys
- ❖ Number of levels of skip list with n keys (i.e. h) is $\Theta(\log n)$ with high prob.



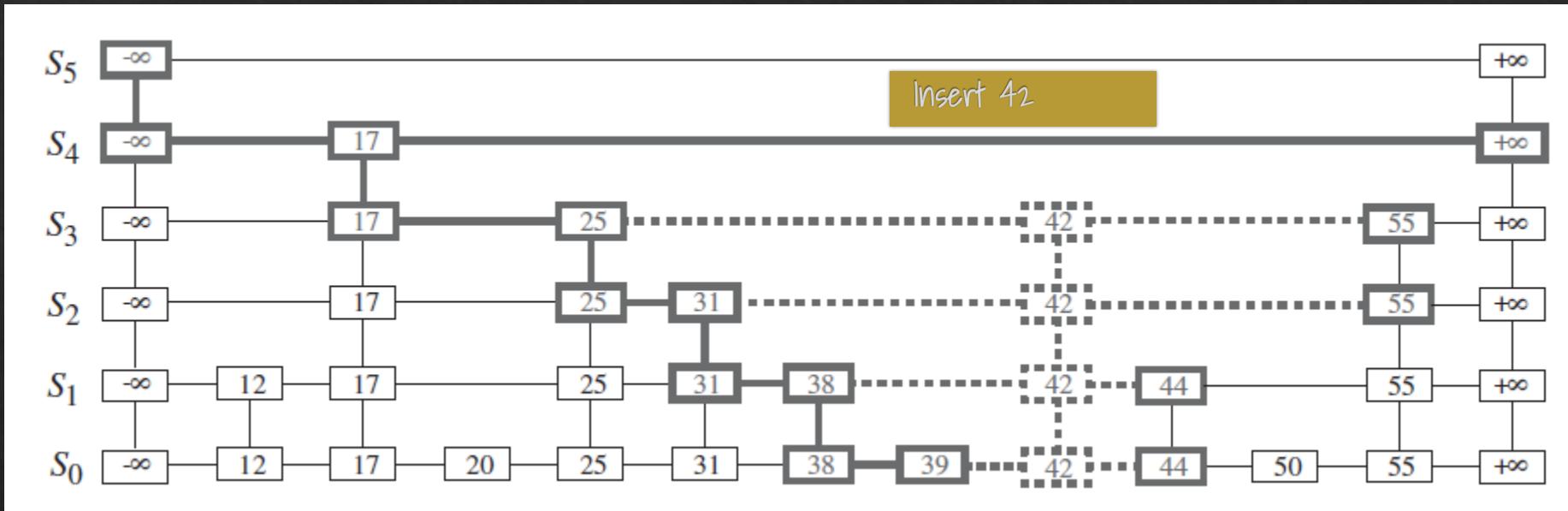
Skip List Searching



Skip List Traversal:

- ◇ $\text{After}(p)$: Return node following p on the same level
- ◇ $\text{Before}(p)$: Return node preceding p on the same level
- ◇ $\text{Below}(p)$: Return node below p on the same tower
- ◇ $\text{Above}(p)$: Return node above p on the same tower

Skip List Insertion



- ❖ Identify node p at S_0 with largest key $\leq k$, insert k
- ❖ Toss a coin (use a bit, if 1 , stop, 0, move up one level), then insert k
- ❖ Toss a coin again... repeat till coin toss = 1

Main Recitation Topics

Quadratic Sorting Algorithm with Priority Queue (Selection Sort & Insertion Sort)

Stock Exchange – Nasdaq Limit Order Book

2500 4.01

1000 4.05

100 4.05

2100 4.02

❖ Price-time Priority

BUY ORDERS

Shares	Price	Age	Shares	Price	Age
1000	4.05	20s	2000	4.06	10s
100	4.05	6s	500	4.07	70s
2100	4.02	2s	1000	4.07	50s
2500	4.01	85s	2100	4.20	5s
			100	4.21	1s

SELL ORDERS

500 4.07

1000 4.07

2000 4.06

2100 4.20

100 4.21

→ Priority queue data structure

Keys determine order to be removed

Priority Queue Used for Sorting Purposes

- ❖ Sorting Algorithms based on Priority Queue include:
 - ❖ Selection sort
 - ❖ Insertion sort
- ❖ Methods for Priority Queue
 - ❖ Insert(k, e): Insert element e with key k into P
 - ❖ removeMin(): Return and remove e with smallest k
- ❖ Total Order
 - ❖ Comparison rule never contradicting itself.
 - ❖ \leq total order relation
 - ❖ Reflexive $k \leq k$
 - ❖ Antisymmetric $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
 - ❖ Transitive: $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$

Priority Queue Used for Sorting Purposes

- ❖ Algorithm:
 - ❖ Put the elements of C (of n elements) into an initially empty P (n insert operations)
 - ❖ Extract the elements in nondecreasing order (n removeMin operations), putting them back to c in order.

Selection Sort Algorithm

PQ is implemented in the form of
Unsorted List



- ❖ Assign index i to a temp variable

$$s = i$$

- ❖ Loop through $i+1$ to n

```
for j = i + 1 to n  
  if  $A[j] < A[s]$  then  
    s = j
```

- ❖ If $s \neq i$
 Swap $A[i]$ and $A[s]$

Loop it $(n-1)$ times from i to $n-1$

In-place Algorithm

Selection Sort Running Time

for $i = 0$ to $n-2$ # or 1 to $n-1$ if starting from 1):

$s = i$

for $j = i+1$ to $n-1$

if $A[j] < A[s]$

$s = j$

if $s \neq i$

$A[i] = A[s]$

$i = 0$: Run $n-1$ comparisons

$i = 1$: Run $n-2$ comparisons

...

$i = n-2$: Run 1 comparisons

$(n-1)+(n-2)+\dots+3+2+1$ comparisons

$O(n)$ swaps/exchanges

$$(n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n-1} i = \frac{n-1}{2}n$$

$O(n^2)$ sort : Quadratic sort

PQ is implemented in the form of sorted List

Insertion Sort Algorithm



- ❖ Assign index i to a temp variable along with its value

$$\begin{aligned}s &= i \\t &= A[i]\end{aligned}$$

- ❖ Loop backward till position 0

```
for j = i - 1 to 0  
if  $A[j] > A[s]$  then  
   $A[s] = A[j]$   
   $s = s - 1$ 
```

- ❖ Found the place for the old index i
 $A[s] = t$

In-place Algorithm

Loop it $(n-1)$ times from 1 to $n-1$

Insertion Sort Running Time

for $i = 1$ to $n-1$ # or 2 to n if starting from 1 :

$s = i$

$t = A[i]$

for $j = i-1$ to 1 #stepping backward ($i-1 > j > 0$)

if $A[j] > A[s]$

$A[s] = A[j]$

$s = s-1$

else

break # stop inside loop

$A[s] = t$

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

$$\sum_{i=1}^{n-1}$$

$$i = \frac{n-1}{2}$$

$i = 0$: Run 1 comparisons

$i = 1$: Run 2 comparisons

...

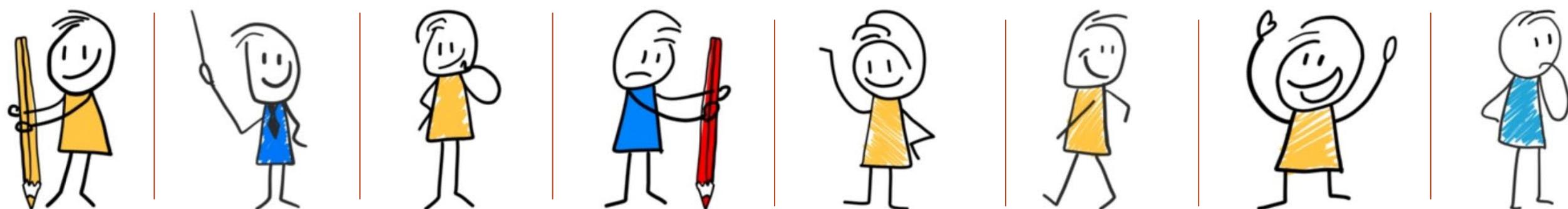
$i = n-1$: Run $n-1$ comparisons

$(n-1)+(n-2)+\dots+3+2+1$ comparisons

$O(n^2)$ sort : Quadratic sort

Win A Prize

- ◆ You are the leader of a group of students which compete in a TV game. Each member of your team is stand in a separate booth next to each other.
- ◆ Challenge: You and your team can't discuss how tall each of you is. Everyone can move freely from booth to booth but can't see the entire team all at once.
- ◆ Goal: All players should be standing in the booth sorted by their height.



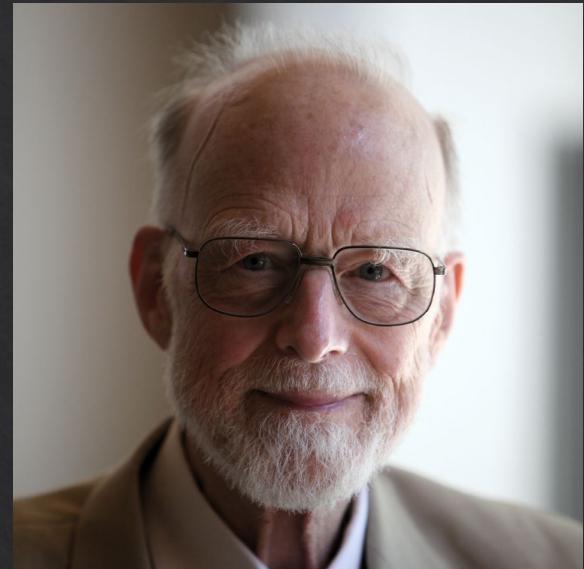
We can't see each
other in the booth

Quick Sort

- ❖ Motivation:
 - ❖ Efficiently sort datasets that fit in the computer's main memory (RAM)
- ❖ Algorithm: Invented in 1959-60 (50+ years ago)

Wikipedia.org extract:

“... The quicksort algorithm was developed in 1959 by [Tony Hoare](#) while he was a visiting student at [Moscow State University](#). At that time, Hoare was working on a [machine translation](#) project for the [National Physical Laboratory](#). As a part of the translation process, he needed to sort the words in Russian sentences before looking them up in a Russian-English dictionary, which was in alphabetical order on [magnetic tape](#).^[5] After recognizing that his first idea, [insertion sort](#), would be slow, he came up with a new idea. He wrote the partition part in [Mercury Autocode](#) but had trouble dealing with the list of unsorted segments. On return to England, he was asked to write code for [Shellsort](#). Hoare mentioned to his boss that he knew of a faster algorithm and his boss bet sixpence that he did not. His boss ultimately accepted that he had lost the bet. Later, Hoare learned about [ALGOL](#) and its ability to do recursion that enabled him to publish the code in [Communications of the Association for Computing Machinery](#), the premier computer science journal of the time. ...”



Inventor: Sir Tony Hoare - Computer Scientist (Also invented insertion sort)

ALGOL was used mostly by research computer scientists in the United States and in Europe. Its use in commercial applications was hindered by the absence of standard [input/output facilities](#) in its description and the lack of interest in the language by large computer vendors other than [Burroughs Corporation](#). ALGOL 60 did however become [the standard for the publication of algorithms](#) and had a profound effect on future language development.

Quick Sort

- ❖ Divide & Conquer Algorithm:
 - ❖ Divide sequence S into 3 parts L, E, G
 - ❖ Conquer Recur(sively) sort sequences L & G
 - ❖ Combine: Insert sorted L , then E , then G
- ❖ Deep dive into in-place quick-sort
 - ❖ Assumption: All elements are unique
- ❖ Pivot element:
 - ❖ Last Element (common practice)
 - ❖ First Element (our Example)
 - ❖ Random (Randomized quick-sort)
- ❖ In practice, quick sort performs better than any other sorting algorithm.

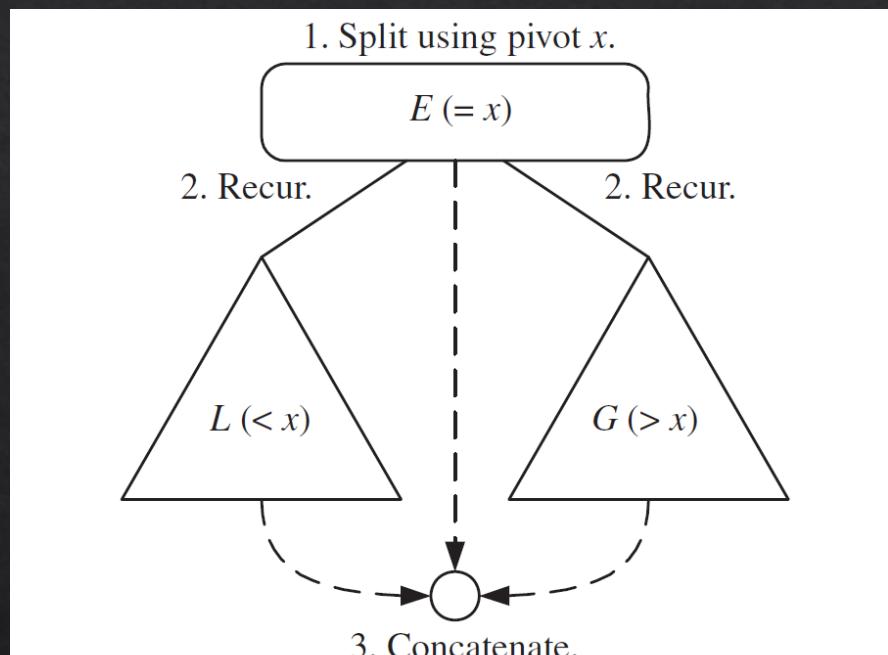


Figure 8.6: A visual schematic of the quick-sort algorithm.

Quick Sort Example

44	75	23	43	55	12	64	77	33
----	----	----	----	----	----	----	----	----

Legend:

Pivot Element A[0]
Greater than
Less than

Step 1: Search for the first value at the left end > pivot value

44	75	23	43	55	12	64	77	33
----	----	----	----	----	----	----	----	----

Step 2: Search for the first value at the right end < pivot value

44	75	23	43	55	12	64	77	33
----	----	----	----	----	----	----	----	----

Step 3: Exchange these values

44	33	23	43	55	12	64	77	75
----	----	----	----	----	----	----	----	----

Left

Right

Quick Sort Example

Repeatedly moving Left, Right till condition is met again



Legend:

Pivot Element A[0]
Greater than
Less than

Swap/Exchange



Stop when Left "passed" Right. Exchange Pivot with Right.

"Partitioning" completed!



Left

Right

Quick Sort Example

Repeat on L



Legend:

Pivot Element A[0]
Greater than
Less than

From Left

From Right

Quick Sort Example

Repeat on G

12	23	33	43	44	55	64	77	75
----	----	----	----	----	----	----	----	----



12	23	33	43	44	55	64	77	75
----	----	----	----	----	----	----	----	----

12	23	33	43	44	55	64	77	75
----	----	----	----	----	----	----	----	----

12	23	33	43	44	55	64	75	77
----	----	----	----	----	----	----	----	----

Legend:

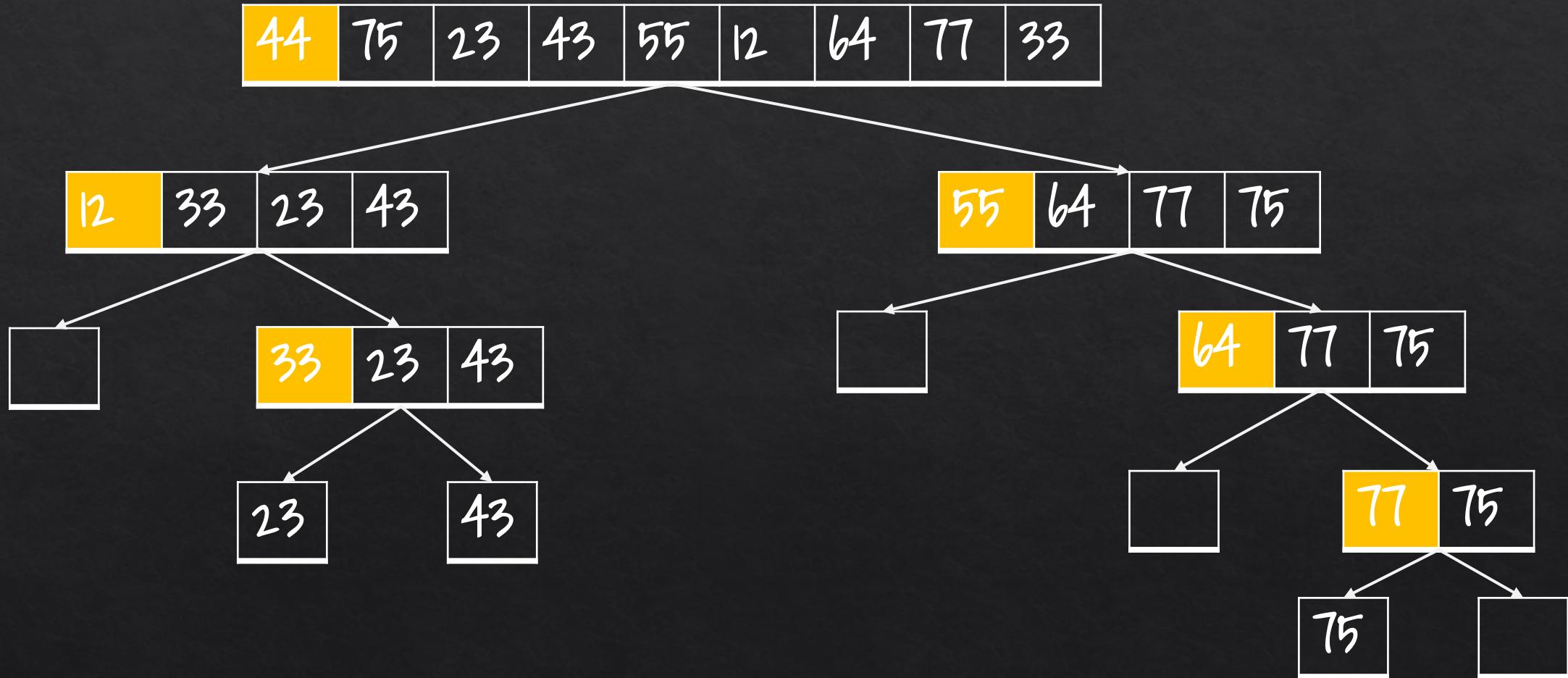
Pivot Element A[0]
Greater than
Less than

From Left

From Right

Done!

Visualize Quick-Sort using Binary Recursion Tree

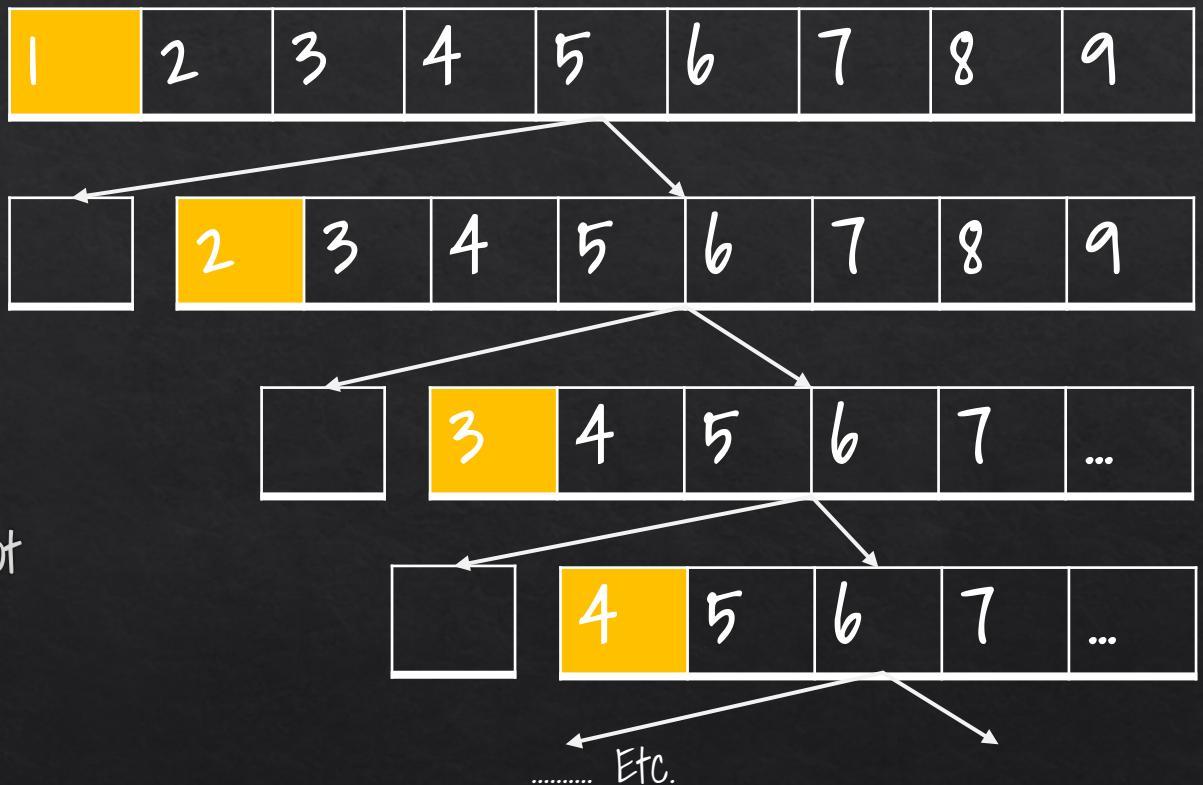


Observation:

Height of the Quick-sort tree is in the worse case. When will this happen?

Worst Case in Quick Sort

- ❖ Worst Case: SORTED Sequence.
 - ❖ Height of the quick-sort tree: $n-1$
 - ❖ If we follow common practice (choosing last element as the pivot)
 - ❖ L sequence has size $n-1$
 - ❖ G sequence has size 0
 - ❖ If we choose the first element as the pivot value
 - ❖ L has size 0
 - ❖ G has size $n-1$



Running Time of Quick Sort

- >Main idea: Quantify the running times for all the nodes in quick-sort tree then sum them up.
- Time spent on each node $v \sim$ input size s_v . Analyze children of v

- L: At least 0, at most $s_v - 1$

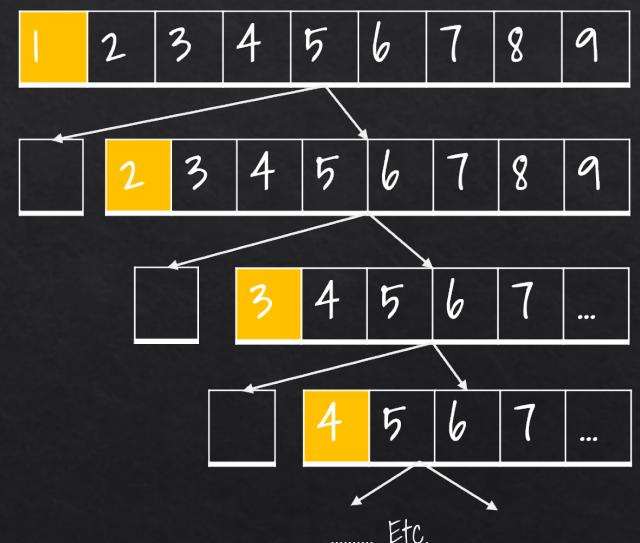
- G: Same as L

- E: At least 1

$$O\left(\sum_{i=0}^{n-1} s_i\right)$$

- Worst case: Height = $n-1$. Running time total:

$$O\left(\sum_{i=0}^{n-1} s_i\right) = O((n-1) + (n-2) + \dots + 1) = O\left(\sum_{i=0}^{n-1} (n-i)\right) = O\left(\sum_{i=1}^n i\right) = O(n^2)$$



- Can you make an educated guess about the best case?

Running Time of Quick Sort

- ◇ Best case:
 - ◇ Balanced input sizes L + G for each node (When it is divided into L, E, G)

depth=0 Root node: processes n values
depth=1 Root's 2 children process $n-1$ values
depth=2 All nodes process $n-(1+2)$ values
...
depth = i All nodes process $n-(1+2+\dots+2^{i-1}) = n-(2^i-1)$ values

Height $O(\log n)$ each level processes max n values $\rightarrow O(n \log n)$

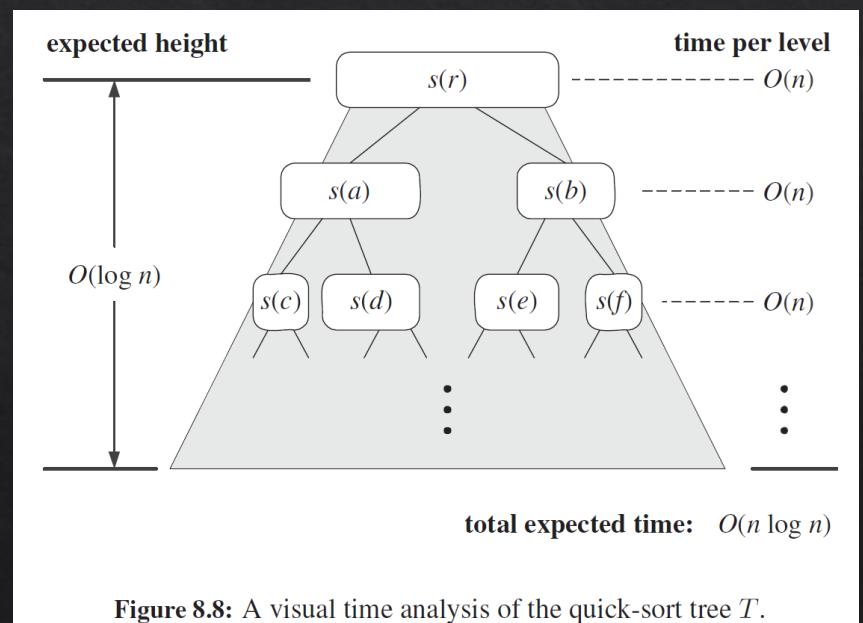


Figure 8.8: A visual time analysis of the quick-sort tree T .

Randomized Quick-Sort

- ◊ Theorem: The expected running time of randomized quick-sort on a sequence of size n is $O(n \log n)$
- ◊ Proof:
 - ◊ Consider a particular recursive invocation. Let $m = \text{size of the input sequence}$.
 - ◊ Invocation is good if pivot divides L and G to have size $[m/4, 3m/4]$. For this to happen, we have $m/2$ chances to choose a good pivot.
 - ◊ v is a good pivot \rightarrow max input size for v 's children is $3s_v/4$ or $(s_v/(4/3))$.
 - ◊ Probability theory: The expected number of times a fair coin must be flipped until it shows k "heads" is $2k$ (*)
 - ◊ From (*) \rightarrow The expected length of any path from root to external node to achieve $\log_{4/3} n$ good invocation = $2 \log_{4/3} n \rightarrow$ Height = $O(\log n)$
 - ◊ Each level $O(n)$
 - ◊ Expected runtime = $O(n \log n)$

More on Quick-sort Performance

◇ Worst Case Recurrence

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

◇ Best Case

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

Applications

- ❖ In the average case, we would have a mix of bad and good splits.
 - ❖ If a bad split 1 and $n-1$ is followed by a good split
 - ❖ The 2nd split leads to an $(n-1)/2$ and $(n-1)/2$ split of the $n-1$ keys.
 - ❖ This give a good split of $(n-1)/2 + 1$ and $(n-1)/2$ after two iterations.
- ❖ The height of the quick-sort tree depends heavily on the choice of pivot.
- ❖ What happens in the case where all the keys are the same?
- ❖ Quicksort is the default library sort subroutine in Unix, appears in C standard library as `qsort`, is also in reference implementation of Java.
- ❖ Multi-pivot quicksort:
 - ❖ In 2009, Vladimir Yoroslavskiy propose a new Quicksort implementation using 2 pivots (chosen to be the new default sorting algo. in Java 7 runtime library after extensive performance tests).

PQ is implemented in the form of sorted List

Insertion Sort Algorithm



- ❖ Assign index i to a temp variable along with its value

$$s = i = 3$$

$$t = A[i] = 3$$

- ❖ Loop backward till position 0

for $j = (i-1)$ to 0
if $A[j] > A[s]$ then
 $A[s] = A[j]$

$$s = s-1$$

- Found the place for the old index i
 $A[s] = t$

1st iteration:
 $s = 3, j = 2$
 $t = 5$
 $A[3] = 6 \rightarrow 2,4,6,6$

In-place Algorithm

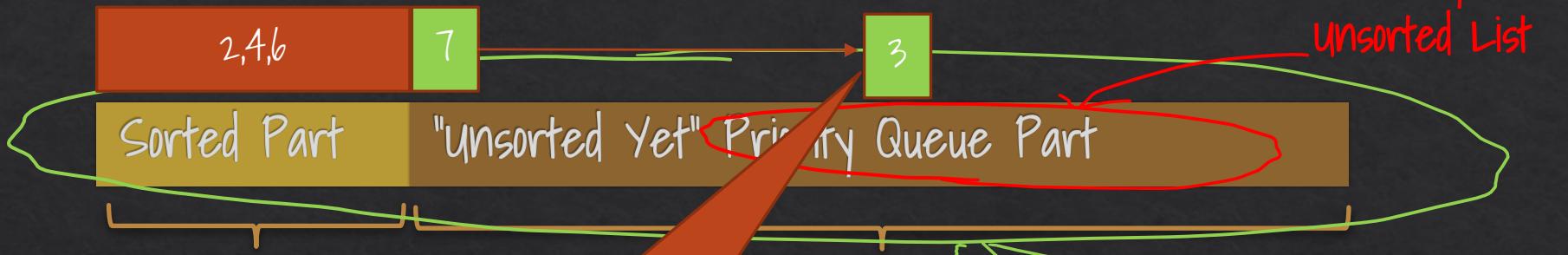
2nd iteration:
 $s = 2, j = 1$
 $t = 5$
 $A[1..i] = 2,4,4,6$

Loop it $(n-1)$ times from 1 to $n-1$

3rd iteration
 $s = 1, j = 0$
 $t = 5 A = 2,4,4,6 \rightarrow A[1] = 3 \rightarrow 2,3,4,6$

NJIT

Selection Sort Algorithm



$A[1..i-1]$

Find the index of the min of unsorted (happen to be value 3 at index x-th)

$A[i..n]$

In-place Algorithm

- ❖ Assign index i to a temp variable
 $s = i$
- ❖ Loop through $i+1$ to n
for $j = i + 1$ to n
if $A[j] < A[s]$ then
 $s = j$
- ❖ If $s \neq i$
Swap $A[i]$ and $A[s]$

Loop it $(n-1)$ times from 1 to $n-1$

Depth First Search

Algorithm $\text{DFS}(G, v)$:

Input: A graph G and a vertex v in G

Output: A labeling of the edges in the connected component of v as discovery edges and back edges, and the vertices in the connected component of v as explored

Label v as explored

for each edge, e , that is incident to v in G do

if e is unexplored **then**

 Let w be the end vertex of e opposite from v

if w is unexplored **then**

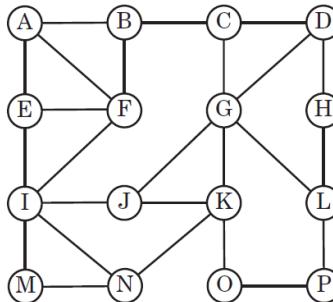
 Label e as a discovery edge

$\text{DFS}(G, w)$

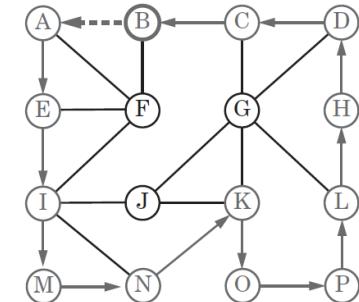
else

 Label e as a back edge

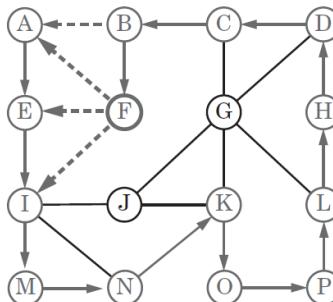
Discovery edges are drawn with solid lines and back edges are drawn with dashed lines. The current vertex is drawn with a thick line: (a) input graph; (b) path of discovery edges traced from A until back edge (B, A) is hit; (c) reaching F , which is a dead end; (d) after backtracking to C , resuming with edge (C, G) , and hitting another dead end, J ; (e) after backtracking to G ; (f) after backtracking to N .



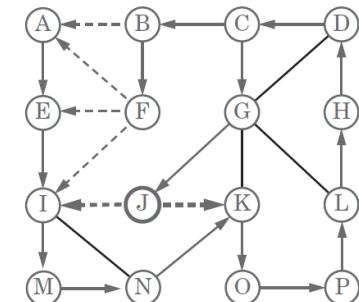
(a)



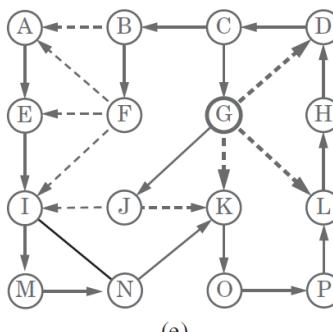
(b)



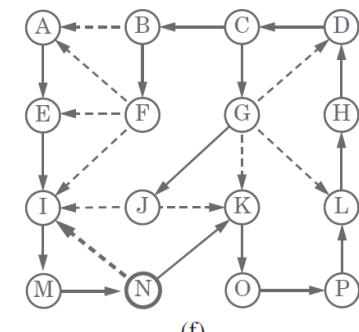
(c)



(d)



(e)



(f)

Breadth First Search

Algorithm BFS(G, s):

Input: A graph G and a vertex s of G

Output: A labeling of the edges in the connected component of s as discovery edges and cross edges

Create an empty list, L_0

Mark s as explored and insert s into L_0

$i \leftarrow 0$

while L_i is not empty **do**

 create an empty list, L_{i+1}

for each vertex, v , in L_i **do**

for each edge, $e = (v, w)$, incident on v in G **do**

if edge e is unexplored **then**

if vertex w is unexplored **then**

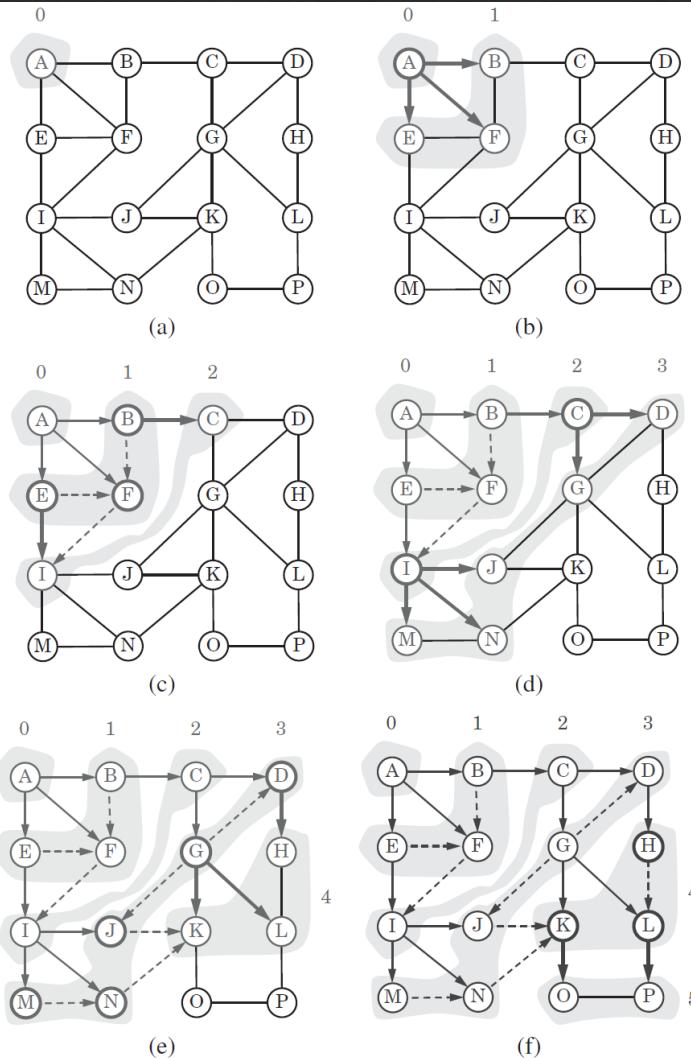
 Label e as a discovery edge

 Mark w as explored and insert w into L_{i+1}

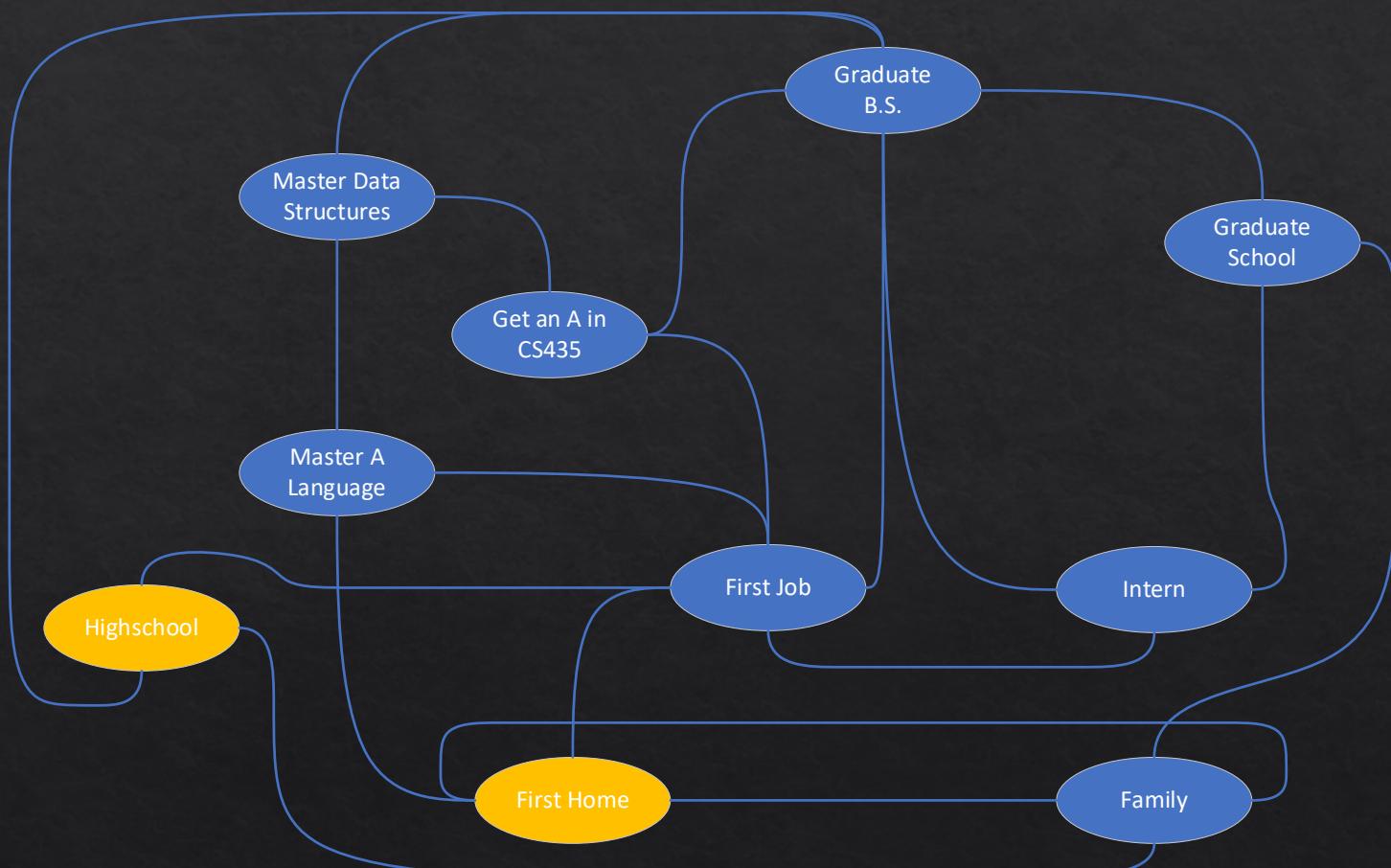
else

 Label e as a cross edge

$i \leftarrow i + 1$



Least Time & Most Money



* Sample goals in graph form in no particular order or priority.

\$ or Miles from JFK to LAX

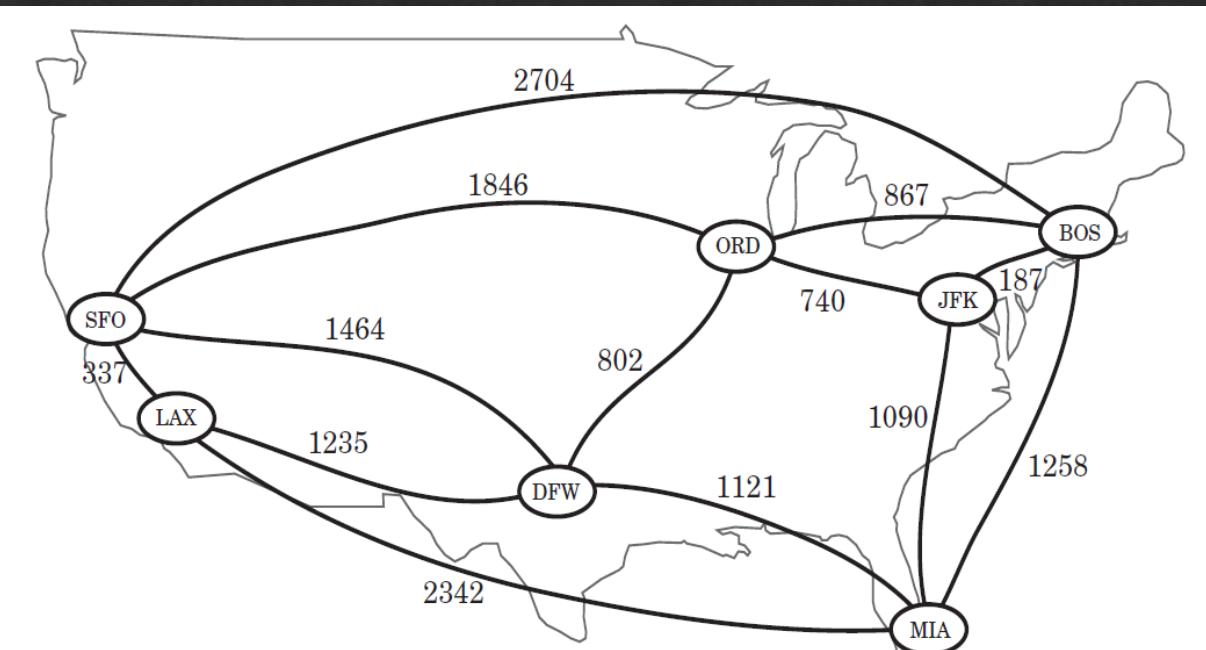


Figure 14.1: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the shortest path in the graph from JFK to LAX.

How map services calculate shortest path (excluding highway/tolls, etc.)? What about ticket route offered to you?

Convention

◇ $G(V, E)$

◇ undirected,

◇ simple (no self-loop, parallel edges),

◇ weighted

◇ $E = \{e_0, e_1, \dots, e_{k-1}\}$. Each edge e_i can be denoted as a vertex pair. E.g. (u, v)

◇ Path P in G .

$$w(P) = \sum_{i=0}^{k-1} w(e_i), \text{ for } w(e_i) \geq 0$$

◇ Distance between vertex u and v in G : $d(u, v)$.

If we declare v as the target $D[u] \leftarrow$ we use to store the best part known so far to v from u .

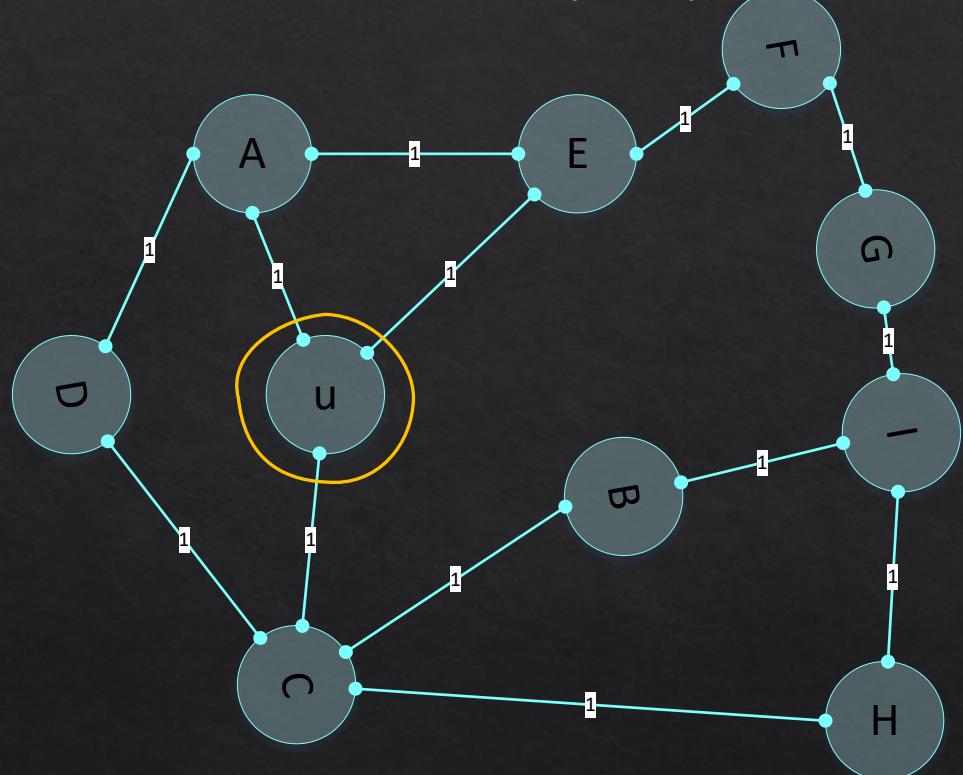
◇ No path at all from u to v : $d(u, v) = +\infty$

Greedy Method

- ◊ Recall Huffman coding, knapsack
- ◊ Definition:
 - ◊ Methods that we use to construct some structure while minimizing or maximizing some property of that structure.
- ◊ Usage: Use to solve optimization problems.
- ◊ Procedure:
 - ◊ Start with a well-understood starting condition(s), compute the cost of this start.
 - ◊ Iteratively make additional choices by identifying the best cost improvement from all currently possible choices (Live in the moment every single time)

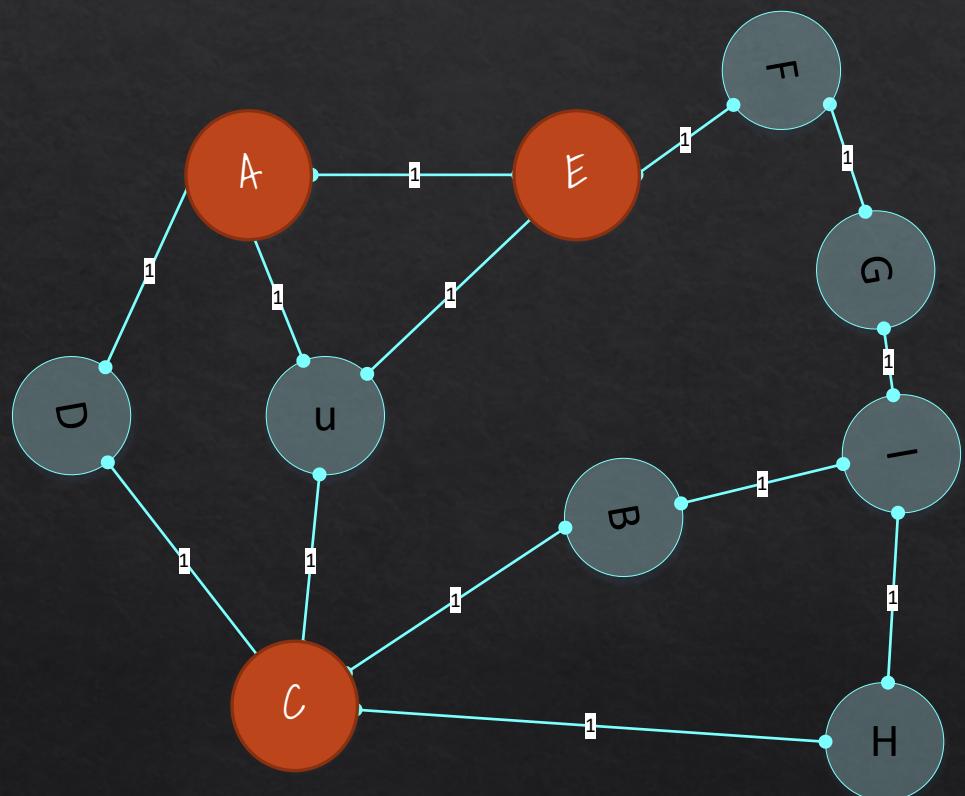
Dijkstra's Algorithm

Example on an unweighted graph



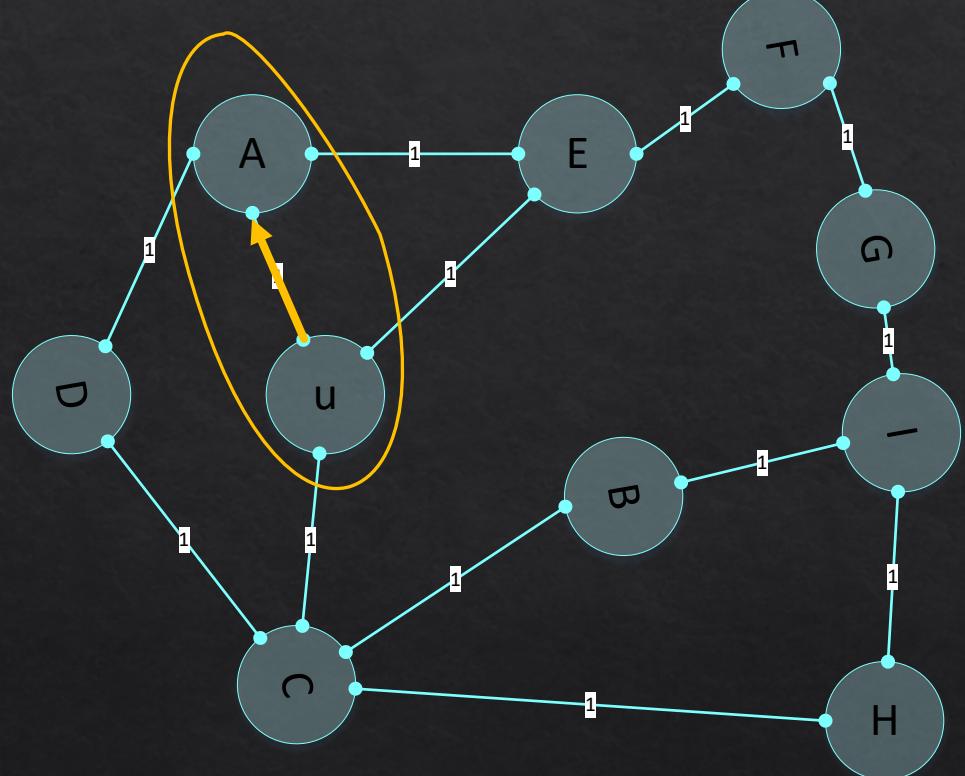
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u.
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G.

Dijkstra's Algorithm



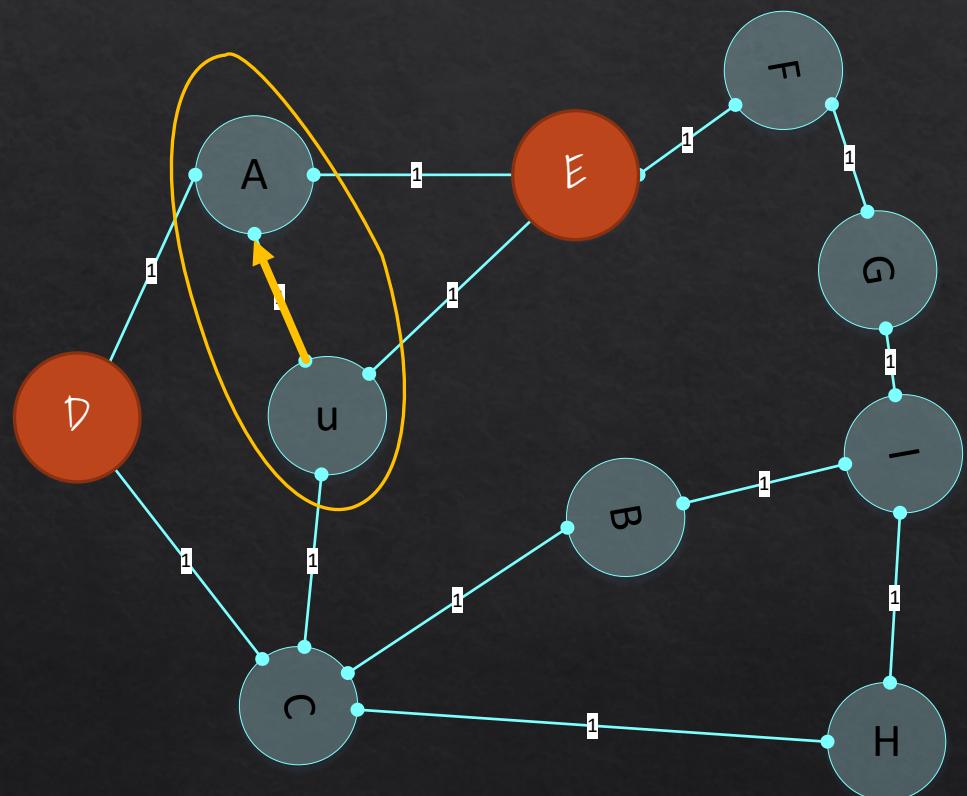
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u.
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G.

Dijkstra's Algorithm



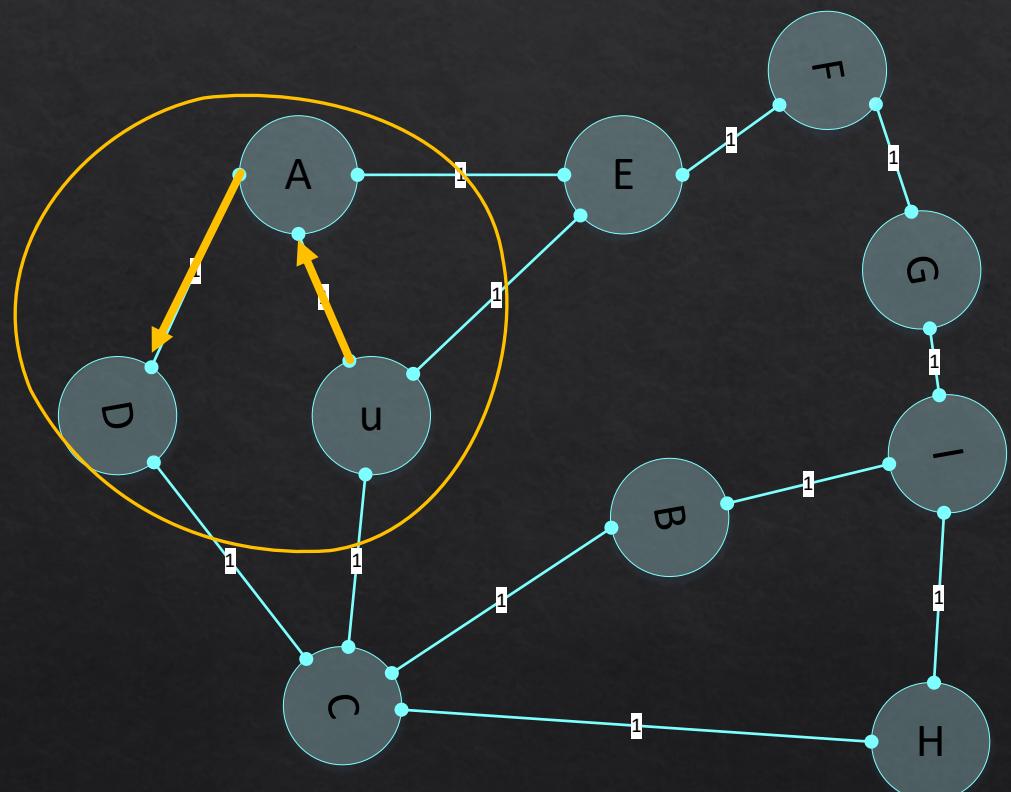
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



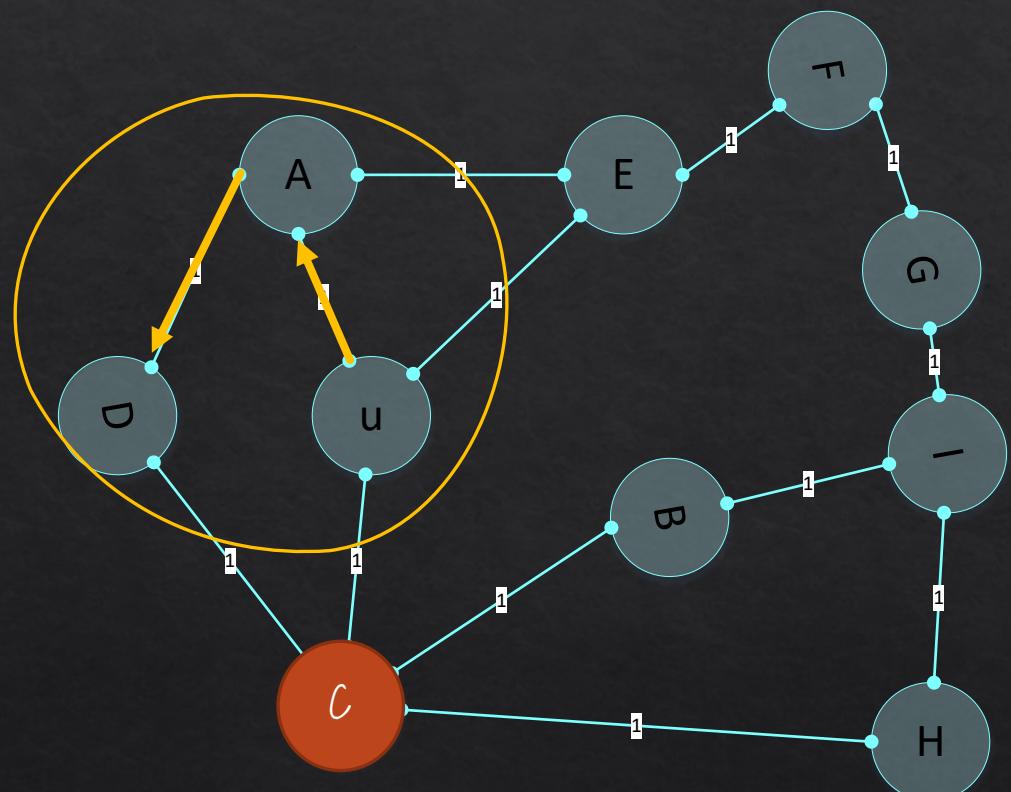
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



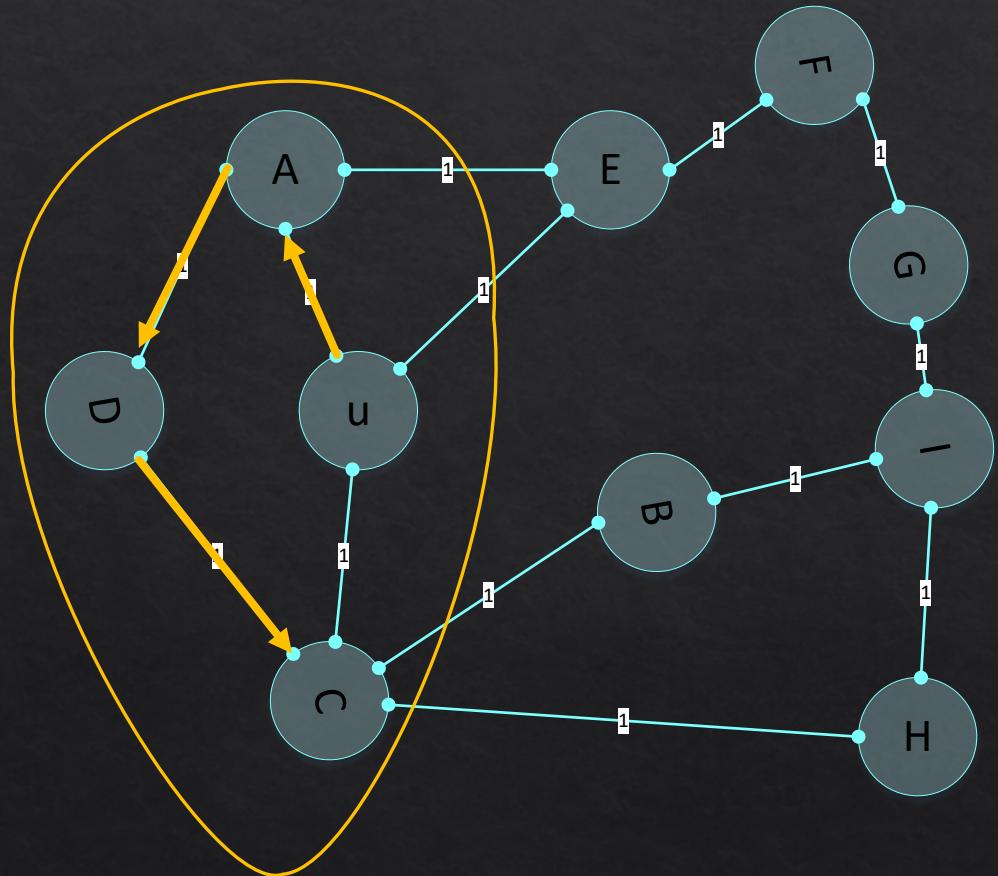
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



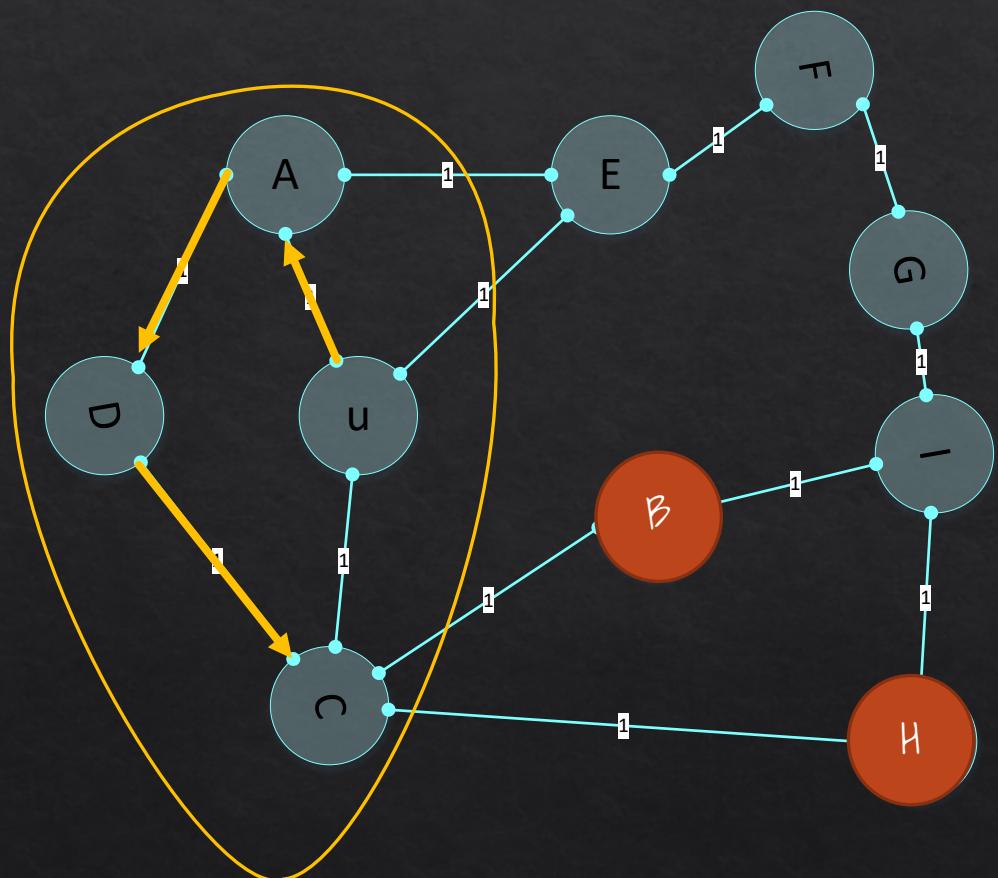
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



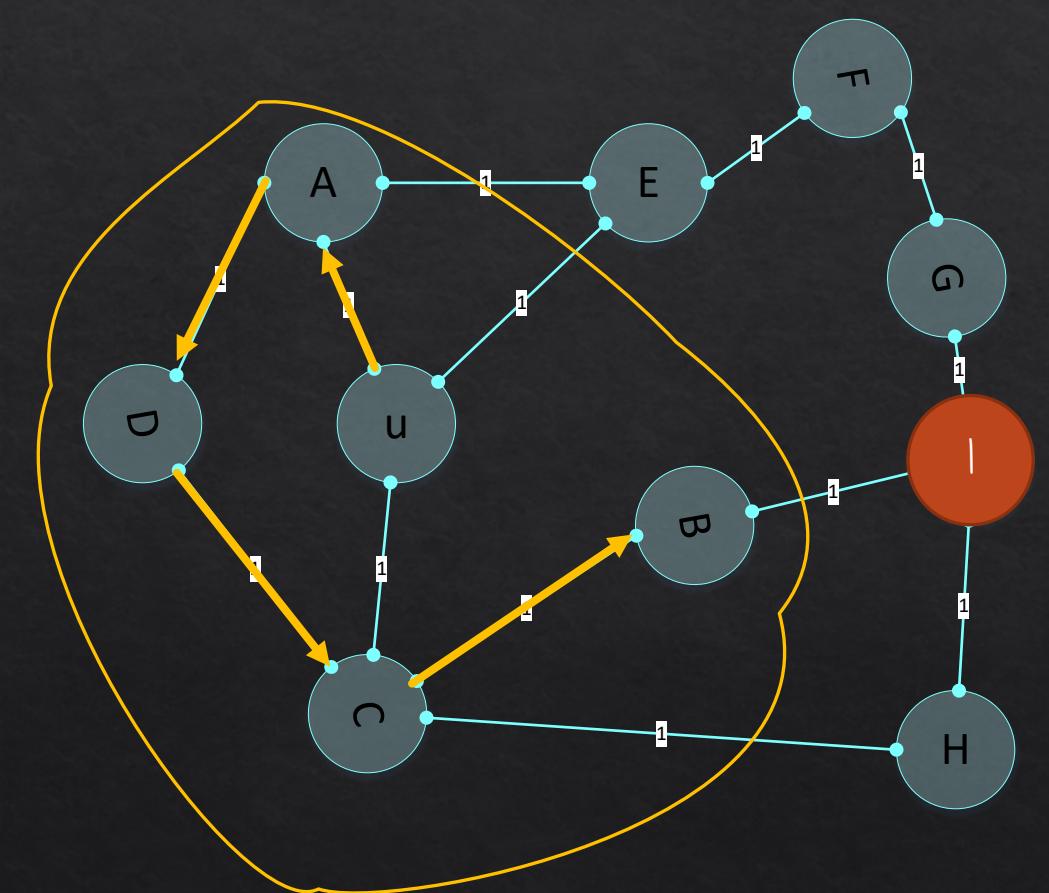
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



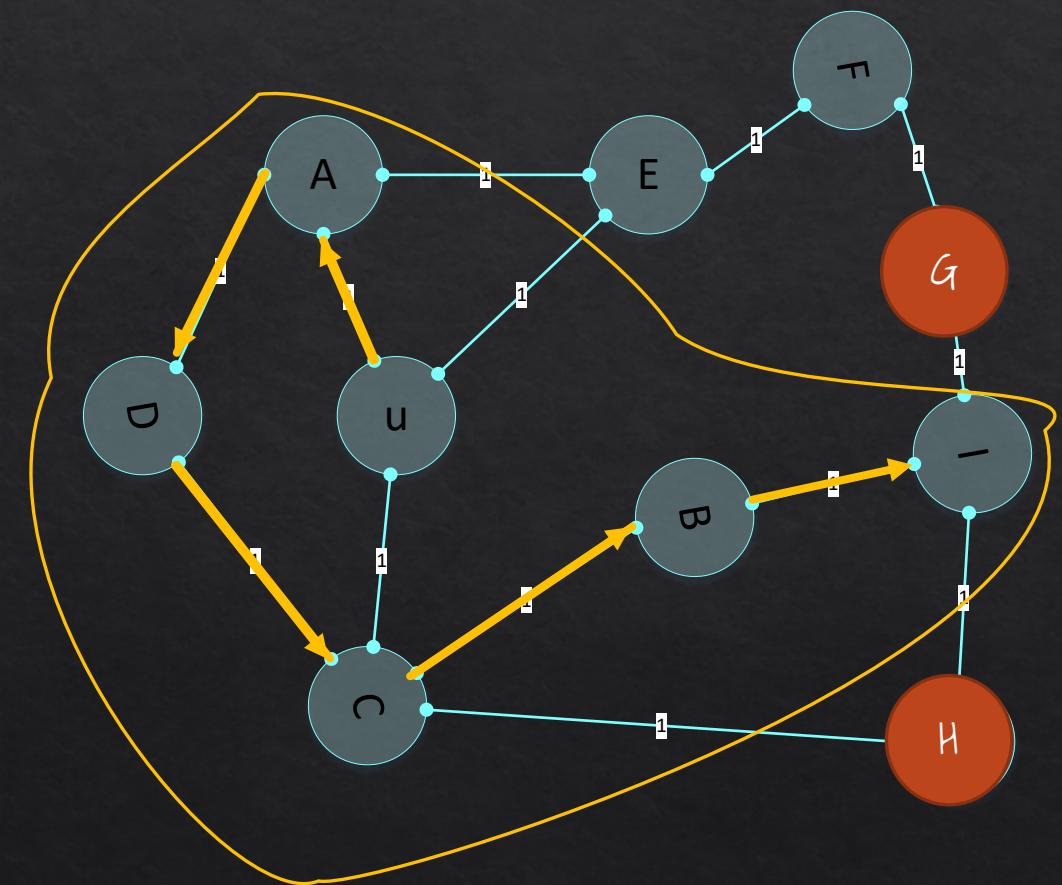
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



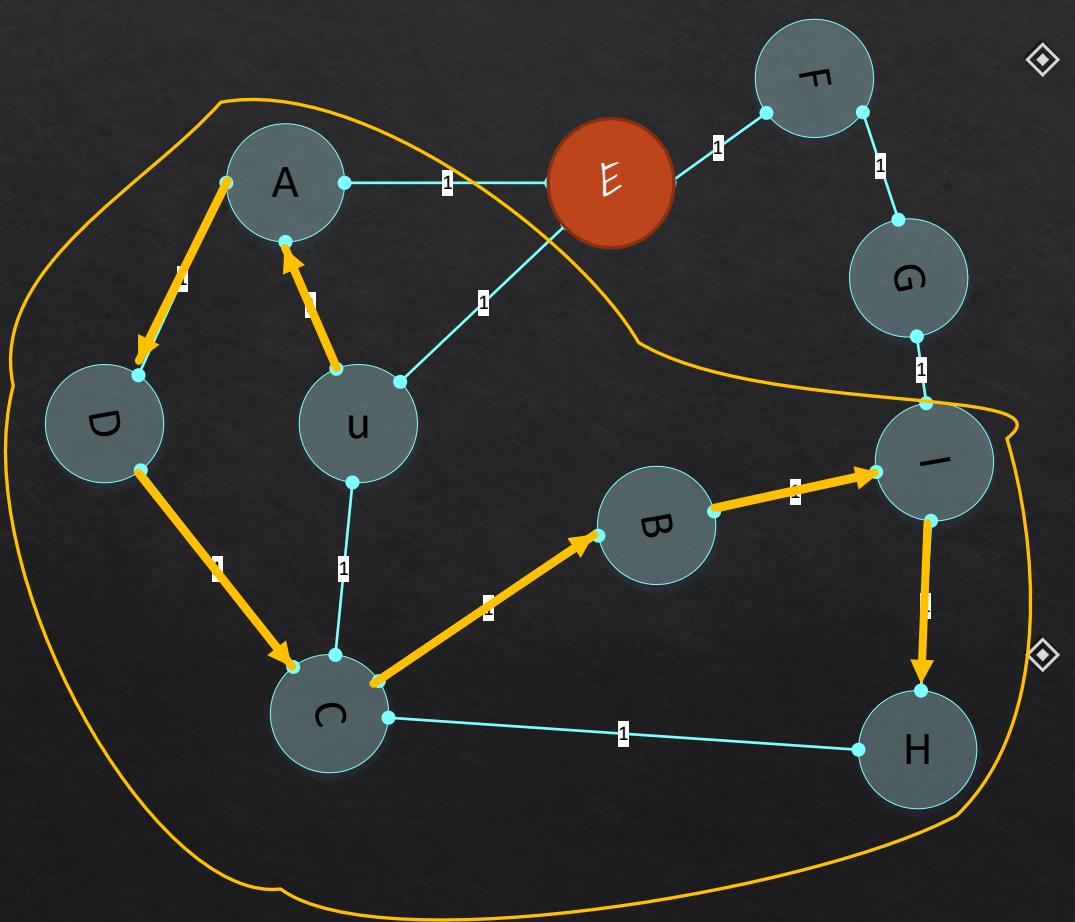
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



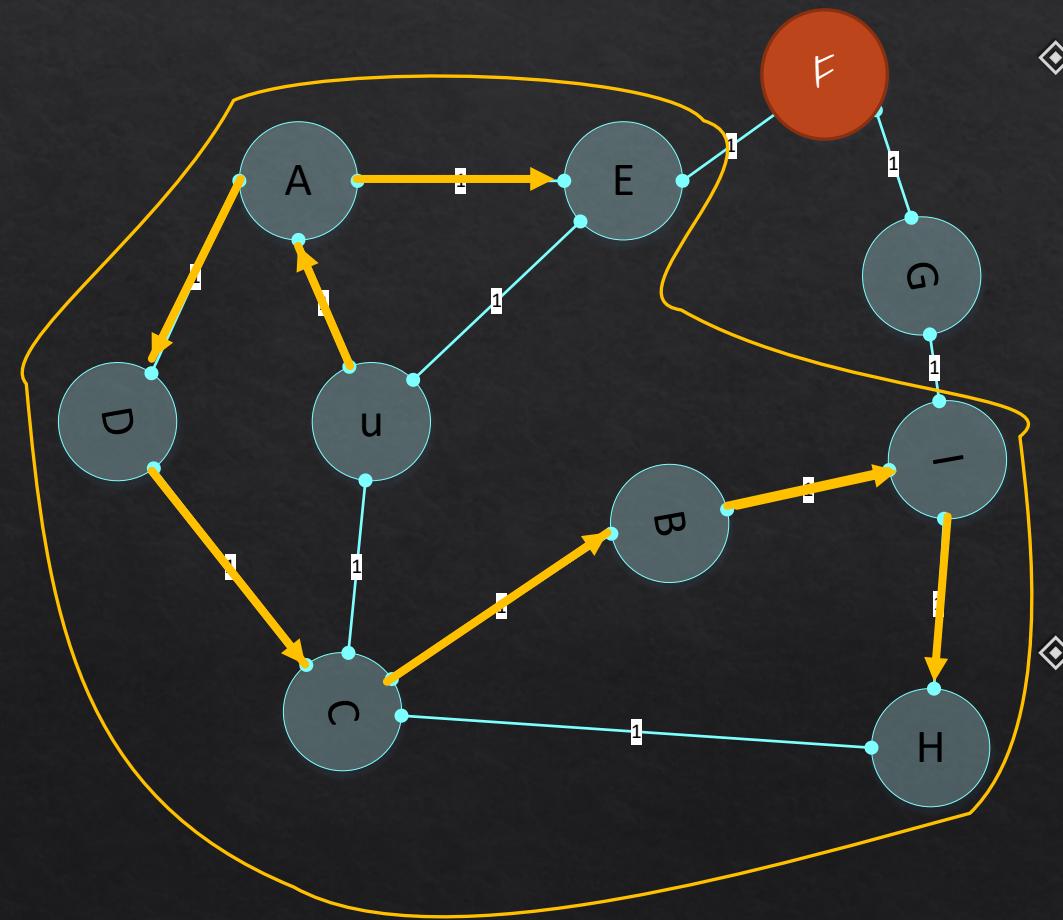
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u.
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G.

Dijkstra's Algorithm



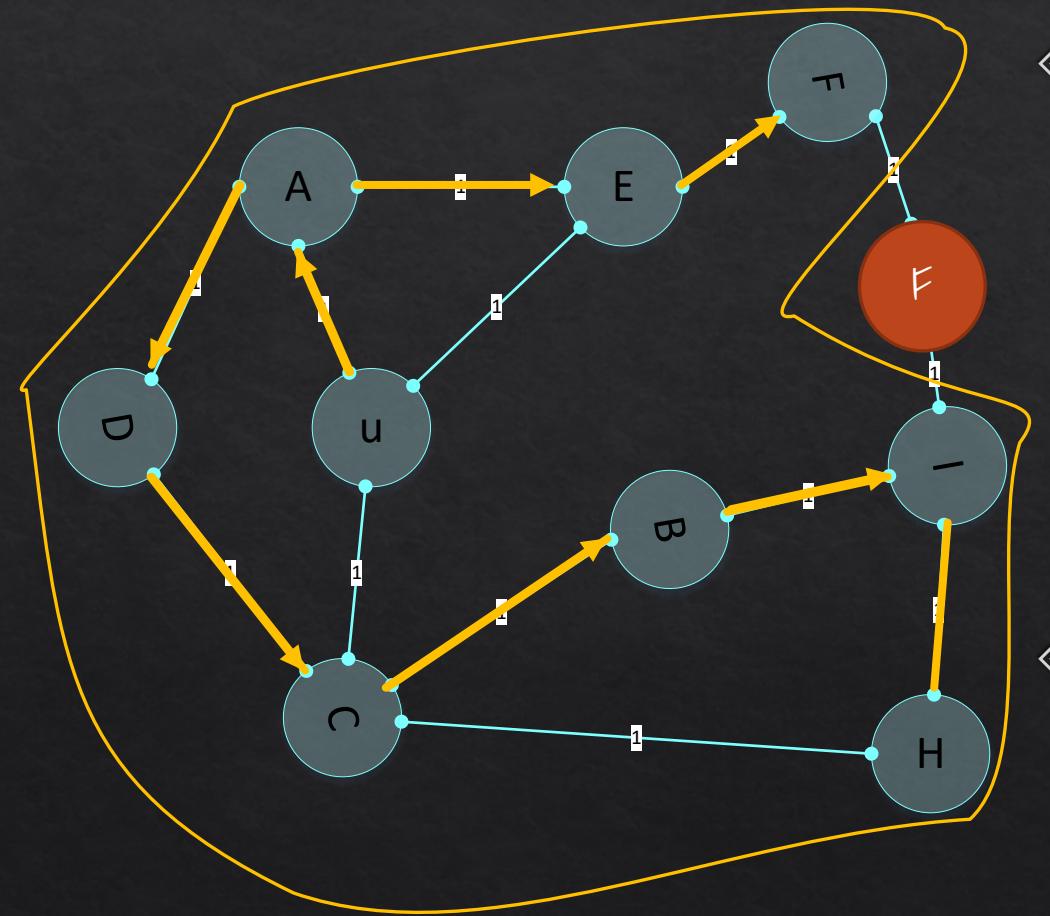
- ❖ Use the greedy method to develop an algorithm that
- ❖ Iteratively grows a "cloud" of vertices out of u
- ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



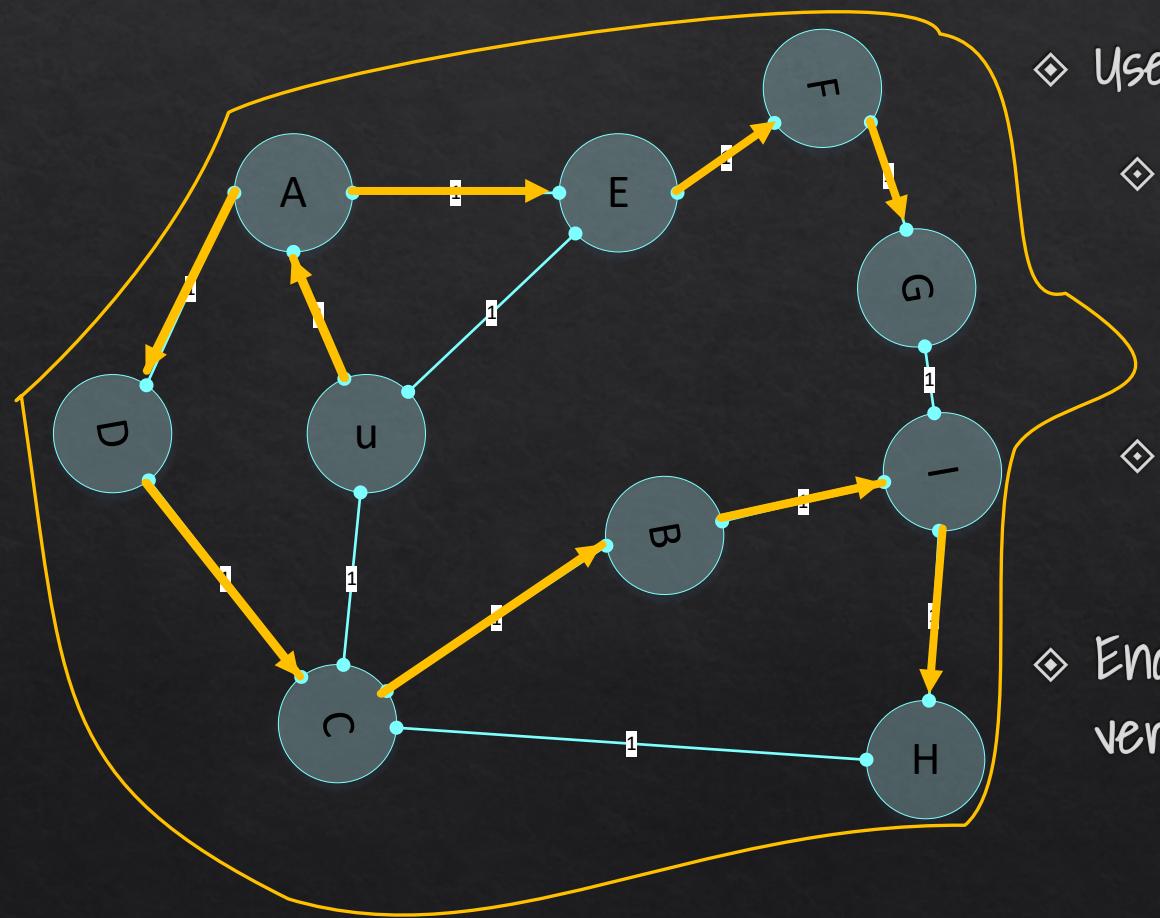
- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm



- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Dijkstra's Algorithm

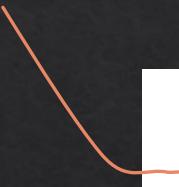


- ❖ Use the greedy method to develop an algorithm that
 - ❖ Iteratively grows a "cloud" of vertices out of u
 - ❖ The vertices entering the cloud by the distances from u .
- ❖ Iterates till all vertices of G are in the cloud.
- ❖ End result: Shortest path from u to every other vertex of G .

Edge Relaxation

Goal: To approximate the distance in G from v to every other vertex u

Always store the length of the best path we have found so far from v to u



```
if  $D[u] + w((u, z)) < D[z]$  then  
     $D[z] \leftarrow D[u] + w((u, z)).$ 
```

Edge Relaxation

Goal: To approximate the distance in G from v to every other vertex $u \neq v$

Always store the length of the best path we have found so far from v to u

$$D[v] = 0$$

$$D[u] = +\infty$$

if $D[u] + w((u, z)) < D[z]$ **then**
 $D[z] \leftarrow D[u] + w((u, z)).$

Edge Relaxation

Goal: To approximate the distance in G from v to every other vertex $u \neq v$

Always store the length of the best path we have found so far from v to u

$$D[v] = 0$$

$$D[u] = +\infty$$

if $D[u] + w((u, z)) < D[z]$ **then**
 $D[z] \leftarrow D[u] + w((u, z)).$

Set $C = \{\emptyset\}$ (our Cloud)

Each iteration,

- > we select a vertex u with smallest $D[u]$ to put into C
- > update $D[z]$ where z is adjacent to u and not yet in C

Edge Relaxation

Goal: To approximate the distance in G from v to every other vertex $u \neq v$

Always store the length of the best path we have found so far from v to u

$$D[v] = 0$$

$$D[u] = +\infty$$

```
if  $D[u] + w((u, z)) < D[z]$  then  
   $D[z] \leftarrow D[u] + w((u, z)).$ 
```

Set $C = \{\emptyset\}$ (our Cloud)

Store length of best path from v to z

Each iteration,

- > we select a u with smallest $D[u]$ to put into C
- > update $D[z]$ where z is adjacent to u and not yet in C

Edge Relaxation

Goal: To approximate the distance in G from v to every other vertex $u \neq v$

Always store the length of the best path we have found so far from v to u

$$D[v] = 0$$

$$D[u] = +\infty$$

```
if  $D[u] + w((u, z)) < D[z]$  then  
   $D[z] \leftarrow D[u] + w((u, z)).$ 
```

Set $C = \{\emptyset\}$ (our Cloud)

Each iteration,

- > we select a u with smallest $D[u]$ to put into C
- > update $D[z]$ where z is adjacent to u and not yet in C

There is a better way to get to z from v
by going through edge (u, z)

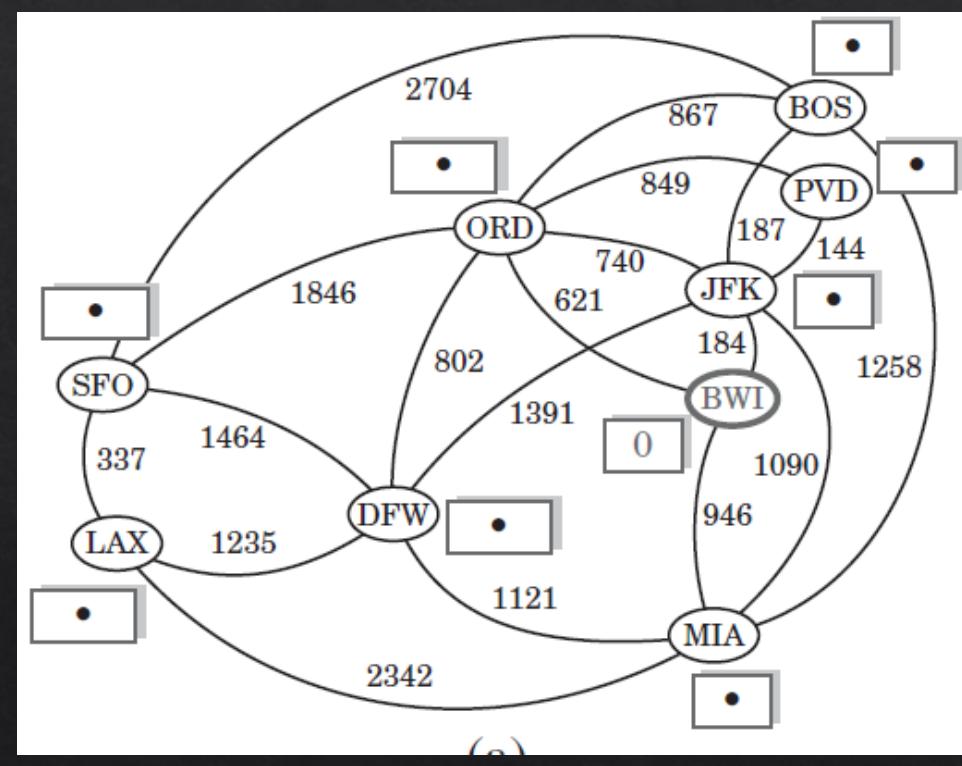
Details of Dijkstra's Algorithm

- ◇ Input: G, v
- ◇ Output: $D[u]$ for all u in $G \neq v$

```
 $D[v] \leftarrow 0$ 
for each vertex  $u \neq v$  of  $G$  do
     $D[u] \leftarrow +\infty$ 
Let a priority queue,  $Q$ , contain all the vertices of  $G$  using the  $D$  labels as keys.
while  $Q$  is not empty do
    // pull a new vertex  $u$  into the cloud
     $u \leftarrow Q.\text{removeMin}()$ 
    for each vertex  $z$  adjacent to  $u$  such that  $z$  is in  $Q$  do
        // perform the relaxation procedure on edge  $(u, z)$ 
        if  $D[u] + w((u, z)) < D[z]$  then
             $D[z] \leftarrow D[u] + w((u, z))$ 
            Change the key for vertex  $z$  in  $Q$  to  $D[z]$ 
return the label  $D[u]$  of each vertex  $u$ 
```

Details of Dijkstra's Algorithm

From BWI

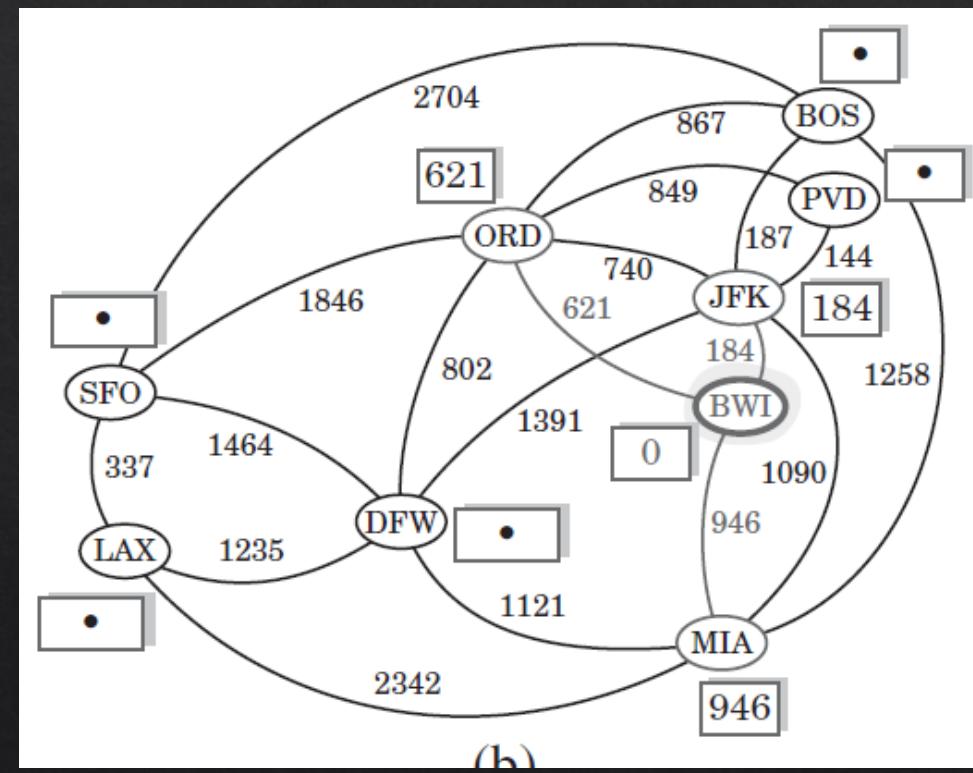


Details of Dijkstra's Algorithm

From BWI

$C = \{JFK\}$

D[BWI]	D[SFO]	D[LAX]	D[DFW]	D[ORD]	D[JFK]	D[BOS]	D[PVD]	D[MIA]
0	+∞	+∞	+∞	+∞	+∞	+∞	+∞	+∞
	+∞	+∞	+∞	621	184	+∞	+∞	946

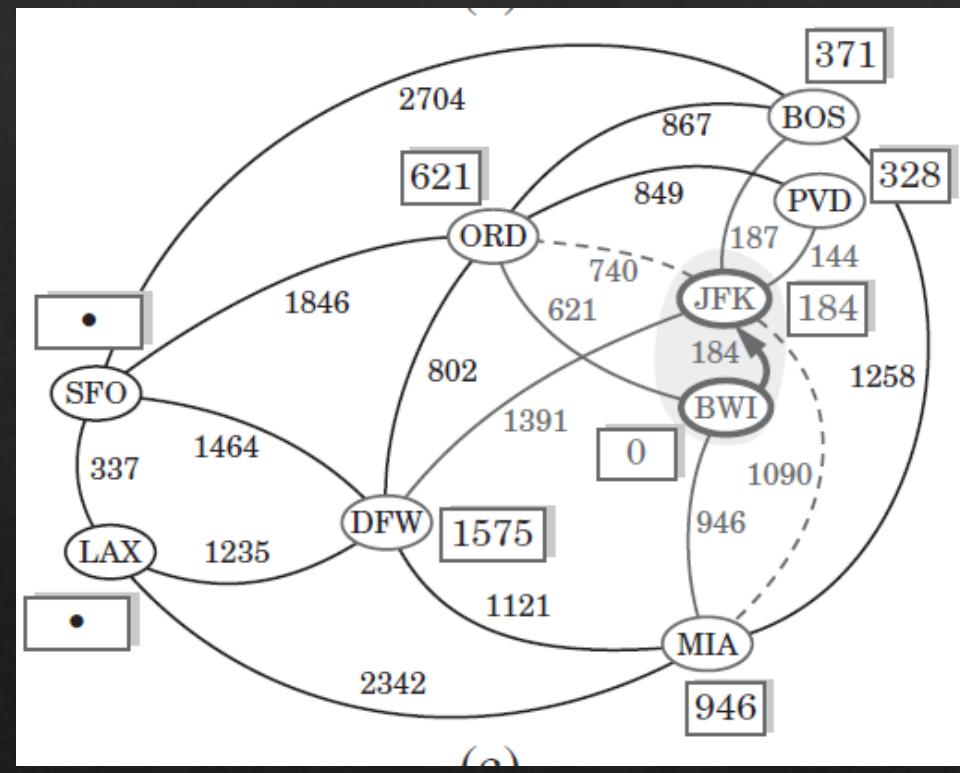


Details of Dijkstra's Algorithm

From BWI

$C = \{JFK\}$

$D[BWI]$	$D[SFO]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[BOS]$	$D[PVD]$	$D[MIA]$
0	$+\infty$							
	$+\infty$	$+\infty$	$+\infty$	621	184	$+\infty$	$+\infty$	946
	$+\infty$	$+\infty$	1575			371	328	

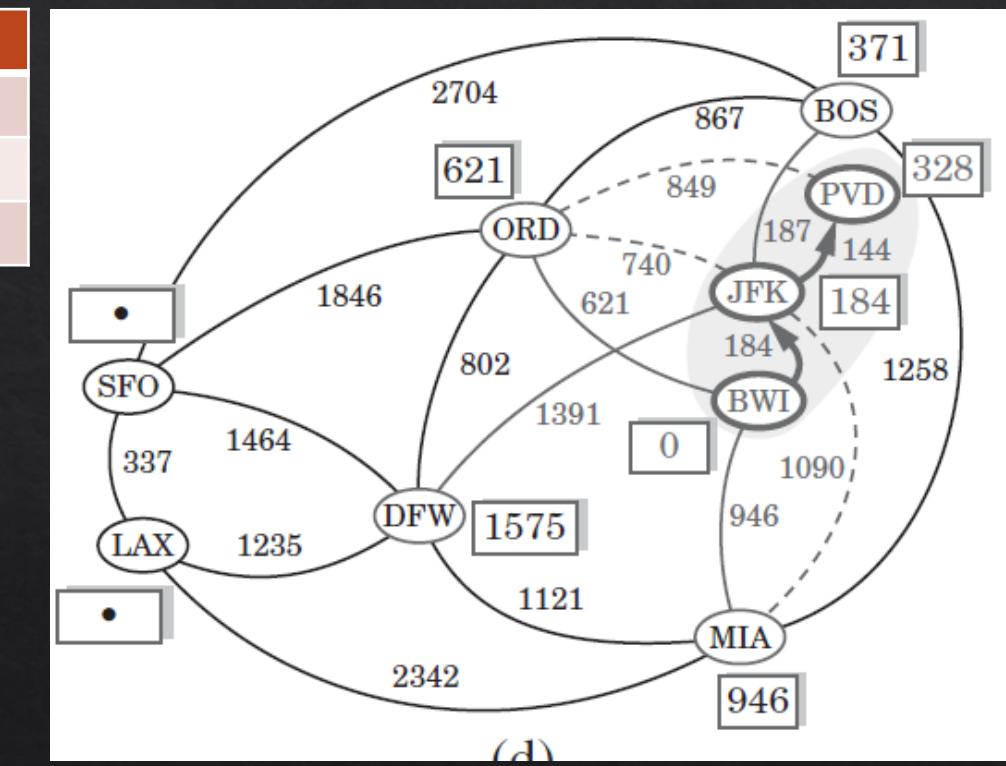


Details of Dijkstra's Algorithm

From BWI

$C = \{\text{JFK}, \text{PVD}\}$

$D[\text{BWI}]$	$D[\text{SFO}]$	$D[\text{LAX}]$	$D[\text{DFW}]$	$D[\text{ORD}]$	$D[\text{JFK}]$	$D[\text{BOS}]$	$D[\text{PVD}]$	$D[\text{MIA}]$
0	$+\infty$							
	$+\infty$	$+\infty$	$+\infty$	621	184	$+\infty$	$+\infty$	946
	$+\infty$	$+\infty$	1575			371	328	

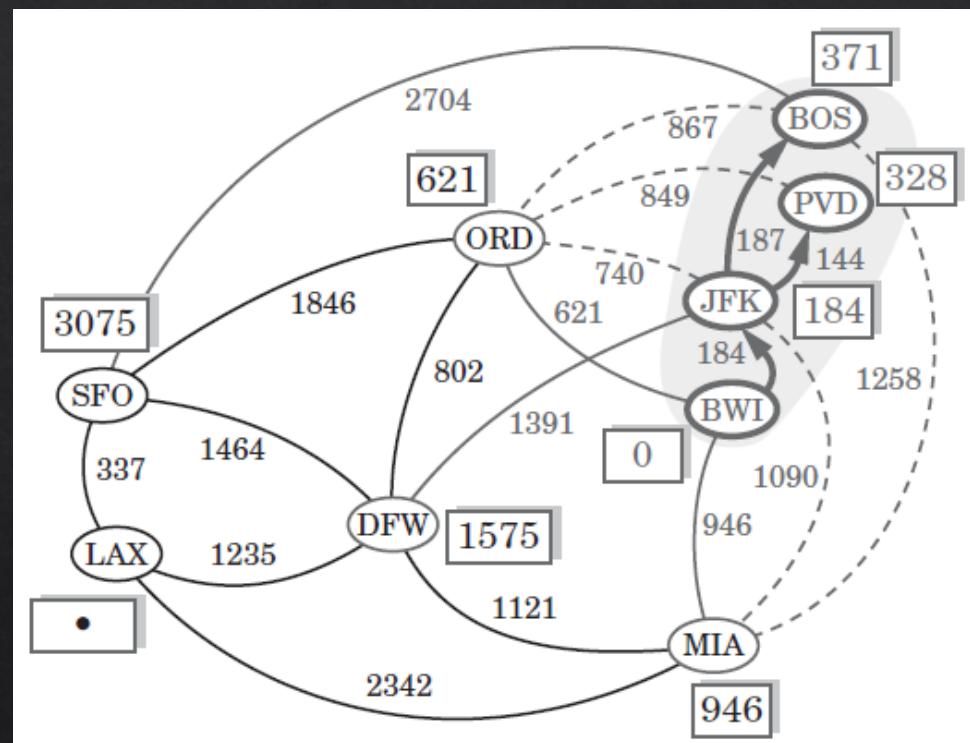


Details of Dijkstra's Algorithm

From BWI

$$C = \{JFK, PVD, BOS\}$$

D[BWI]	D[SFO]	D[LAX]	D[DFW]	D[ORD]	D[JFK]	D[BOS]	D[PVD]	D[MIA]
0	+∞	+∞	+∞	+∞	+∞	+∞	+∞	+∞
	+∞	+∞	+∞	621	184	+∞	+∞	946
	+∞	+∞	1575			371	328	
	3075	+∞						

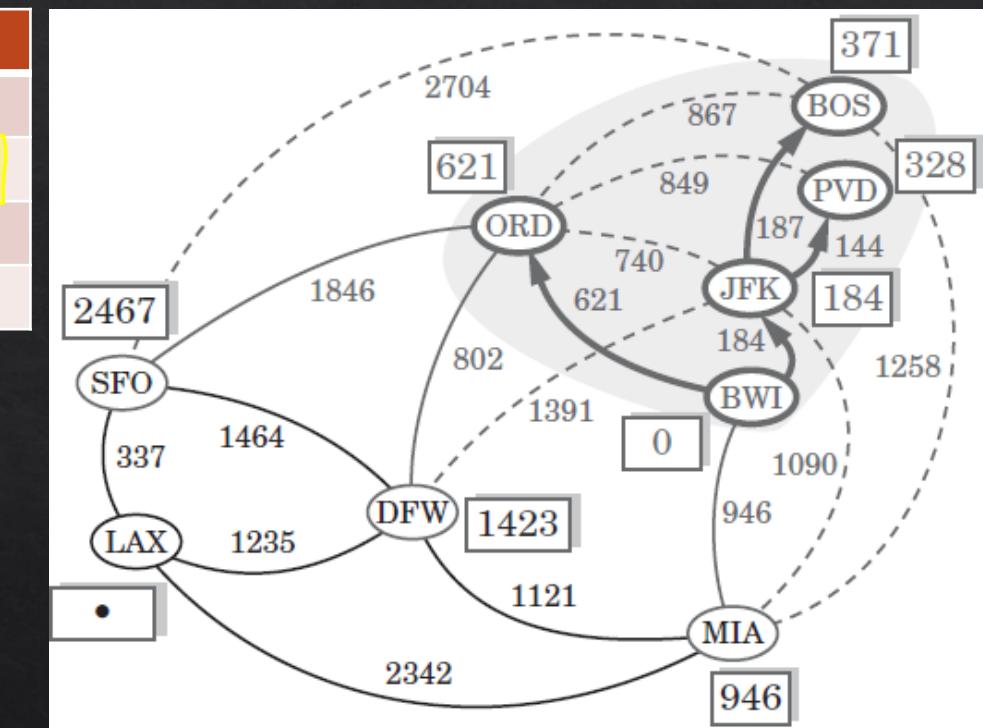


Details of Dijkstra's Algorithm

From BWI

$C = \{\text{JFK}, \text{PVD}, \text{BOS}, \text{ORD}\}$

$D[\text{BWI}]$	$D[\text{SFO}]$	$D[\text{LAX}]$	$D[\text{DFW}]$	$D[\text{ORD}]$	$D[\text{JFK}]$	$D[\text{BOS}]$	$D[\text{PVD}]$	$D[\text{MIA}]$
0	$+\infty$							
	$+\infty$	$+\infty$	$+\infty$	621	184	$+\infty$	$+\infty$	946
	$+\infty$	$+\infty$	1575			371	328	
	2467	$+\infty$	1423					

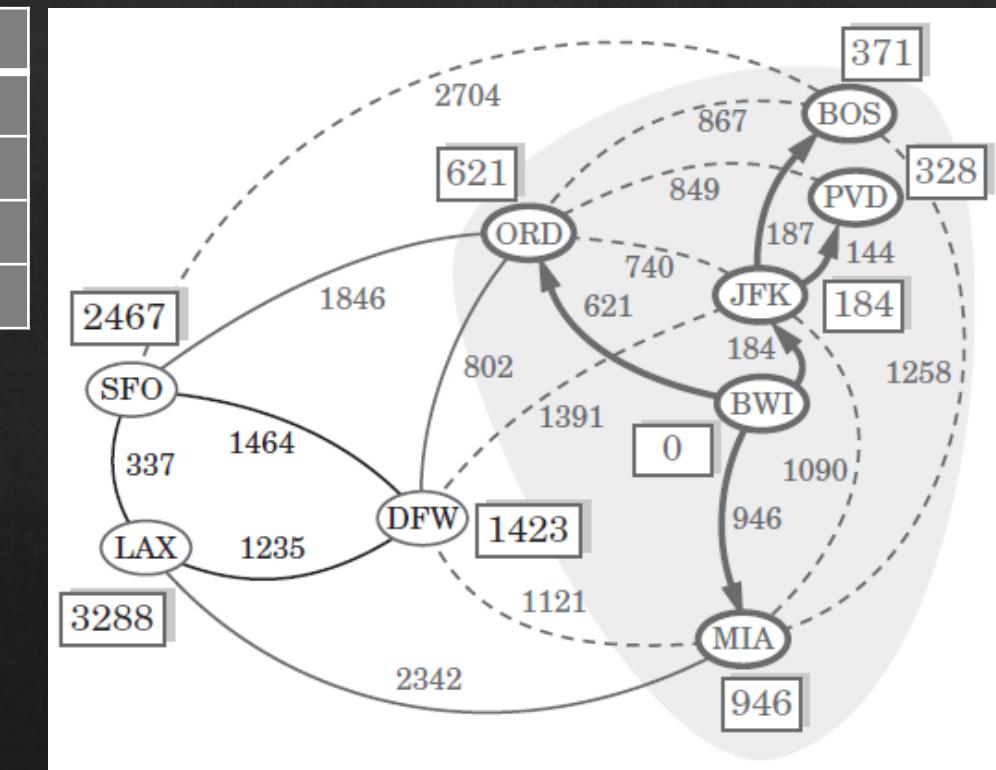


Details of Dijkstra's Algorithm

From BWI

$C = \{JFK, PVD, BOS, ORD, MIA\}$

$D[BWI]$	$D[SFO]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[BOS]$	$D[PVD]$	$D[MIA]$
0	$+\infty$							
	$+\infty$	$+\infty$	$+\infty$	621	184	$+\infty$	$+\infty$	946
	$+\infty$	$+\infty$	1575			371	328	
	2467	3288	1423					

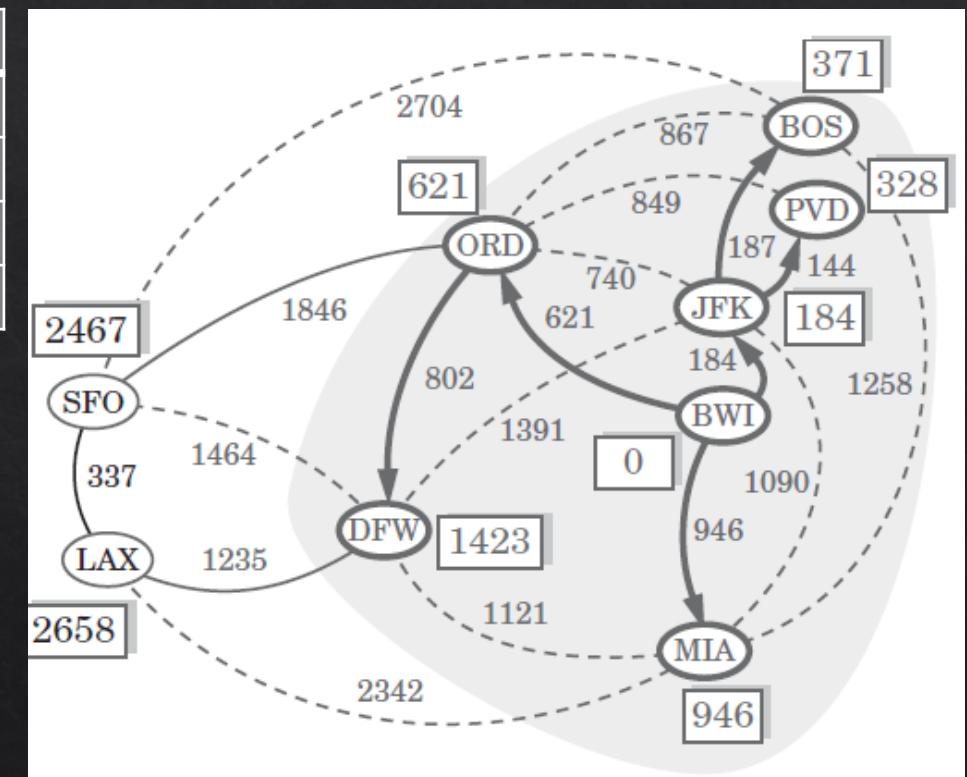


Details of Dijkstra's Algorithm

From BWI

$C = \{JFK, PVD, BOS, ORD, MIA, DFW\}$

$D[BWI]$	$D[SFO]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[BOS]$	$D[PVD]$	$D[MIA]$
0	$+\infty$							
	$+\infty$	$+\infty$	$+\infty$	621	184	$+\infty$	$+\infty$	946
	$+\infty$	$+\infty$	1575			371	328	
2467	2658	1423						

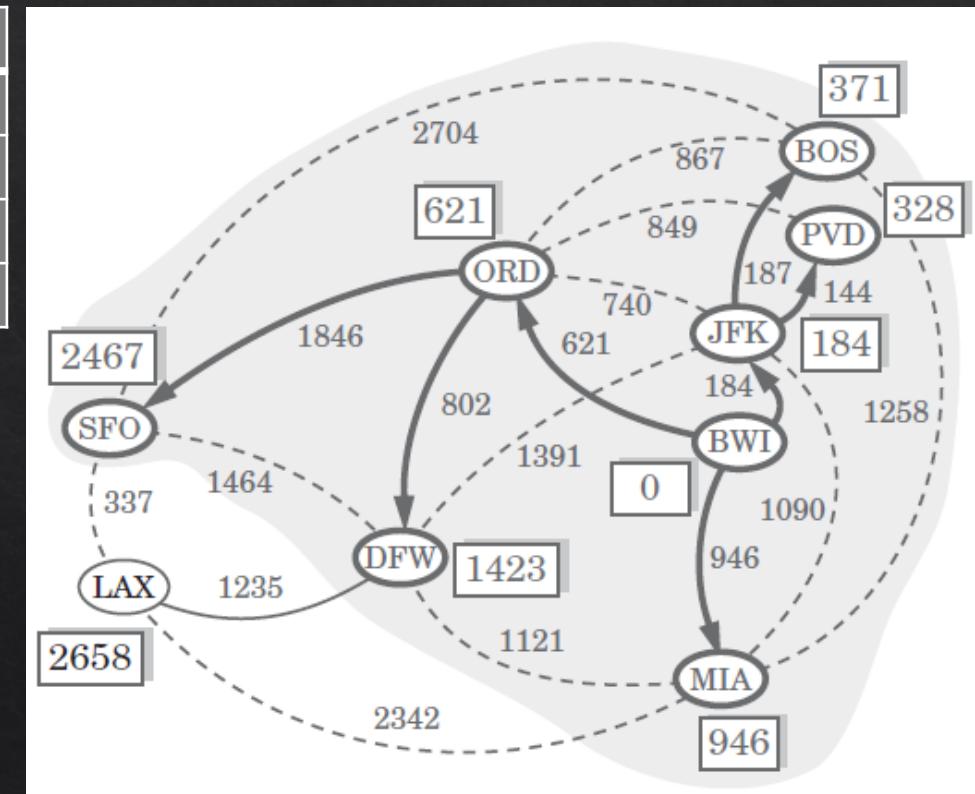


Details of Dijkstra's Algorithm

From BWI

$C = \{JFK, PVD, BOS, ORD, MIA, DFW, SFO\}$

$D[BWI]$	$D[SFO]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[BOS]$	$D[PVD]$	$D[MIA]$
0	$+\infty$							
	$+\infty$	$+\infty$	$+\infty$	621	184	$+\infty$	$+\infty$	946
	$+\infty$	$+\infty$	1575			371	328	
2467	2658	1423						



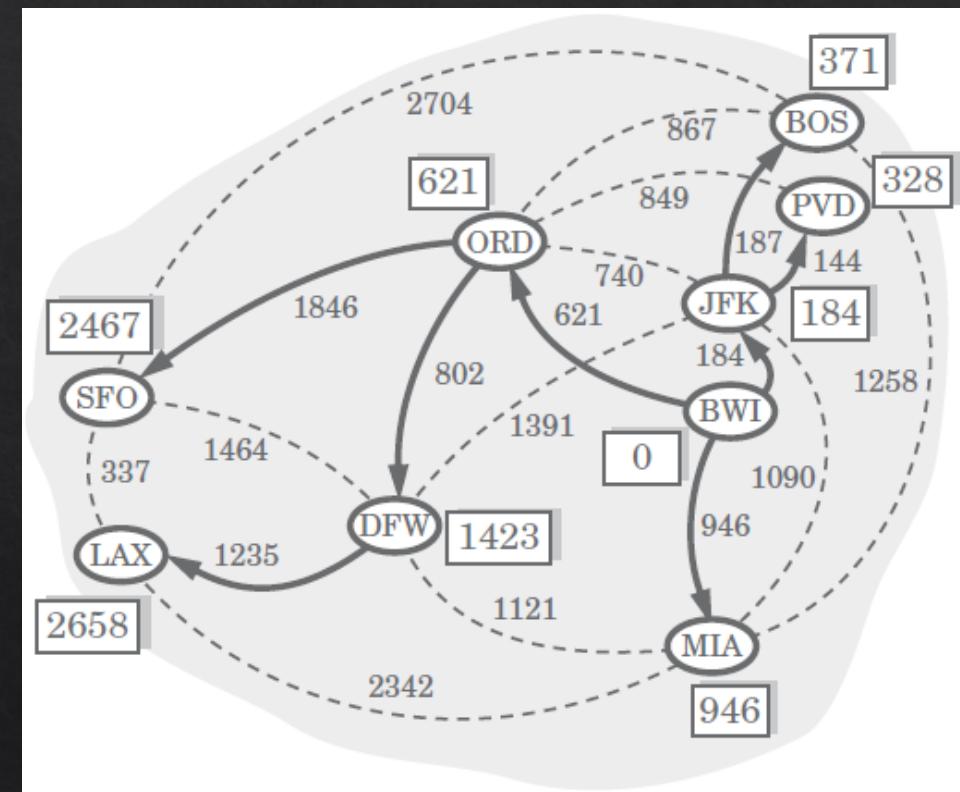
Details of Dijkstra's Algorithm

From BWI

$C = \{JFK, PVD, BOS, ORD, MIA, DFW, SFO, LAX\}$

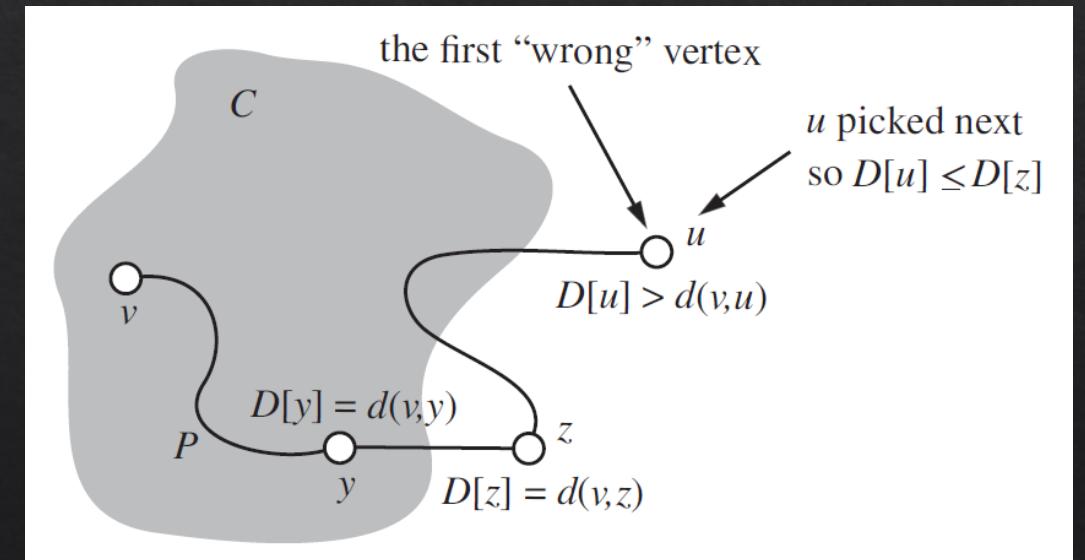
$D[BWI]$	$D[SFO]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[BOS]$	$D[PVD]$	$D[MIA]$
0	$+\infty$							
	$+\infty$	$+\infty$	$+\infty$	621	184	$+\infty$	$+\infty$	946
	$+\infty$	$+\infty$	1575			371	328	
	2467	2658	1423					

Single-source shortest path problem solved.



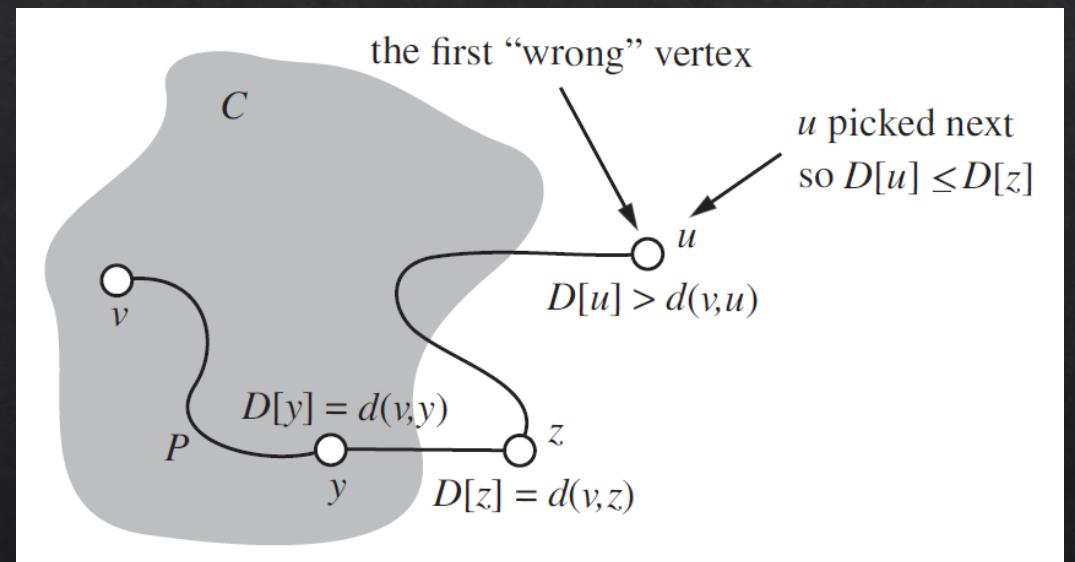
Lemma

- ❖ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ."
- ❖ Proof by contradiction:
 - ❖ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$



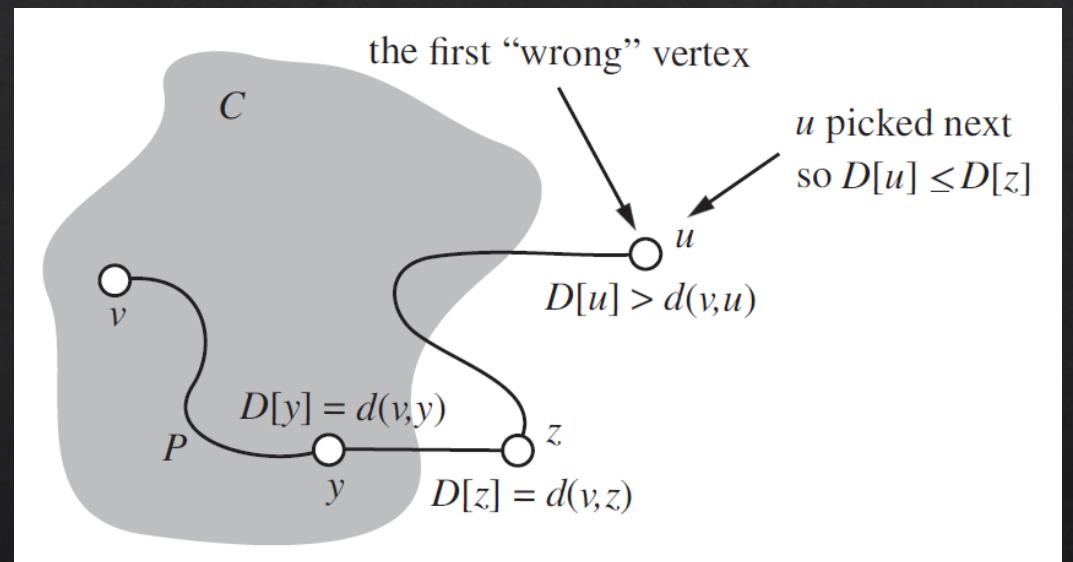
Lemma

- ❖ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ".
- ❖ Proof by contradiction:
 - ❖ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$
 - ❖ We know there exists a shortest path P .



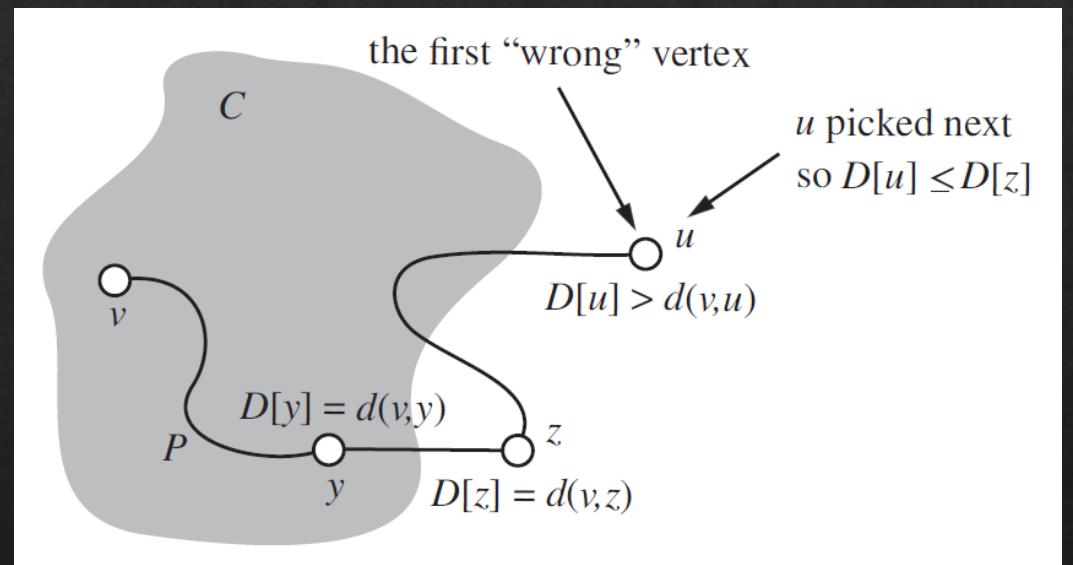
Lemma

- ❖ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ."
- ❖ Proof by contradiction:
 - ❖ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$
 - ❖ We know there exists a shortest path P .
 - ❖ Let z be the first vertex of P going from v to u and z is not yet in C



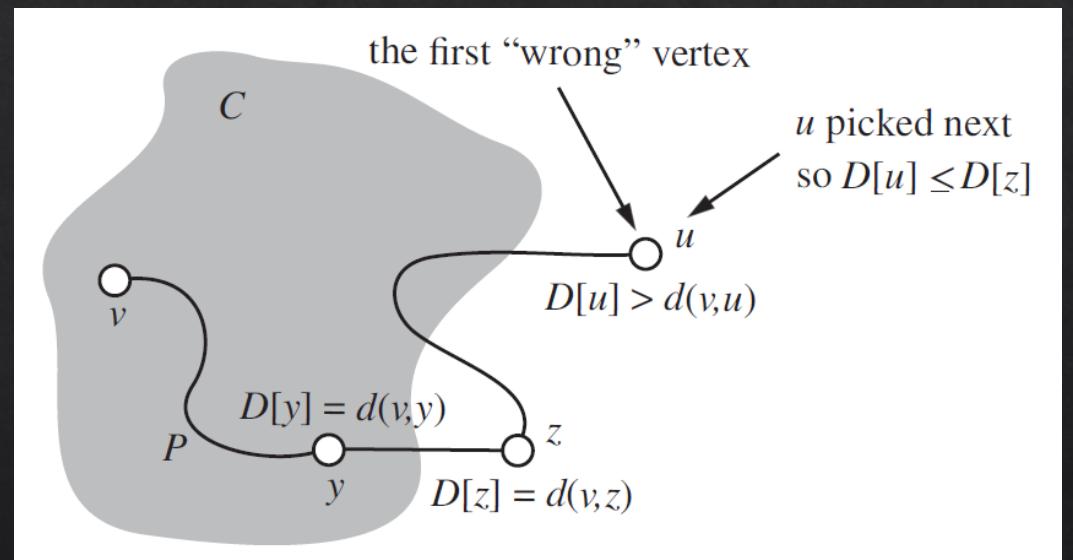
Lemma

- ❖ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ."
- ❖ Proof by contradiction:
 - ❖ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$
 - ❖ We know there exists a shortest path P .
 - ❖ Let z be the first vertex of P going from v to u and z is not yet in C . By our choice of z ,
 - ❖ $D[z] = d(v, z)$
 - ❖ z 's predecessor y is already in C .



Lemma

- ❖ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ."
- ❖ Proof by contradiction:
 - ❖ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$
 - ❖ We know there exists a shortest path P .
 - ❖ Let z be the first vertex of P going from v to u and z is not yet in C . By our choice of z ,
 - ❖ $D[z] = d(v, z)$
 - ❖ z 's predecessor y is already in C . This means:
 - ❖ $D[y] = d(v, y)$



Lemma

- ◆ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ."

- ◆ Proof by contradiction:

- ◆ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$

- ◆ We know there exists a shortest path P .

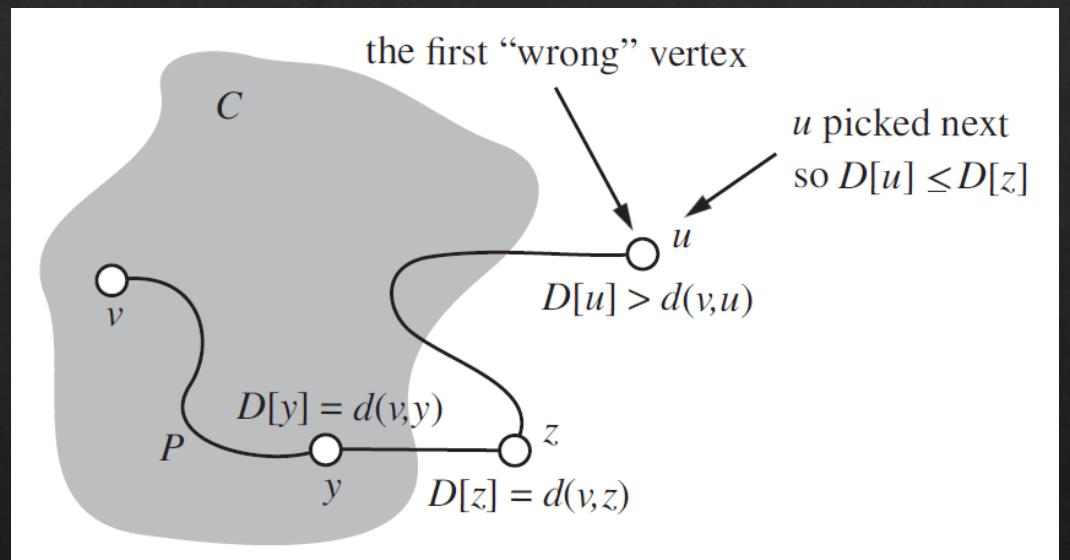
- ◆ Let z be the first vertex of P going from v to u and z is not yet in C . By our choice of z ,

- ◆ $D[z] = d(v, z)$

- ◆ z 's predecessor y is already in C . This means:

- ◆ $D[y] = d(v, y)$

- ◆ $D[z] \leq D[y] + w((y, z)) = d(v, y) + w((y, z))$



Lemma

- ❖ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ."

- ❖ Proof by contradiction:

❖ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$

❖ We know there exists a shortest path P .

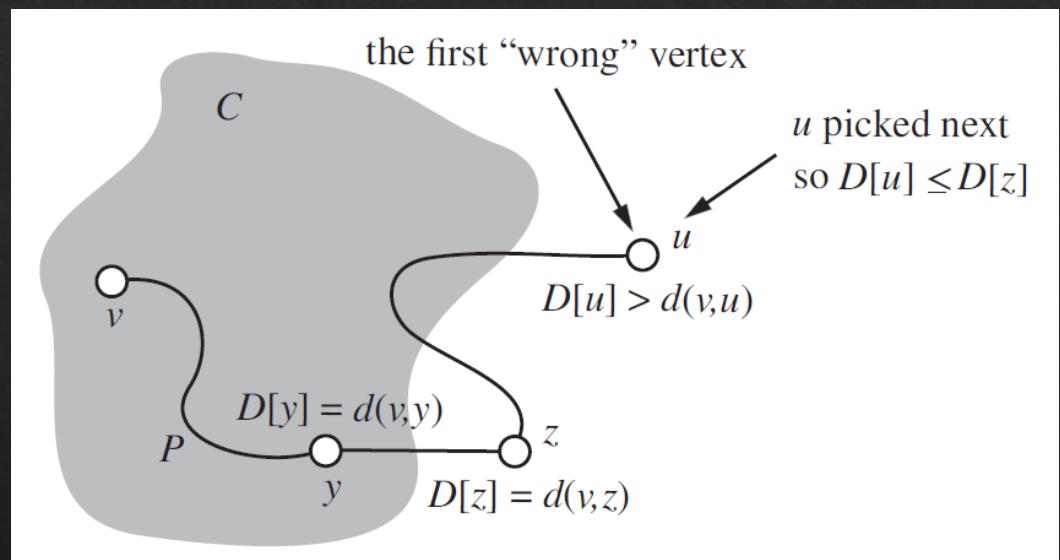
❖ Let z be the first vertex of P going from v to u and z is not yet in C . By our choice of z ,

❖ $D[z] = d(v, z)$

❖ z 's predecessor y is already in C . This means:

❖ $D[y] = d(v, y)$

❖ $D[z] \leq D[y] + w((y, z)) = d(v, y) + w((y, z))$



❖ We pick u and not z : $D[u] \leq D[z]$

❖ But $d(v, z) + d(z, u) = d(v, u)$

Lemma

- ❖ Prove that "In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u ."

- ❖ Proof by contradiction:

❖ Suppose u is the first vertex pulled in to C but $D[u] > d(v, u)$

❖ We know there exists a shortest path P .

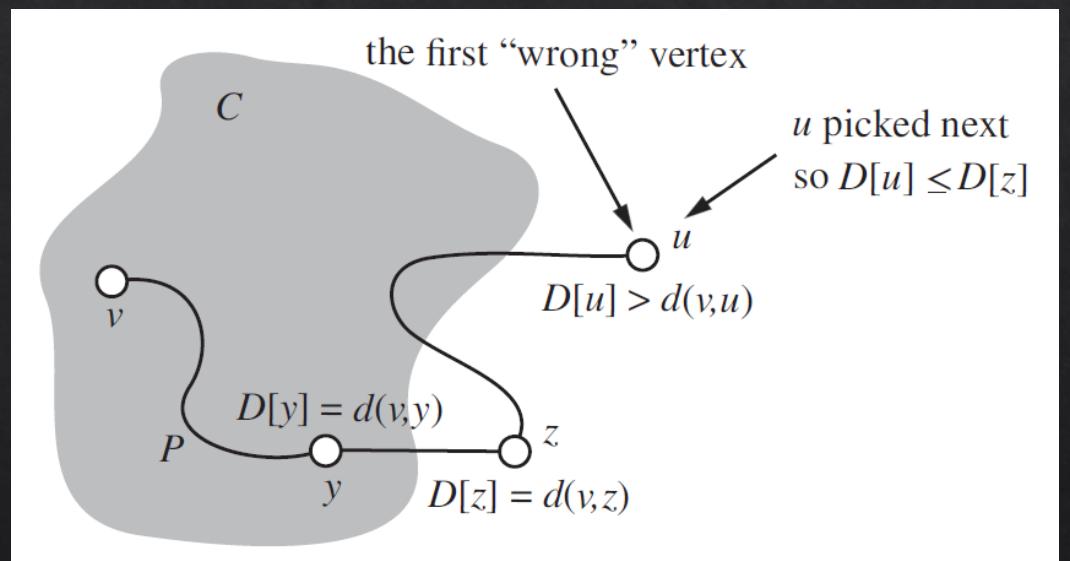
❖ Let z be the first vertex of P going from v to u and z is not yet in C . By our choice of z ,

❖ $D[z] = d(v, z)$

❖ z 's predecessor y is already in C . This means:

❖ $D[y] = d(v, y)$

❖ $D[z] \leq D[y] + w((y, z)) = d(v, y) + w((y, z))$



- ❖ We pick u and not z : $D[u] \leq D[z]$
- ❖ But $d(v, z) + d(z, u) = d(v, u)$. Therefore,
- $$D[u] \leq D[z] = d(v, z) \leq d(v, z) + d(z, u) = d(v, u) \leftarrow \text{contradiction}$$

The Running Time of Dijkstra's Algorithm

$$|V| = n, |E| = m$$

From v:

G: Adjacency list structure.

Q: Heap, key $D[u]$ for all u from V . removeMin in $O(\log n)$ time.

Keep an array to access to all the keys of the vertices in the heap Q .

Update $D[z]$ by first removing object containing z then insert z with new $D[z]$ via heapification in $O(\log n)$ time

- I. One way of implementation:

The Running Time of Dijkstra's Algorithm

$$|V| = n, |E| = m$$

From v:

G: Adjacency list structure.

Q: Heap, key $D[u]$ for all u from v. removeMin in $O(\log n)$ time.

Keep an array to access to all the keys of the vertices in the heap Q.

Update $D[z]$ by first removing object containing z then insert z with new $D[z]$ via heapification in $O(\log n)$ time

I. One way of implementation:

1. Inserting all the vertices in Q with their initial $D[]$: $O(n \log n)$

2. Inside the while loop:

1. Remove u and maintain heap: $O(\log n)$

2. Relaxation procedure for all adjacent edges of u: $O(\deg(u) \log n)$

$$D[v] \leftarrow 0$$

for each vertex $u \neq v$ of G **do**
 $D[u] \leftarrow +\infty$

Let a priority queue, Q , contain all the vertices of G using the D labels as keys.
while Q is not empty **do**

// pull a new vertex u into the cloud

$$u \leftarrow Q.\text{removeMin}()$$

for each vertex z adjacent to u such that z is in Q **do**

// perform the **relaxation** procedure on edge (u, z)

if $D[u] + w((u, z)) < D[z]$ **then**

$$D[z] \leftarrow D[u] + w((u, z))$$

Change the key for vertex z in Q to $D[z]$

return the label $D[u]$ of each vertex u

The Running Time of Dijkstra's Algorithm

$$|V| = n, |E| = m$$

From v:

G: Adjacency list structure.

Q: Heap, key $D[u]$ for all u from v. removeMin in $O(\log n)$ time.

Keep an array to access to all the keys of the vertices in the heap Q.

Update $D[z]$ by first removing object containing z then insert z with new $D[z]$ via heapification in $O(\log n)$ time

I. One way of implementation:

1. Inserting all the vertices in Q with their initial $D[]$: $O(n \log n)$

2. Inside the while loop:

1. Remove u and maintain heap: $O(\log n)$

2. Relaxation procedure for all adjacent edges of u: $O(\deg(u) \log n)$

3. Total while loop:

$$\sum_{u \in G} \log n + \deg(u) \log n = \sum_{u \in G} (1 + \deg(u)) \log n, \text{ which is } O((n+m) \log n)$$

```
 $D[v] \leftarrow 0$ 
for each vertex  $u \neq v$  of  $G$  do
     $D[u] \leftarrow +\infty$ 
Let a priority queue,  $Q$ , contain all the vertices of  $G$  using the  $D$  labels as keys.
while  $Q$  is not empty do
    // pull a new vertex  $u$  into the cloud
     $u \leftarrow Q.\text{removeMin}()$ 
    for each vertex  $z$  adjacent to  $u$  such that  $z$  is in  $Q$  do
        // perform the relaxation procedure on edge  $(u, z)$ 
        if  $D[u] + w((u, z)) < D[z]$  then
             $D[z] \leftarrow D[u] + w((u, z))$ 
            Change the key for vertex  $z$  in  $Q$  to  $D[z]$ 
return the label  $D[u]$  of each vertex  $u$ 
```

The Running Time of Dijkstra's Algorithm

$$|V| = n, |E| = m$$

$$\sum_{u \in G} \log n + \deg(u) \log n = \sum_{u \in G} (1 + \deg(u)) \log n, \text{ which is } O((n+m) \log n)$$

If $m \gg n$, total run time is $O(m \log n)$

```
D[v] ← 0
for each vertex  $u \neq v$  of  $G$  do
     $D[u] \leftarrow +\infty$ 
Let a priority queue,  $Q$ , contain all the vertices of  $G$  using the  $D$  labels as keys.
while  $Q$  is not empty do
    // pull a new vertex  $u$  into the cloud
     $u \leftarrow Q.\text{removeMin}()$ 
    for each vertex  $z$  adjacent to  $u$  such that  $z$  is in  $Q$  do
        // perform the relaxation procedure on edge  $(u, z)$ 
        if  $D[u] + w((u, z)) < D[z]$  then
             $D[z] \leftarrow D[u] + w((u, z))$ 
            Change the key for vertex  $z$  in  $Q$  to  $D[z]$ 
return the label  $D[u]$  of each vertex  $u$ 
```

The Running Time of Dijkstra's Algorithm

$|V| = n, |E| = m$

$$\sum_{u \in G} \log n + \deg(u) \log n = \sum_{u \in G} (1 + \deg(u)) \log n, \text{ which is } O((n+m) \log n)$$

If $m \gg n$, total run time is $O(m \log n)$

(This is just the running time of one way to do this.)

```
D[v] ← 0
for each vertex  $u \neq v$  of  $G$  do
     $D[u] \leftarrow +\infty$ 
Let a priority queue,  $Q$ , contain all the vertices of  $G$  using the  $D$  labels as keys.
while  $Q$  is not empty do
    // pull a new vertex  $u$  into the cloud
     $u \leftarrow Q.\text{removeMin}()$ 
    for each vertex  $z$  adjacent to  $u$  such that  $z$  is in  $Q$  do
        // perform the relaxation procedure on edge  $(u, z)$ 
        if  $D[u] + w((u, z)) < D[z]$  then
             $D[z] \leftarrow D[u] + w((u, z))$ 
            Change the key for vertex  $z$  in  $Q$  to  $D[z]$ 
return the label  $D[u]$  of each vertex  $u$ 
```

Negative-weight Edges

So far, we have avoided negative edge weights (no negative-weight cycle)

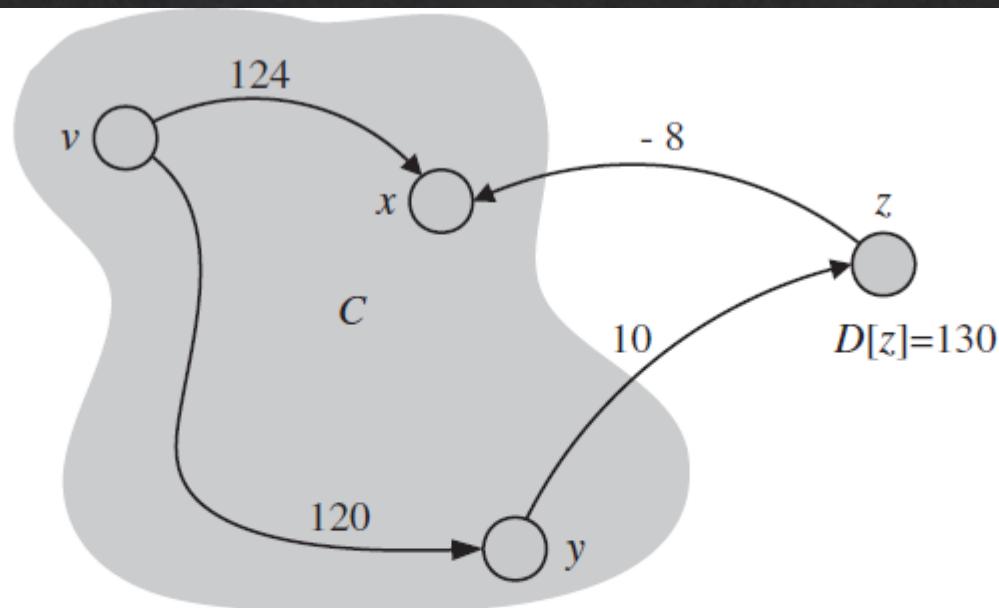


Figure 14.6: An illustration of why Dijkstra's algorithm fails for graphs with negative-weight edges. Bringing z into C and performing edge relaxations will invalidate the previously computed shortest-path distance (124) to x .

The Bellman-Ford Algorithm

To consider negative edge weights, the graph must be directed
otherwise shortest path does not make sense

Algorithm BellmanFordShortestPaths(\vec{G}, v):

Input: A weighted directed graph \vec{G} with n vertices, and a vertex v of \vec{G}

Output: A label $D[u]$, for each vertex u of \vec{G} , such that $D[u]$ is the distance
from v to u in \vec{G} , or an indication that \vec{G} has a negative-weight cycle

$D[v] \leftarrow 0$

for each vertex $u \neq v$ of \vec{G} **do**

$D[u] \leftarrow +\infty$

for $i \leftarrow 1$ to $n - 1$ **do**

for each (directed) edge (u, z) outgoing from u **do**

 // Perform the **relaxation** operation on (u, z)

if $D[u] + w((u, z)) < D[z]$ **then**

$D[z] \leftarrow D[u] + w((u, z))$

if there are no edges left with potential relaxation operations **then**

return the label $D[u]$ of each vertex u

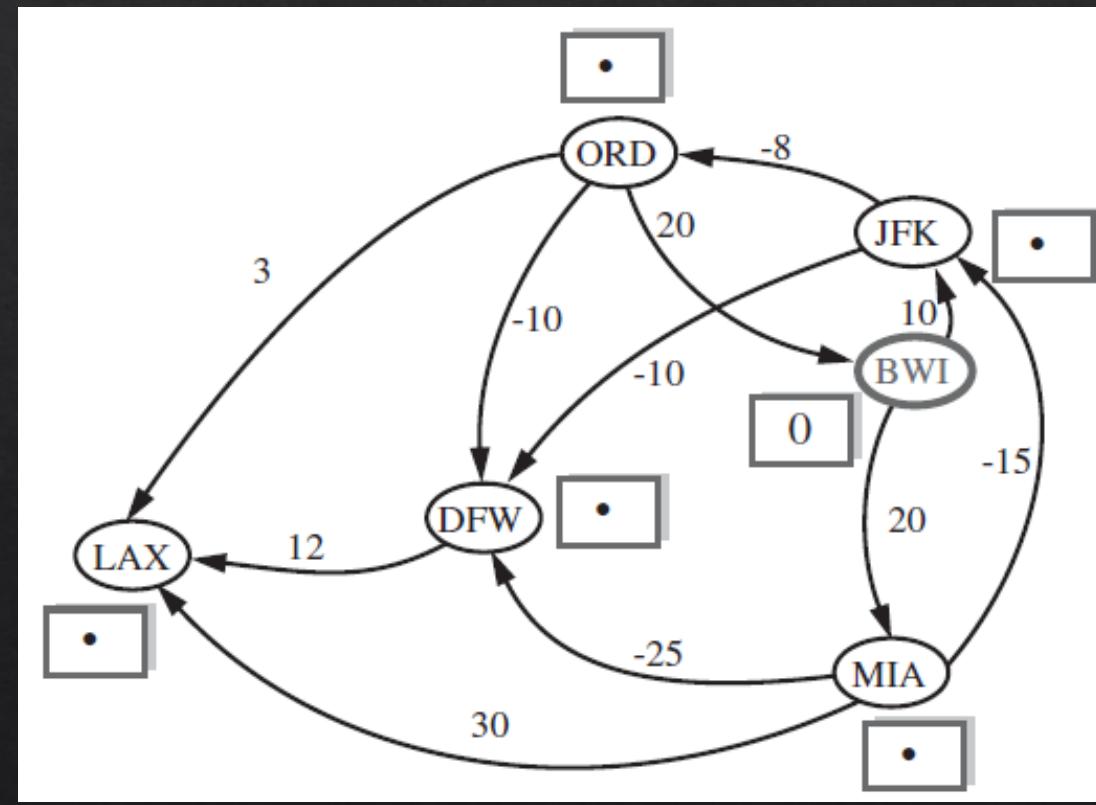
else

return “ \vec{G} contains a negative-weight cycle”

The Bellman-Ford Algorithm

From BWI

D[BWI]	D[LAX]	D[DFW]	D[ORD]	D[JFK]	D[MIA]
0	+∞	+∞	+∞	+∞	+∞



The Bellman-Ford Algorithm

From BWI

$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20

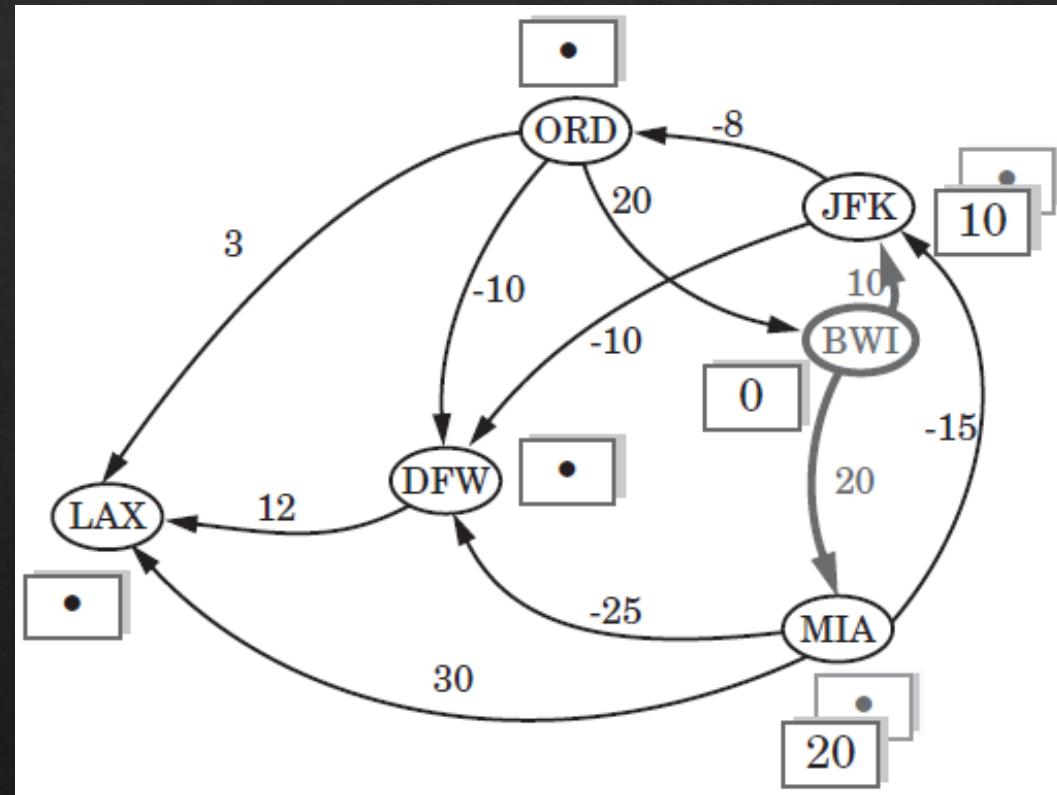
Adjacency list: (BWI, JFK) , (BWI, MIA)

Consider (BWI, JFK)

$$D[BWI] + W((BWI, JFK)) = 0 + 10 = 10 < D[JFK] = +\infty \\ \rightarrow D[JFK] = 10$$

Consider (BWI, MIA)

$$D[MIA] + W((BWI, MIA)) = 0 + 20 = 20 < D[MIA] = +\infty \\ \rightarrow D[MIA] = 20$$



The Bellman-Ford Algorithm

From BWI

$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20
	50	-5		5	

Adjacency list FOR MIA: (MIA,DFW), (MIA, JFK), (MIA,LAX)

Consider (MIA, DFW)

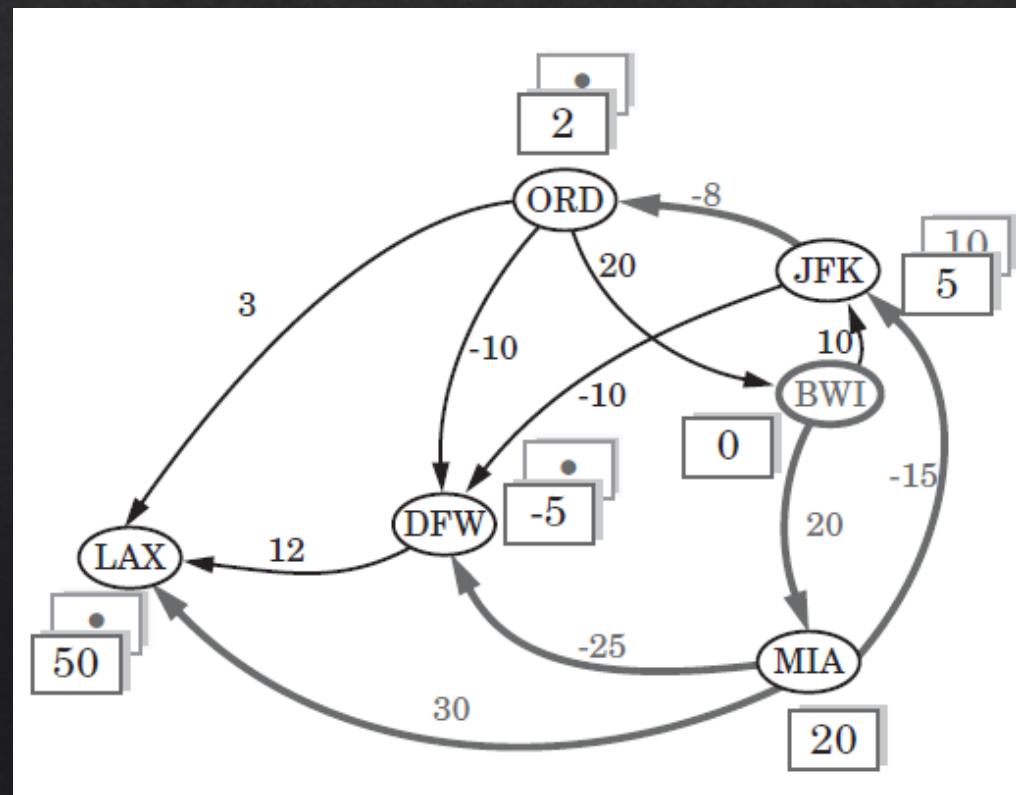
$$D[MIA] + w((MIA, DFW)) = 20 + -25 = -5 < D[DFW] = +\infty \\ \rightarrow D[JFK] = -5$$

Consider (MIA,JFK)

$$D[MIA] + w((MIA, JFK)) = 20 + -15 = 5 < D[MIA] = 10 \\ \rightarrow D[JFK] = 5$$

Consider (MIA,LAX)

$$D[MIA] + w((MIA, LAX)) = 20 + 30 = 50 < D[LAX] = +\infty \\ \rightarrow D[LAX] = 50$$



The Bellman-Ford Algorithm

From BWI

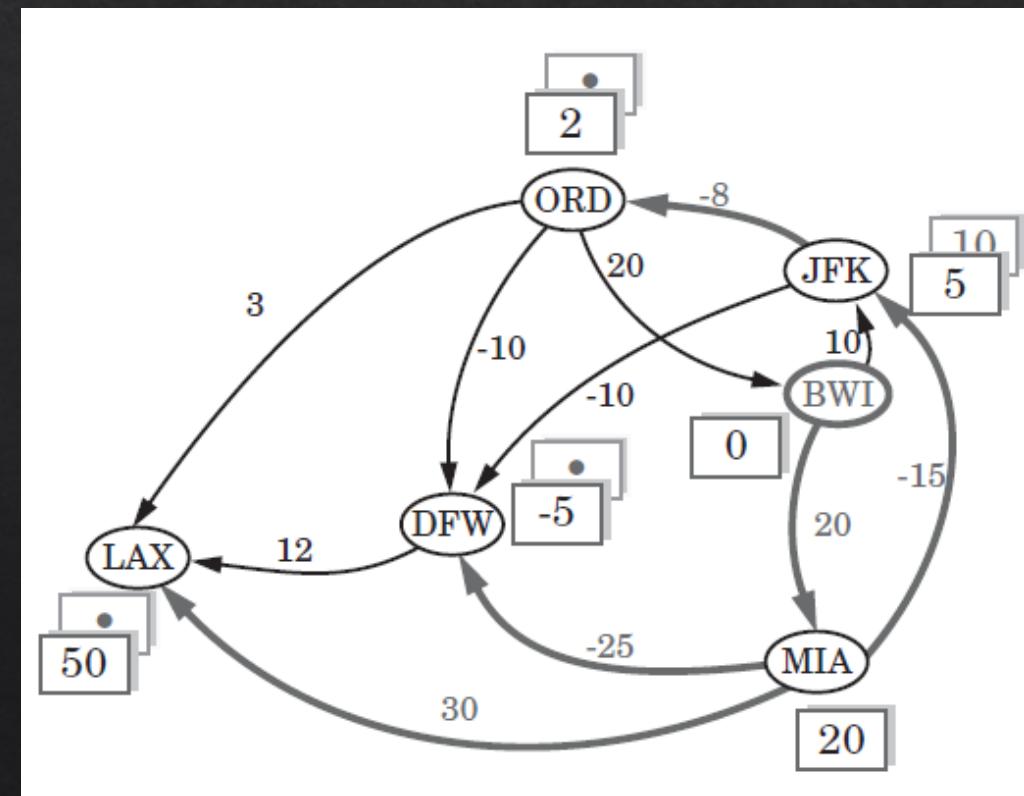
$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20
50	-5	2	5		

Adjacency list FOR JFK: (JFK, ORD)

Consider (JFK, ORD)

$$D[JFK] + w(JFK, ORD) = 10 + -8 = 2 < D[ORD] = +\infty \\ \rightarrow D[ORD] = 2$$

* Use original previous round.



The Bellman-Ford Algorithm

From BWI

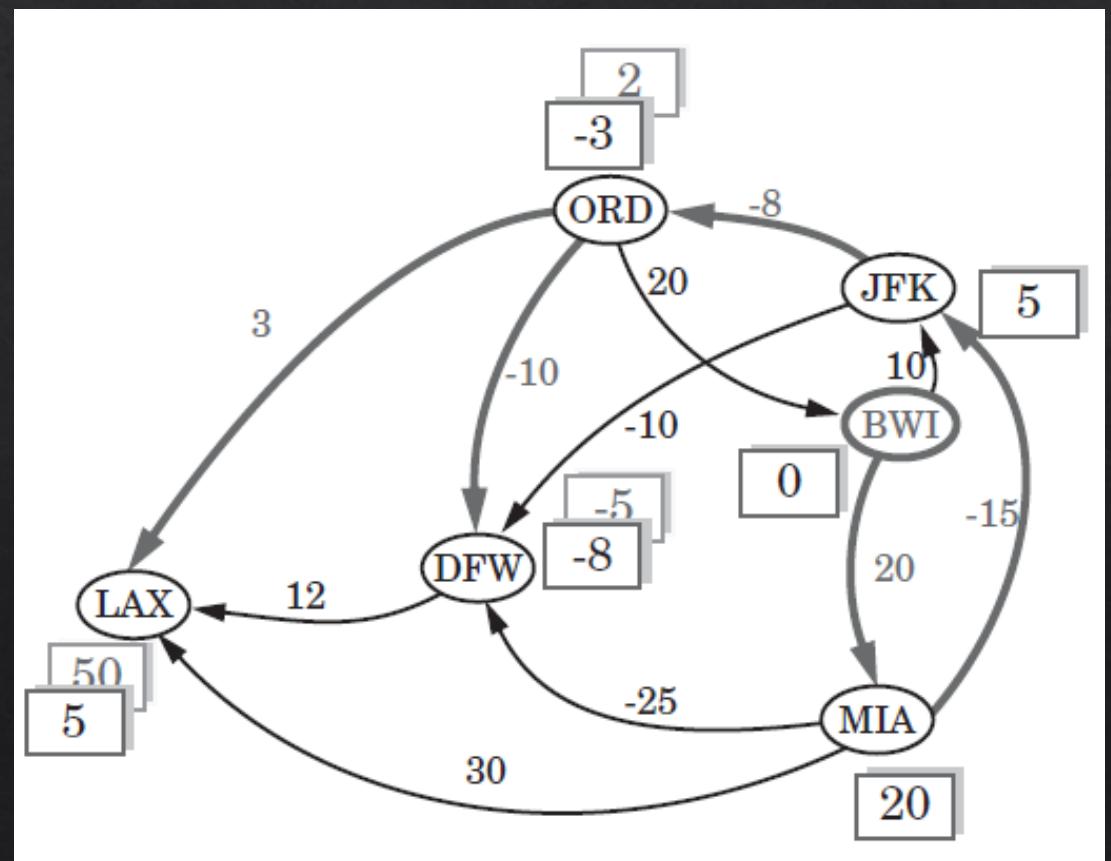
$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20
	50	-5	2	5	
			-3		

Adjacency list FOR JFK: (JFK, ORD)

Consider (JFK, ORD)

$$D[JFK] + w((JFK, ORD)) = 5 + -8 = -3 < D[ORD] = 2 \\ \rightarrow D[ORD] = -3$$

Use relaxed



The Bellman-Ford Algorithm

From BWI

$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20
	50	-5	2	5	
	5	-8	-3		

Adjacency list FOR ORD: (ORD, LAX), (ORD, DFW)

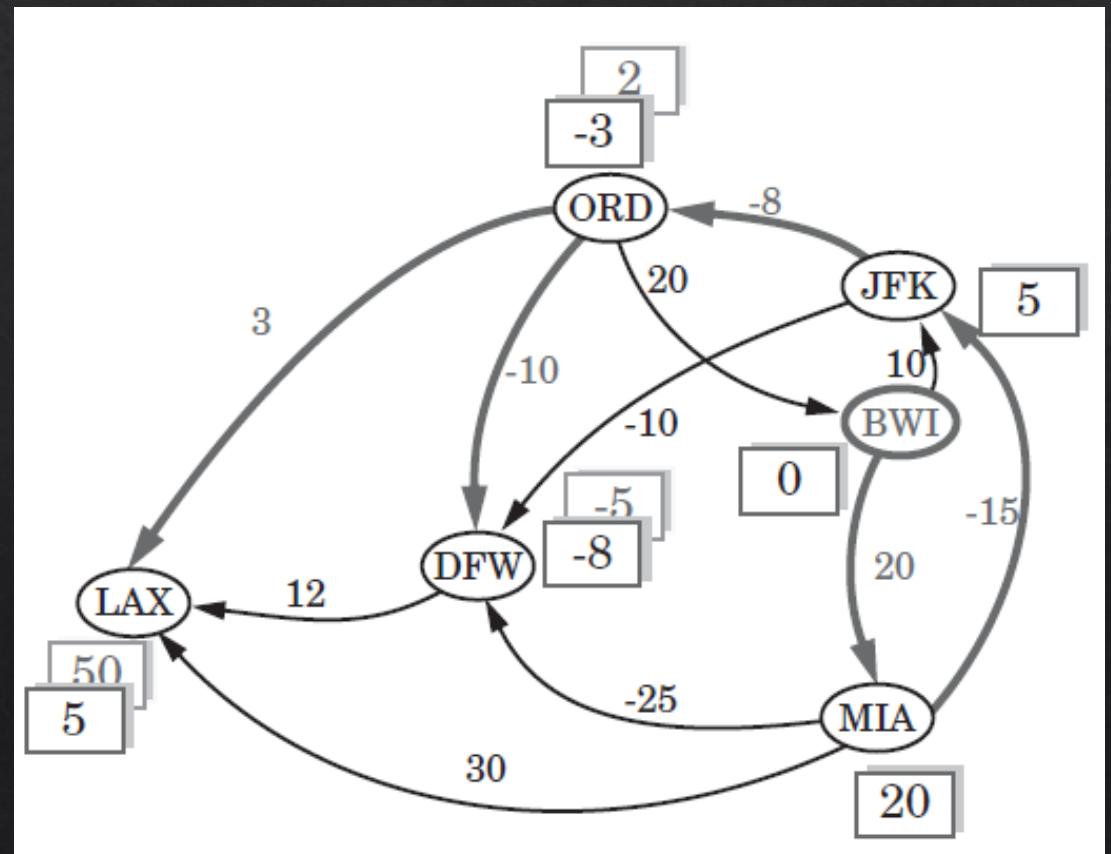
Consider (ORD, LAX)

$$D[ORD] + w(ORD, LAX) = 2 + 3 = 5 < D[LAX] = 50 \\ \rightarrow D[ORD] = 5$$

Consider (ORD, DFW)

$$D[ORD] + w(ORD, DFW) = 2 + -10 = -8 < D[DFW] = -5 \\ \rightarrow D[ORD] = -8$$

* Use original previous round.



The Bellman-Ford Algorithm

From BWI

$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20
	50	-5	2	5	
	5	-8	-3		
	4	-13			

Adjacency list for ORD: (ORD, DFW)

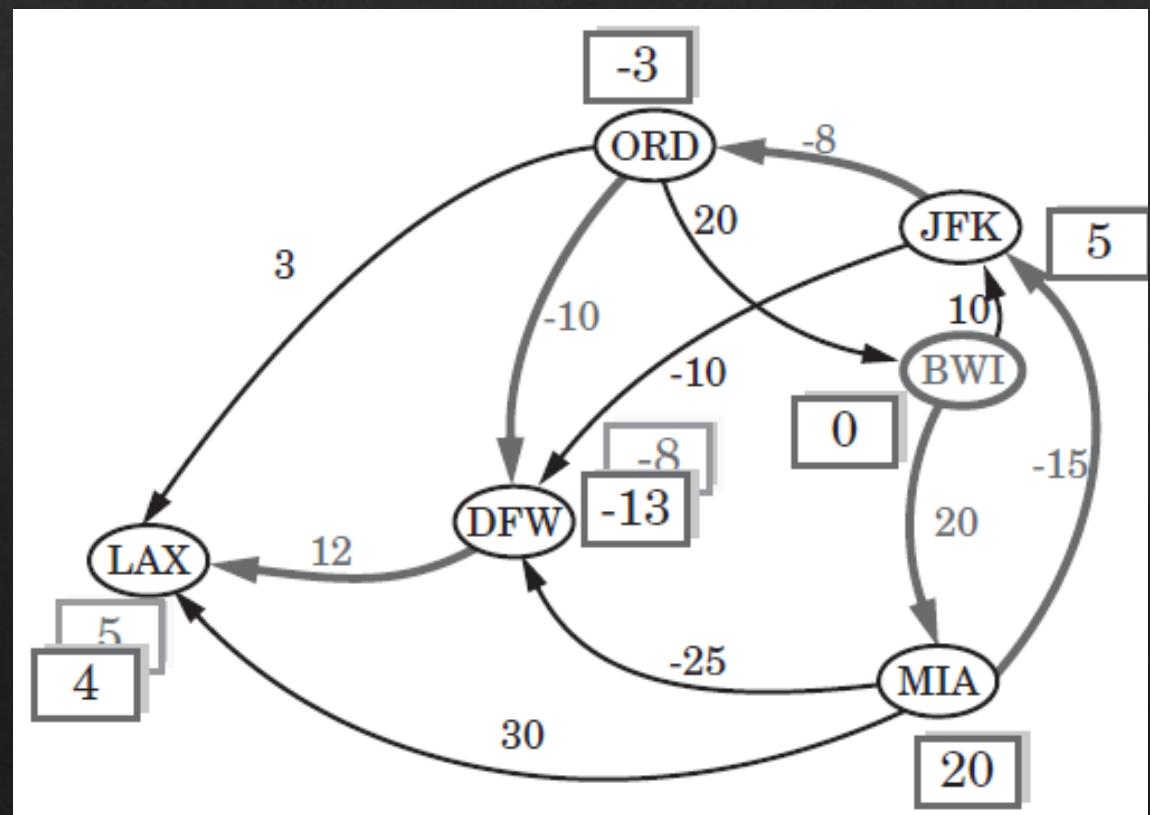
Consider (ORD, DFW)

$$D[ORD] + w(ORD, DFW) = -3 + -10 = -13 < D[DFW] = -8 \\ \rightarrow D[DFW] = -13$$

Adjacency list FOR DFW: (DFW, LAX)

Consider (DFW, LAX)

$$D[DFW] + w(DFW, LAX) = -8 + 12 = 4 < D[LAX] = 5 \\ \rightarrow D[LAX] = 4$$



The Bellman-Ford Algorithm

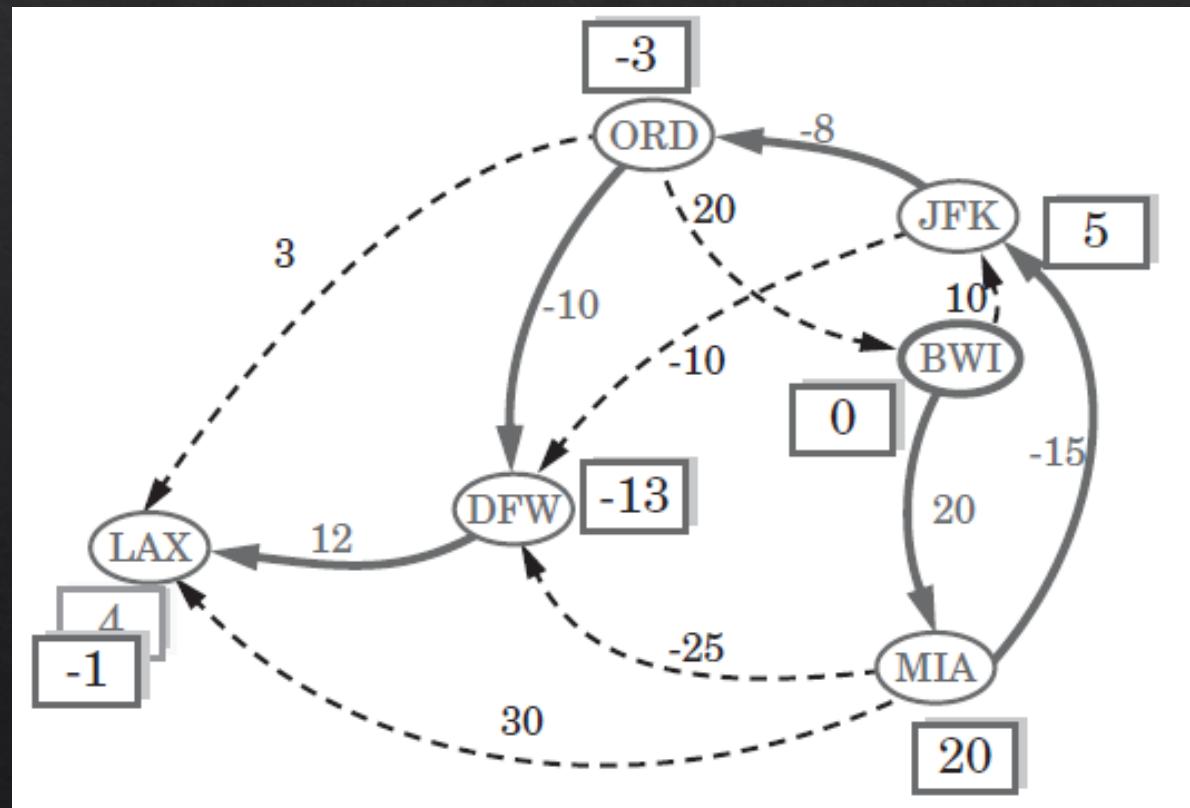
From BWI

$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20
50	-5	2	5		
5	-8	-3			
4	-13				
-1					

Adjacency list FOR DFW: (DFW, LAX)

Consider (DFW, LAX)

$$D[DFW] + w(DFW, LAX) = -13 + 12 = -1 < D[LAX] = 4 \\ \rightarrow D[ORD] = -1$$



The Bellman-Ford Algorithm

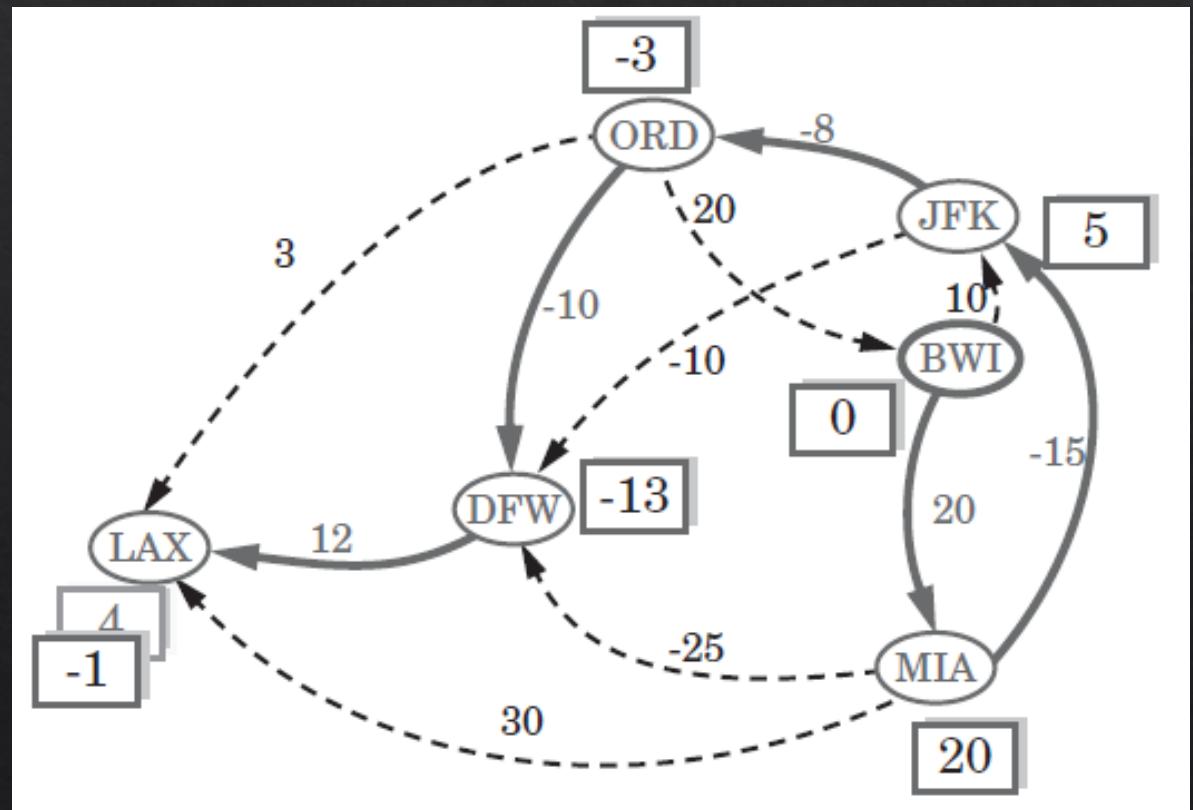
From BWI

$D[BWI]$	$D[LAX]$	$D[DFW]$	$D[ORD]$	$D[JFK]$	$D[MIA]$
0	$+\infty$	$+\infty$	$+\infty$	10	20
50	-5	2	5		
5	-8	-3			
4	-13				
-1					

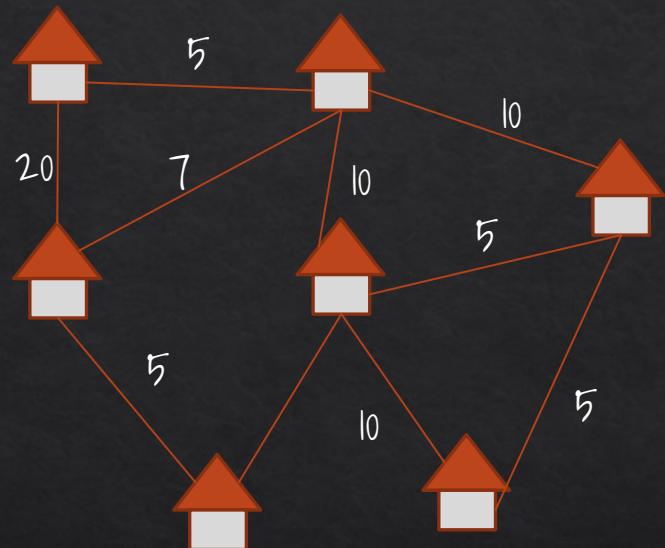
Adjacency list FOR DFW: (DFW, LAX)

Consider (DFW, LAX)

$$D[DFW] + w((DFW, LAX)) = -13 + 12 = -1 < D[LAX] = 4 \\ \rightarrow D[ORD] = -1$$



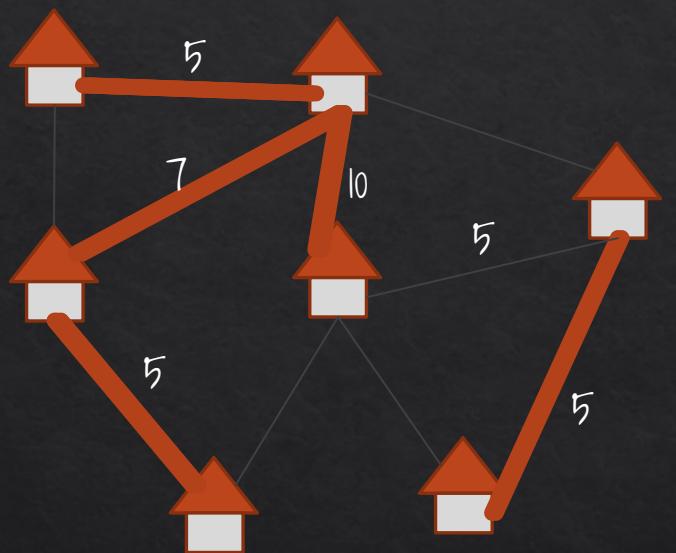
Spanning Tree



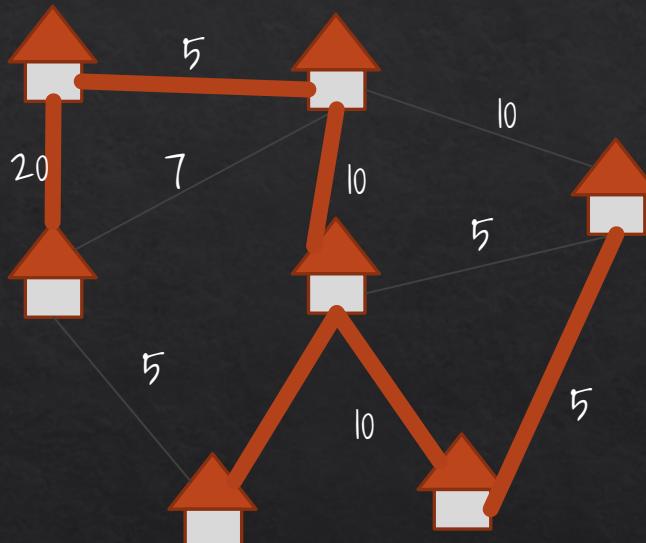
I. Requirement:

1. Run Internet cable through the neighborhood.
2. Interconnect all the houses as cheaply as possible.

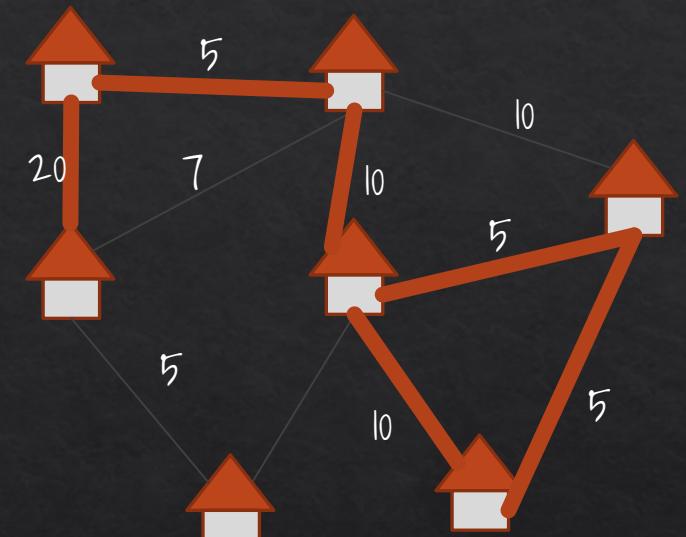
Spanning Tree



Two Trees



One Spanning Tree



Not a Tree

Definition: A tree that contain all the vertices in a weighted undirected graph G is a spanning tree.

Minimum Spanning Tree

Definition: A spanning tree that minimizes the sum of the weights of the edges of T.

$$w(T) = \sum_{e \in T} w(e),$$

How many edges does a spanning tree T have?

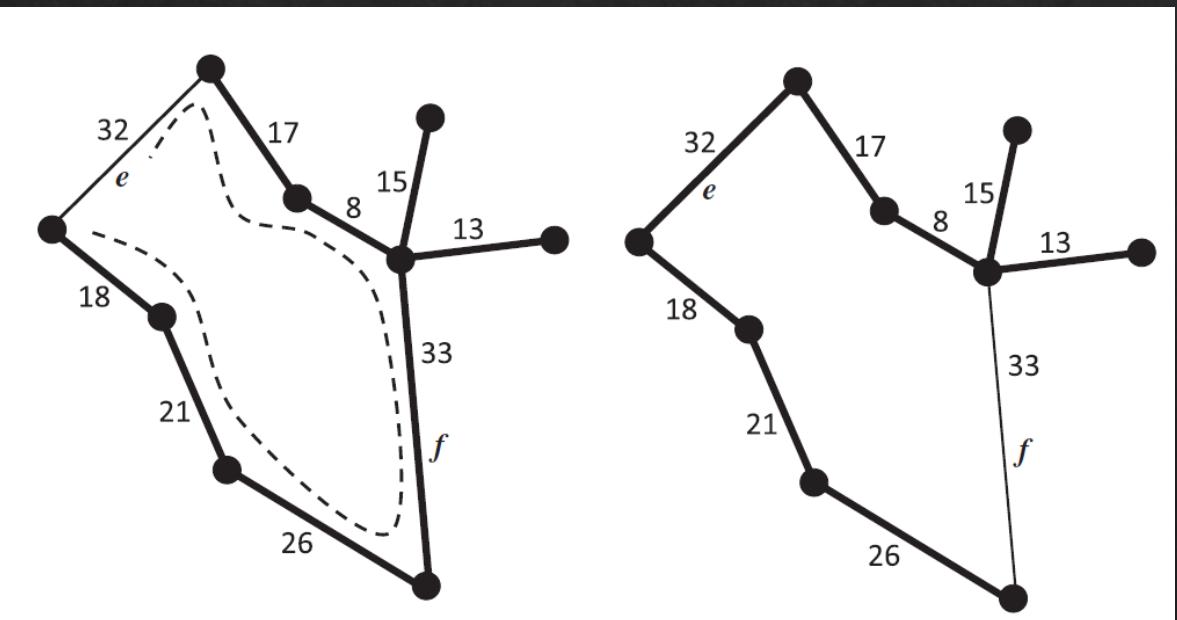
Properties of Minimum Spanning Trees

Lemma: Let G be a weighted connected graph, and let T be a minimum spanning tree for G .

If e is an edge of G that is not in T , then

the weight of e is at least as great as any edge in the cycle created by adding e to T .

Proof by contradiction:



Suppose, for the sake of contradiction, that there is an edge, f , in T
 $w(e) < w(f)$

Then we can remove f from T and replace it with e , and this will result in a spanning tree, T_-

$$w(T_-) < w(T)$$

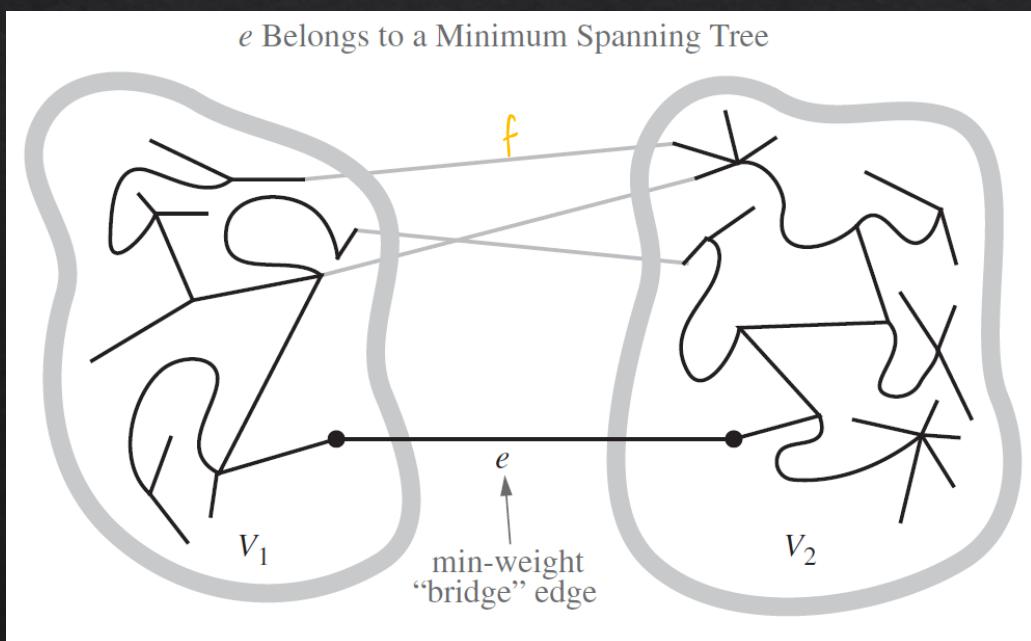
But the existence of such a tree, T_- , would contradict the fact that T is a minimum spanning tree.

So no such edge, f , can exist.

Properties of Minimum Spanning Trees

Theorem: Let G be a weighted connected graph. Let G be a weighted connected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2 . There exists a minimum spanning tree T that has e as one of its edges.

Proof by contradiction:



Suppose, for the sake of contradiction, that there is an edge, f , in T
 $w(e) < w(f)$

Then we can remove f from T and replace it with e , and this will result in a spanning tree, T_-

$$w(T_-) < w(T)$$

But the existence of such a tree, T_- , would contradict the fact that T is a minimum spanning tree.

So no such edge, f , can exist.

Kruskal's Algorithm

- ❖ Previous theorem leads Kruskal's algorithm
 - ❖ Find the minimum spanning tree T for connected weighted graph G with n vertices and m edges.
 - ❖ Data structures for this algo:
 - ❖ A priority Q storing e_i . Keys of Q are w_e
 - ❖ Sets, each of which is a vertex $\{u\}$
 - ❖ A $(n-1)$ array to store e of T

Kruskal's Algorithm

Algorithm KruskalMST(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

 Define an elementary cluster $C(v) \leftarrow \{v\}$.

Let Q be a priority queue storing the edges in G , using edge weights as keys

$T \leftarrow \emptyset$ // T will ultimately contain the edges of the MST

while T has fewer than $n - 1$ edges **do**

$(u, v) \leftarrow Q.\text{removeMin}()$

 Let $C(v)$ be the cluster containing v

 Let $C(u)$ be the cluster containing u

if $C(v) \neq C(u)$ **then**

 Add edge (v, u) to T

 Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$

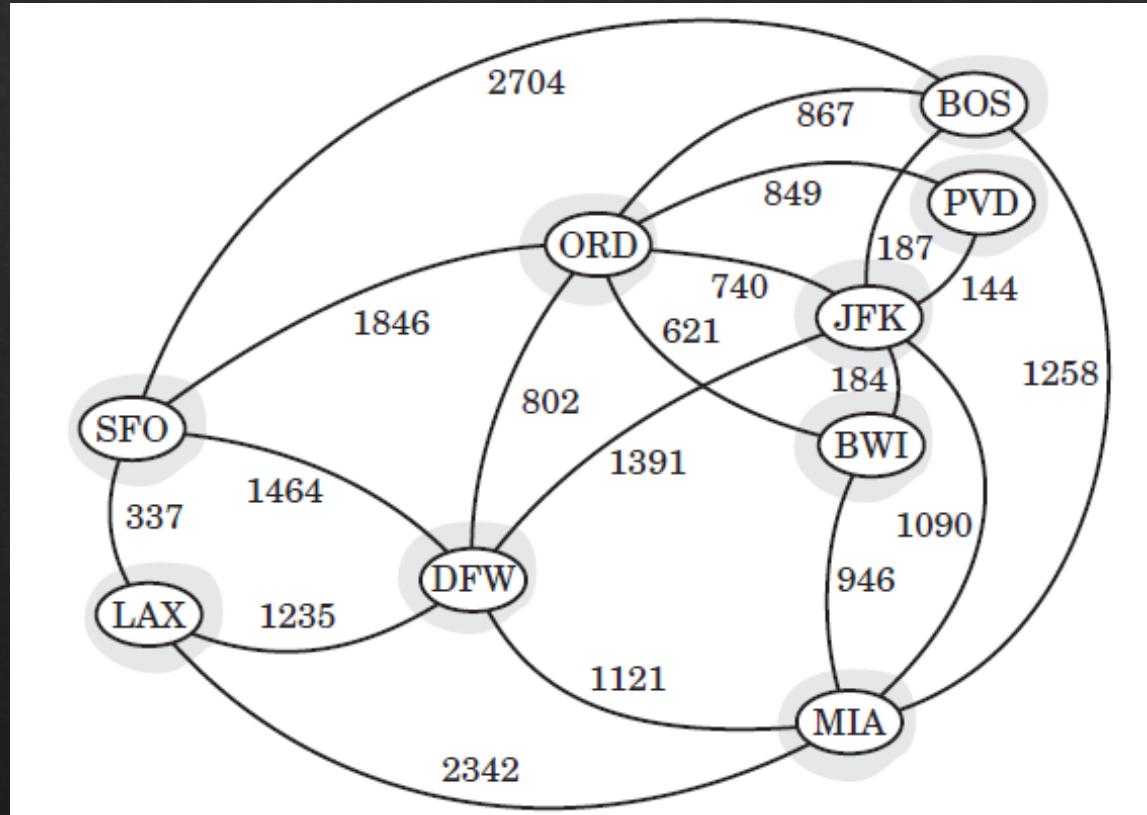
return tree T

◆ Kruskal's algorithm always adds a valid minimum spanning tree edge. Why?

{BOS}, {BWI}, {SFO}, {LAX}, {DFW}, {ORD}, {PVD}, {JFK}, {MIA}

Kruskal's Algorithm Example

Edge (Weights)
(JFK, PVD) 144
(BWI, JFK) 184
(BOS, JFK) 187
(LAX, SFO) 337
(BWI, ORD) 621
(JFK, ORD) 740
(DFW, ORD) 802
(ORD, PVD) 849
(BOS, ORD) 867
(BWI, MIA) 946
(JFK, MIA) 1090
(DFW, MIA) 1121
(DFW, LAX) 1235
(BOS, MIA) 1258
(DFW, SFO) 1464
(LAX, MIA) 2342
(BOS, SFO) 2704



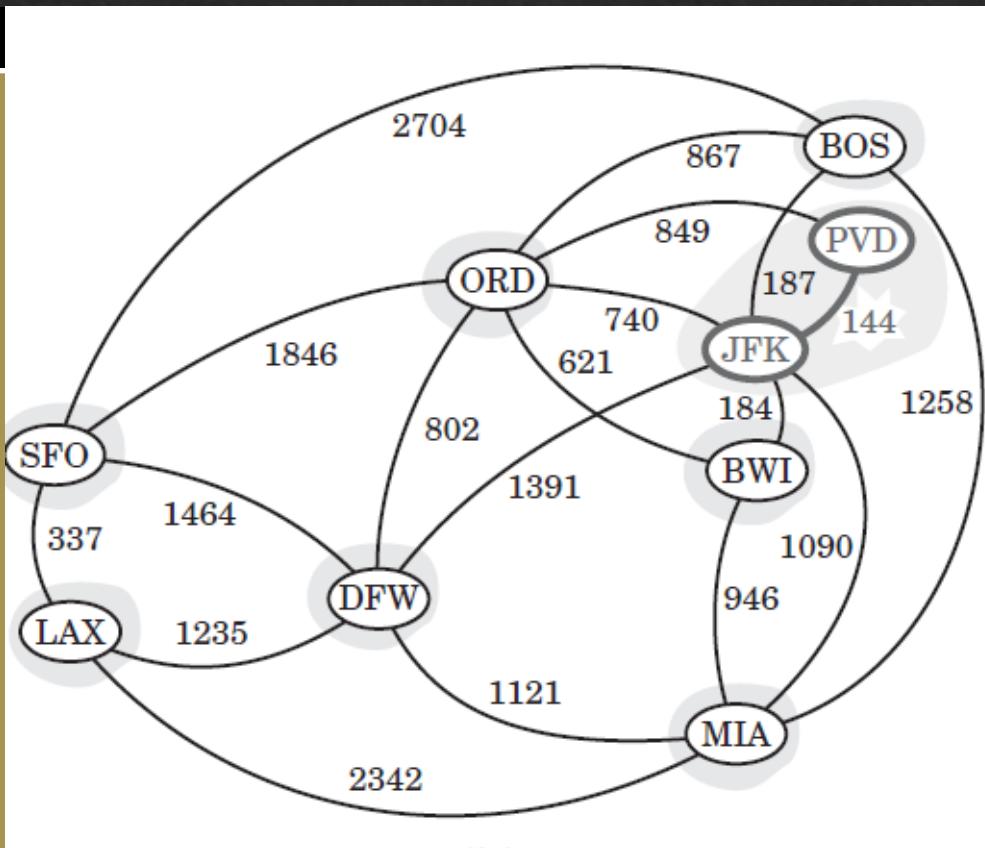
Kruskal for Minimum Spanning Tree

{BOS}, {BWI}, {SFO}, {LAX}, {DFW}, {ORD}, {PVD, JFK}, {MIA}

❖ Priority Q

$$T[0] = (\text{JFK}, \text{PVD})$$

Edge (Weights)
(JFK, PVD) 144
(BWI, JFK) 184
(BOS, JFK) 187
(LAX, SFO) 337
(BWI, ORD) 621
(JFK, ORD) 740
(DFW, ORD) 802
(ORD, PVD) 849
(BOS, ORD) 867
(BWI, MIA) 946
(JFK, MIA) 1090
(DFW, MIA) 1121
(DFW, LAX) 1235
(BOS, MIA) 1258
(DFW, SFO) 1464
(LAX, MIA) 2342
(BOS, SFO) 2704

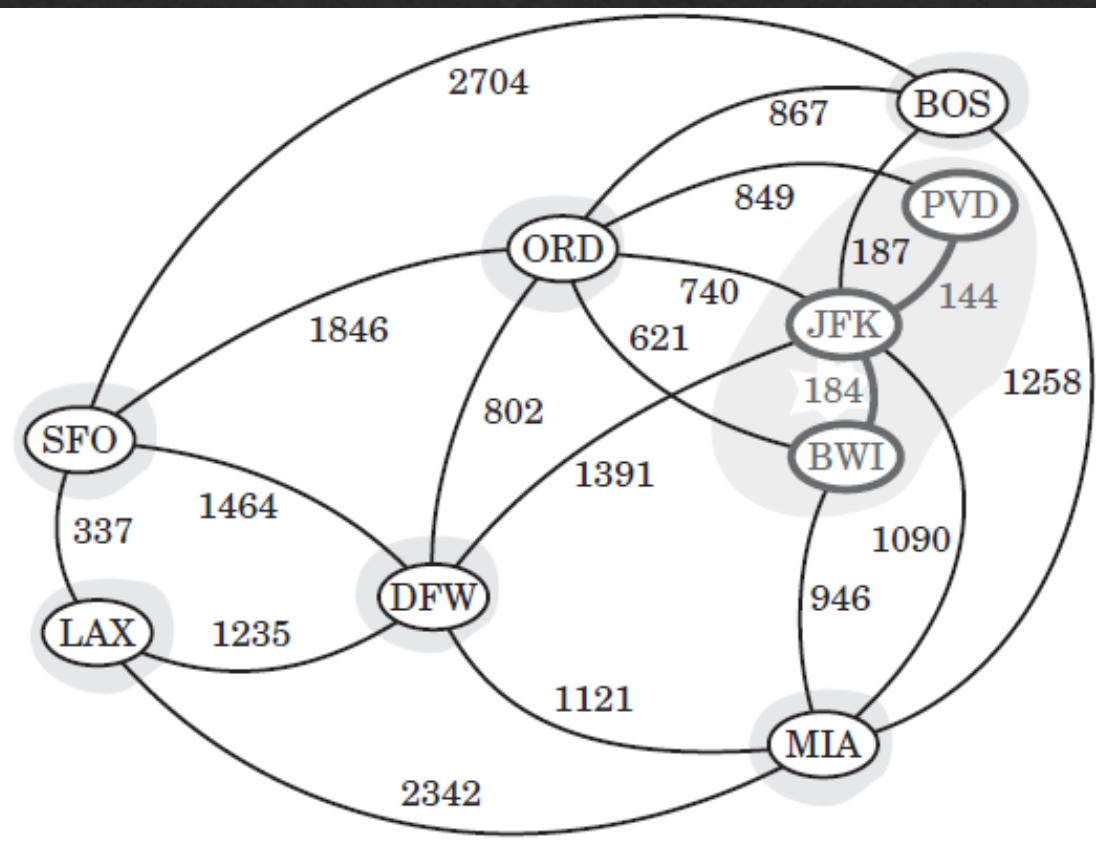


Kruskal for Minimum Spanning Tree

{BOS} {SFO} {LAX} {DFW}, {ORD} {PVD, JFK, BWI} {MIA}

❖ Priority Q

Edge (Weights)
(JFK, PVD) 144
(BWI, JFK) 184
(BOS, JFK) 187
(LAX, SFO) 337
(BWI, ORD) 621
(JFK, ORD) 740
(DFW, ORD) 802
(ORD, PVD) 849
(BOS, ORD) 867
(BWI, MIA) 946
(JFK, MIA) 1090
(DFW, MIA) 1121
(DFW, LAX) 1235
(BOS, MIA) 1258
(DFW, SFO) 1464
(LAX, MIA) 2342
(BOS, SFO) 2704



$$T[0] = (\text{JFK, PVD})$$
$$T[1] = (\text{BWI, JFK})$$

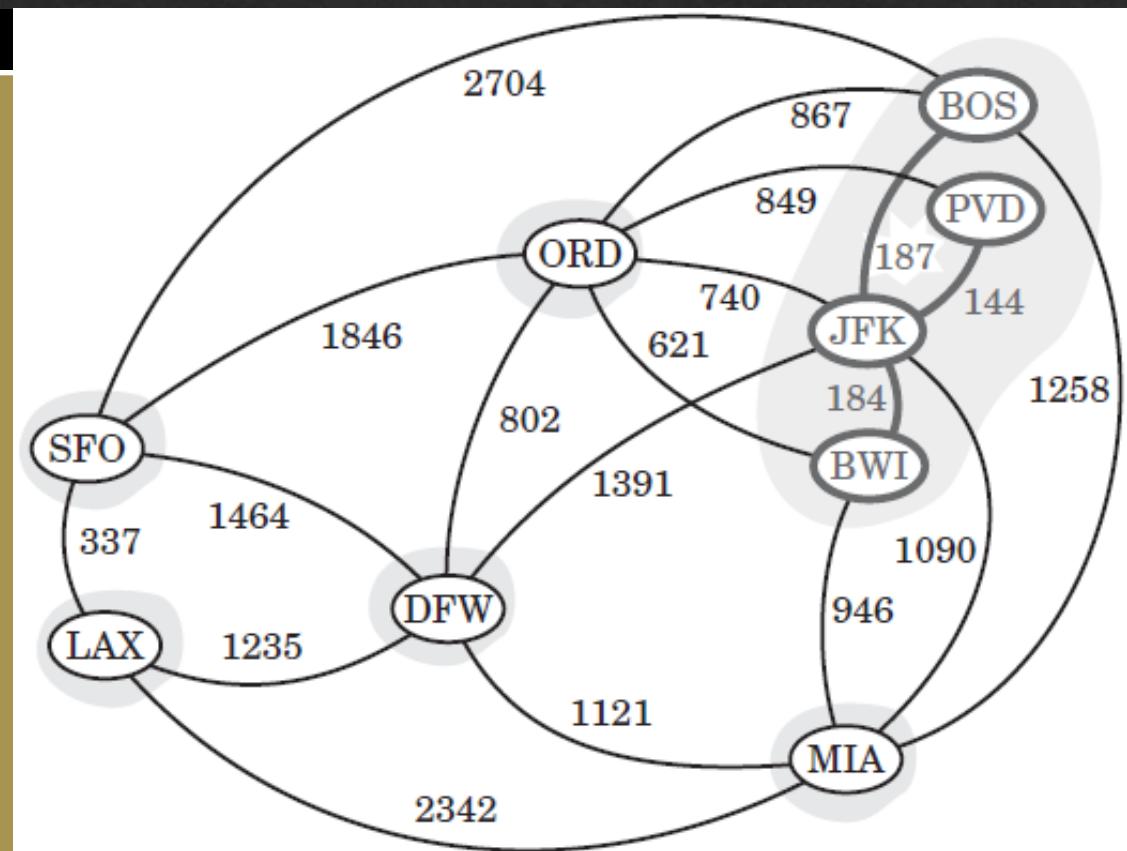
Kruskal for Minimum Spanning Tree

{SFO}, {LAX}, {DFW}, {ORD}, {PVD, JFK, BWI, BOS}, {MIA}

❖ Priority Q

Edge (Weights)

(JFK, PVD)	144
(BWI, JFK)	184
(BOS, JFK)	187
(LAX, SFO)	337
(BWI, ORD)	621
(JFK, ORD)	740
(DFW, ORD)	802
(ORD, PVD)	849
(BOS, ORD)	867
(BWI, MIA)	946
(JFK, MIA)	1090
(DFW, MIA)	1121
(DFW, LAX)	1235
(BOS, MIA)	1258
(DFW, SFO)	1464
(LAX, MIA)	2342
(BOS, SFO)	2704



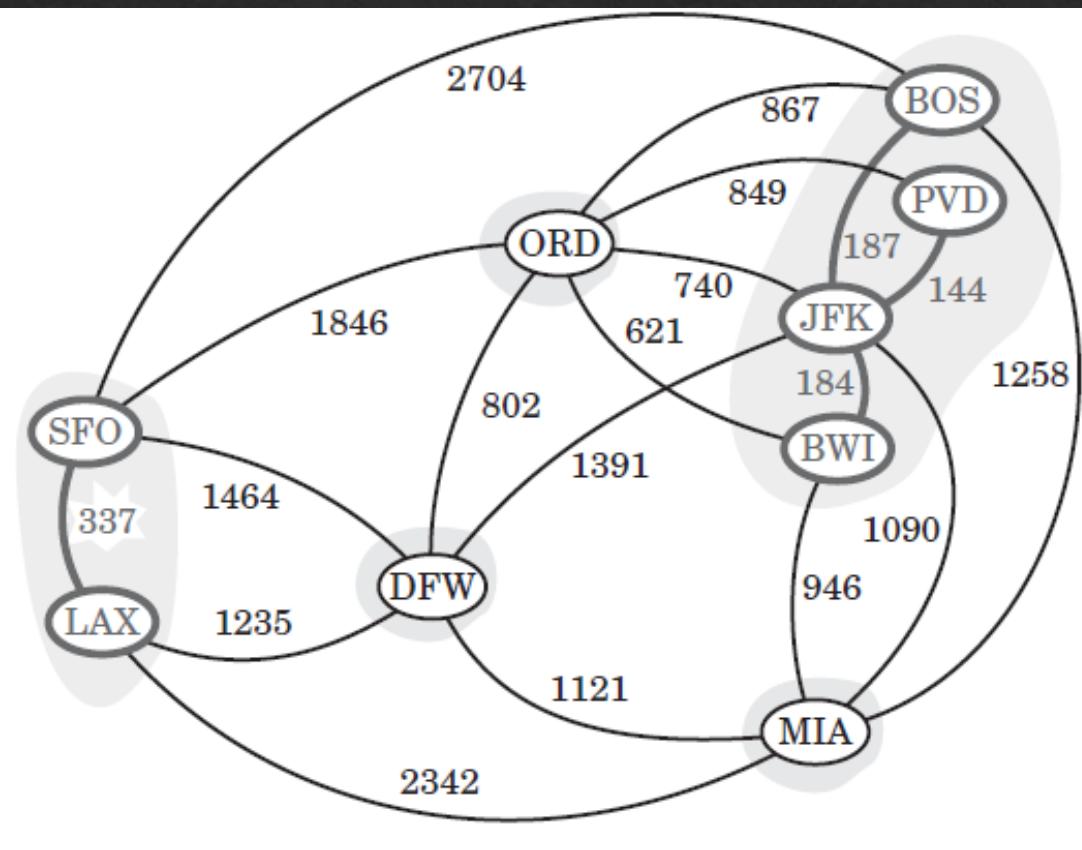
$$\begin{aligned} T[0] &= (\text{JFK, PVD}) \\ T[1] &= (\text{BWI, JFK}) \\ T[2] &= (\text{BOS, JFK}) \end{aligned}$$

Kruskal for Minimum Spanning Tree

{SFO, LAX}, {DFW}, {ORD}, {PVD, JFK, BWI, BOS}, {MIA}

❖ Priority Q

Edge (Weights)
(JFK, PVD) 144
(BWI, JFK) 184
(BOS, JFK) 187
(LAX, SFO) 337
(BWI, ORD) 621
(JFK, ORD) 740
(DFW, ORD) 802
(ORD, PVD) 849
(BOS, ORD) 867
(BWI, MIA) 946
(JFK, MIA) 1090
(DFW, MIA) 1121
(DFW, LAX) 1235
(BOS, MIA) 1258
(DFW, SFO) 1464
(LAX, MIA) 2342
(BOS, SFO) 2704



$$\begin{aligned}T[0] &= (\text{JFK, PVD}) \\T[1] &= (\text{BWI, JFK}) \\T[2] &= (\text{BOS, JFK}) \\T[3] &= (\text{LAX, SFO})\end{aligned}$$

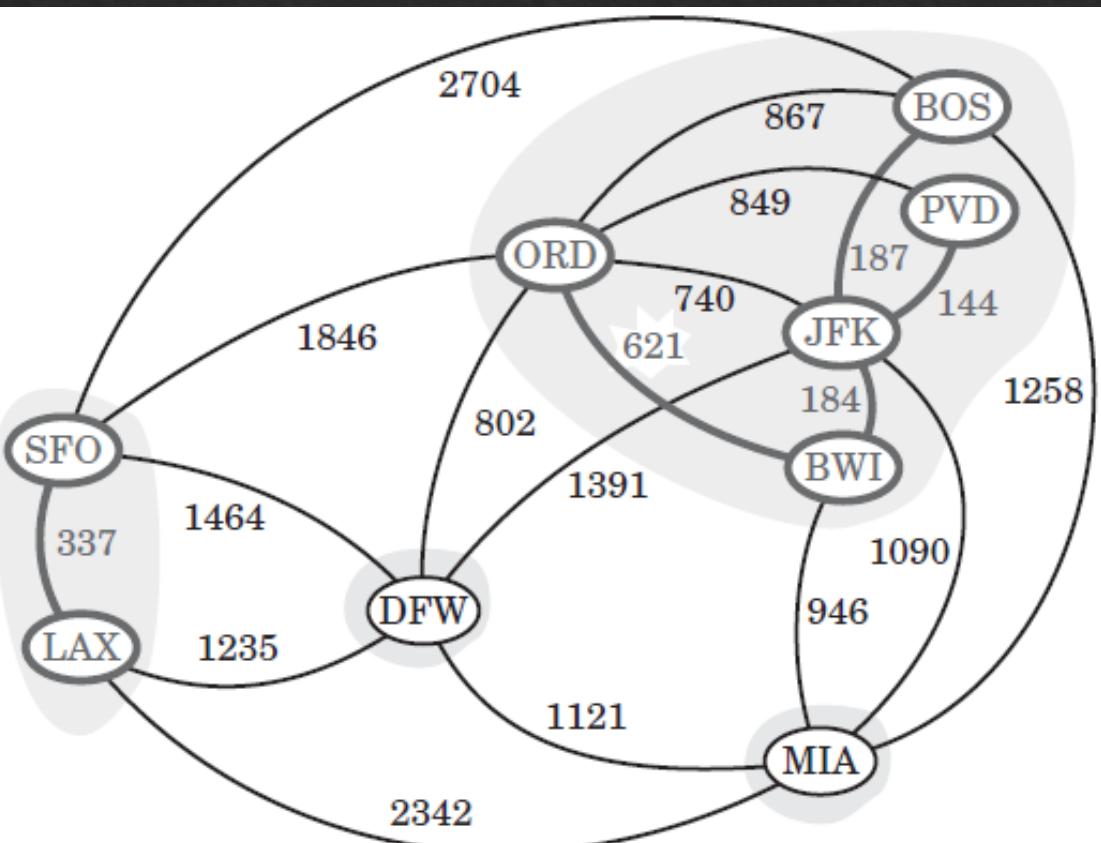
Kruskal for Minimum Spanning Tree

{SFO, LAX}, {DFW}, {ORD, PVD, JFK, BWI, BOS}, {MIA}

❖ Priority Q

Edge (Weights)

(JFK, PVD)	144
(BWI, JFK)	184
(BOS, JFK)	187
(LAX, SFO)	337
(BWI, ORD)	621
(JFK, ORD)	740
(DFW, ORD)	802
(ORD, PVD)	849
(BOS, ORD)	867
(BWI, MIA)	946
(JFK, MIA)	1090
(DFW, MIA)	1121
(DFW, LAX)	1235
(BOS, MIA)	1258
(DFW, SFO)	1464
(LAX, MIA)	2342
(BOS, SFO)	2704



$$\begin{aligned} T[0] &= (\text{JFK, PVD}) \\ T[1] &= (\text{BWI, JFK}) \\ T[2] &= (\text{BOS, JFK}) \\ T[3] &= (\text{LAX, SFO}) \\ T[4] &= (\text{BWI, ORD}) \end{aligned}$$

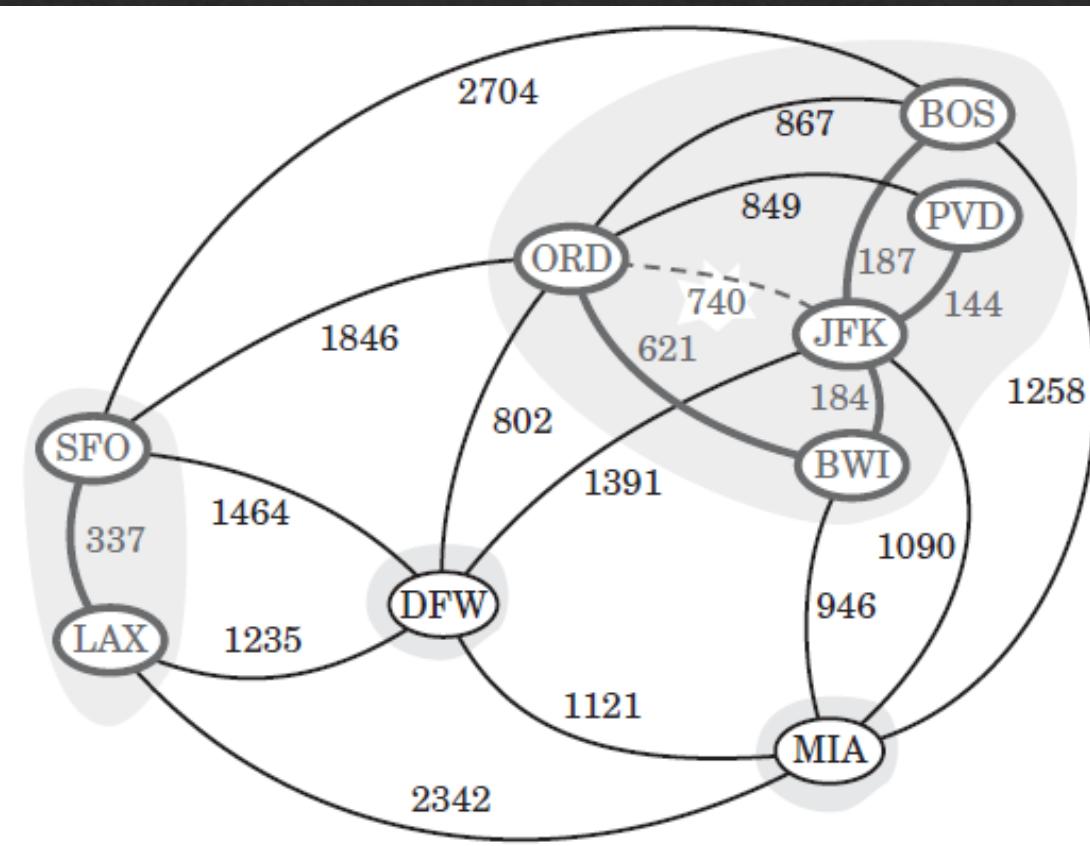
Kruskal for Minimum Spanning Tree

{SFO, LAX}, {DFW}, {ORD, PVD, JFK, BWI, BOS}, {MIA}

❖ Priority Q

Edge (Weights)

(JFK, PVD)	144
(BWI, JFK)	184
(BOS, JFK)	187
(LAX, SFO)	337
(BWI, ORD)	621
(JFK, ORD)	740
(DFW, ORD)	802
(ORD, PVD)	849
(BOS, ORD)	867
(BWI, MIA)	946
(JFK, MIA)	1090
(DFW, MIA)	1121
(DFW, LAX)	1235
(BOS, MIA)	1258
(DFW, SFO)	1464
(LAX, MIA)	2342
(BOS, SFO)	2704



$$\begin{aligned} T[0] &= (\text{JFK, PVD}) \\ T[1] &= (\text{BWI, JFK}) \\ T[2] &= (\text{BOS, JFK}) \\ T[3] &= (\text{LAX, SFO}) \\ T[4] &= (\text{BWI, ORD}) \end{aligned}$$

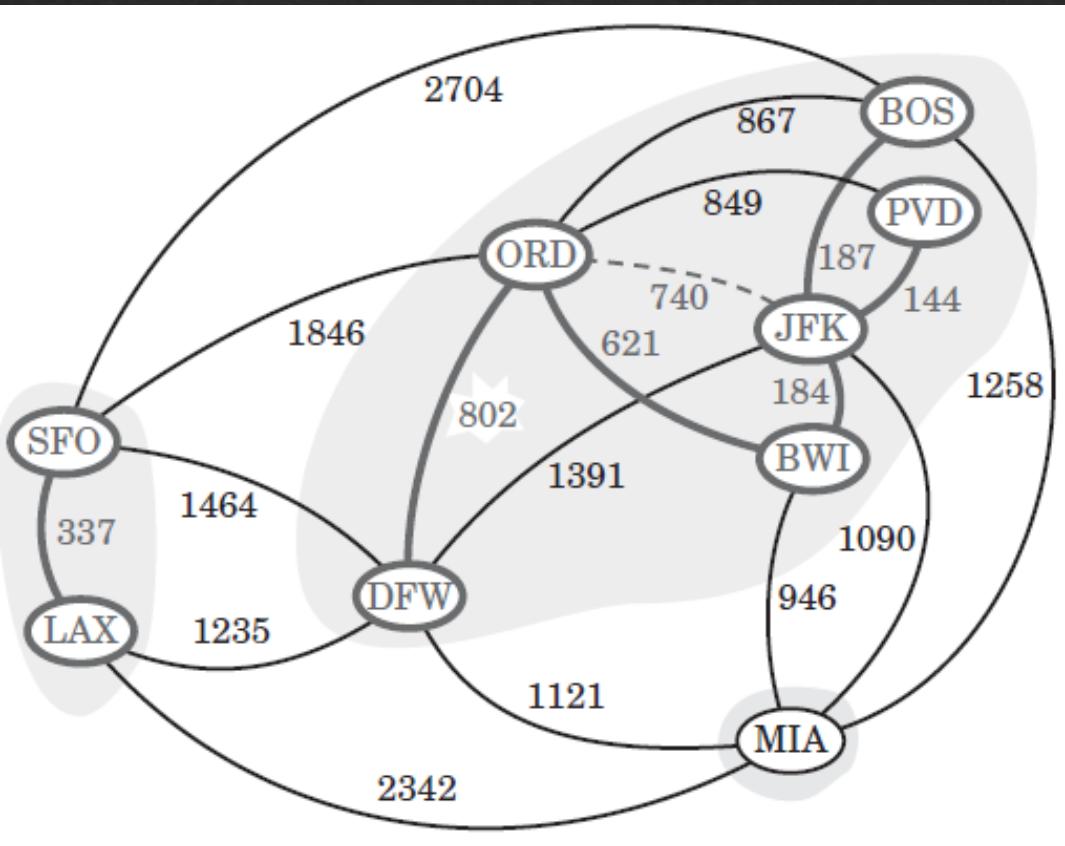
Kruskal for Minimum Spanning Tree

{SFO, LAX} {DFW, ORD, PVD, JFK, BWI, BOS} {MIA}

❖ Priority Q

Edge (Weights)

(JFK, PVD)	144
(BWI, JFK)	184
(BOS, JFK)	187
(LAX, SFO)	337
(BWI, ORD)	621
(JFK, ORD)	740
(DFW, ORD)	802
(ORD, PVD)	849
(BOS, ORD)	867
(BWI, MIA)	946
(JFK, MIA)	1090
(DFW, MIA)	1121
(DFW, LAX)	1235
(BOS, MIA)	1258
(DFW, SFO)	1464
(LAX, MIA)	2342
(BOS, SFO)	2704



$$\begin{aligned} T[0] &= (\text{JFK}, \text{PVD}) \\ T[1] &= (\text{BWI}, \text{JFK}) \\ T[2] &= (\text{BOS}, \text{JFK}) \\ T[3] &= (\text{LAX}, \text{SFO}) \\ T[4] &= (\text{BWI}, \text{ORD}) \\ T[5] &= (\text{DFW}, \text{ORD}) \end{aligned}$$

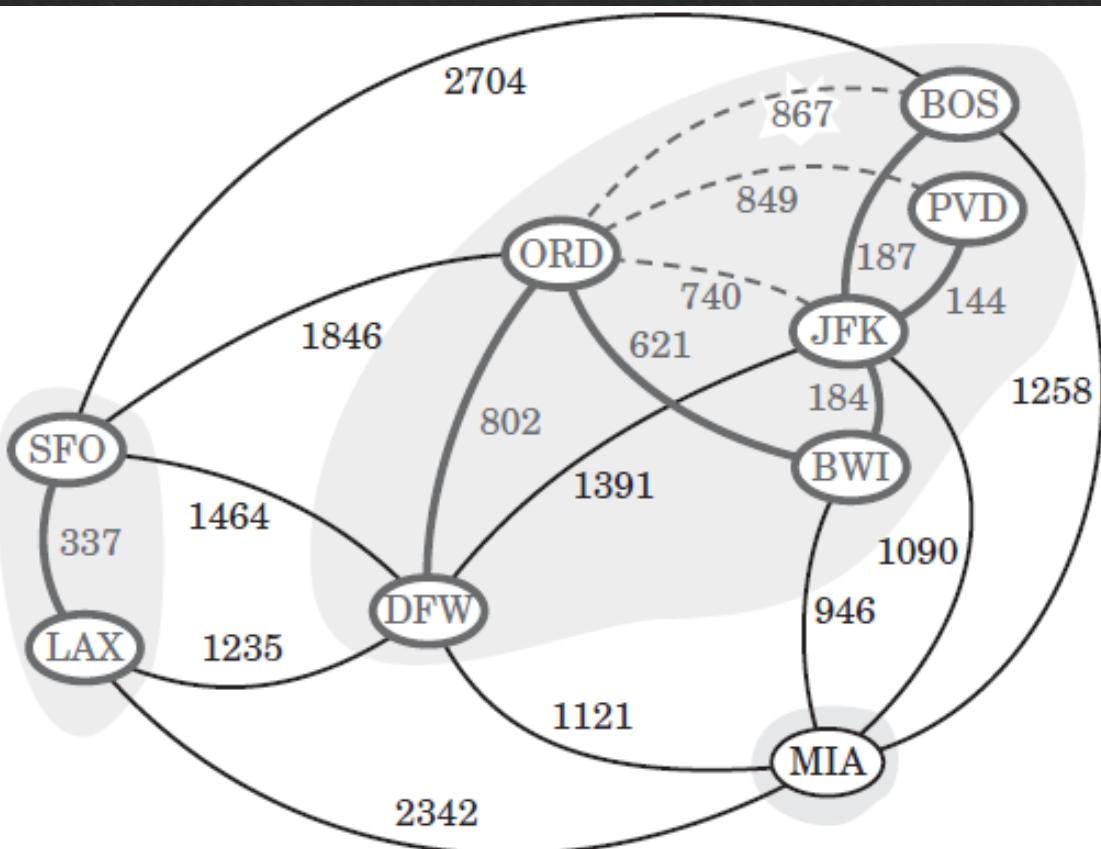
Kruskal for Minimum Spanning Tree

{SFO, LAX} {DFW, ORD, PVD, JFK, BWI, BOS} {MIA}

❖ Priority Q

Edge (Weights)

(JFK, PVD)	144
(BWI, JFK)	184
(BOS, JFK)	187
(LAX, SFO)	337
(BWI, ORD)	621
(JFK, ORD)	740
(DFW, ORD)	802
(ORD, PVD)	849
(BOS, ORD)	867
(BWI, MIA)	946
(JFK, MIA)	1090
(DFW, MIA)	1121
(DFW, LAX)	1235
(BOS, MIA)	1258
(DFW, SFO)	1464
(LAX, MIA)	2342
(BOS, SFO)	2704



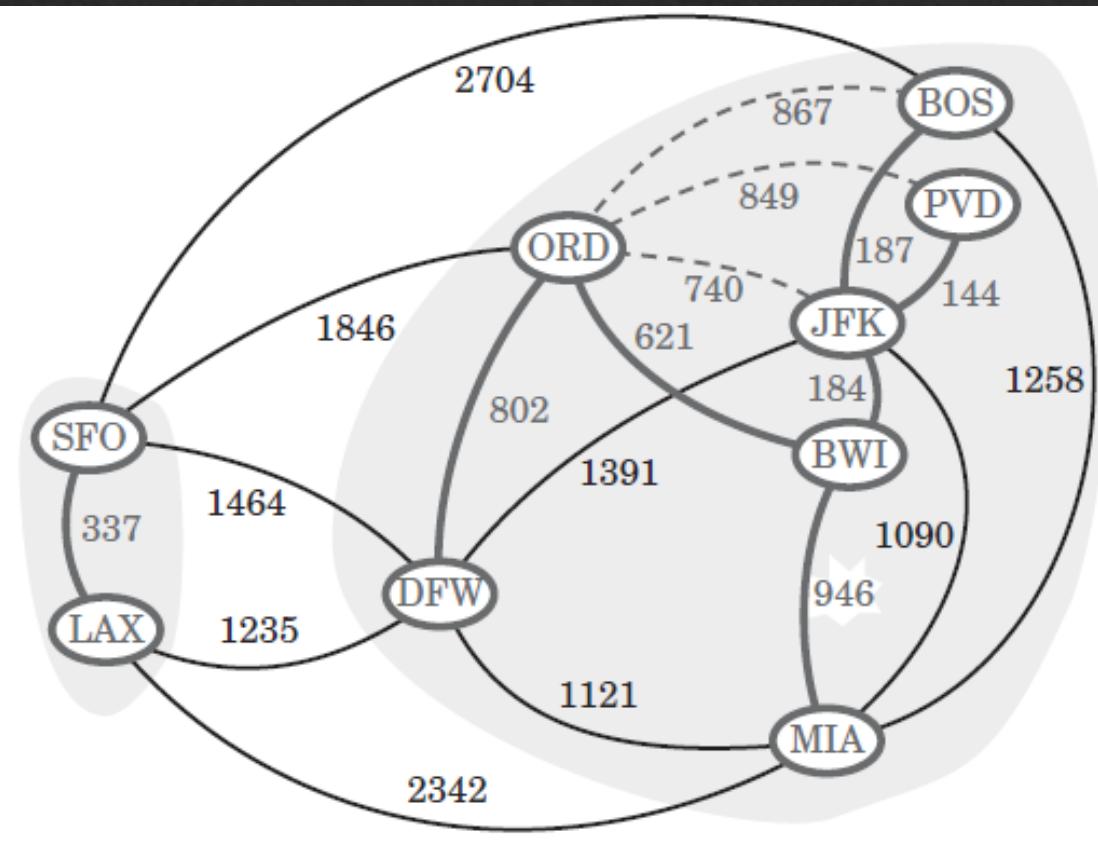
$$\begin{aligned} T[0] &= (\text{JFK}, \text{PVD}) \\ T[1] &= (\text{BWI}, \text{JFK}) \\ T[2] &= (\text{BOS}, \text{JFK}) \\ T[3] &= (\text{LAX}, \text{SFO}) \\ T[4] &= (\text{BWI}, \text{ORD}) \\ T[5] &= (\text{DFW}, \text{ORD}) \end{aligned}$$

Kruskal for Minimum Spanning Tree

{SFO, LAX} {DFW, ORD, PVD, JFK, BWI, BOS, MIA}

❖ Priority Q

Edge (Weights)
(JFK, PVD) 144
(BWI, JFK) 184
(BOS, JFK) 187
(LAX, SFO) 337
(BWI, ORD) 621
(JFK, ORD) 740
(DFW, ORD) 802
(ORD, PVD) 849
(BOS, ORD) 867
(BWI, MIA) 946
(JFK, MIA) 1090
(DFW, MIA) 1121
(DFW, LAX) 1235
(BOS, MIA) 1258
(DFW, SFO) 1464
(LAX, MIA) 2342
(BOS, SFO) 2704



$$\begin{cases} T[0] = (\text{JFK}, \text{PVD}) \\ T[1] = (\text{BWI}, \text{JFK}) \\ T[2] = (\text{BOS}, \text{JFK}) \\ T[3] = (\text{LAX}, \text{SFO}) \\ T[4] = (\text{BWI}, \text{ORD}) \\ T[5] = (\text{DFW}, \text{ORD}) \\ T[6] = (\text{BWI}, \text{MIA}) \end{cases}$$

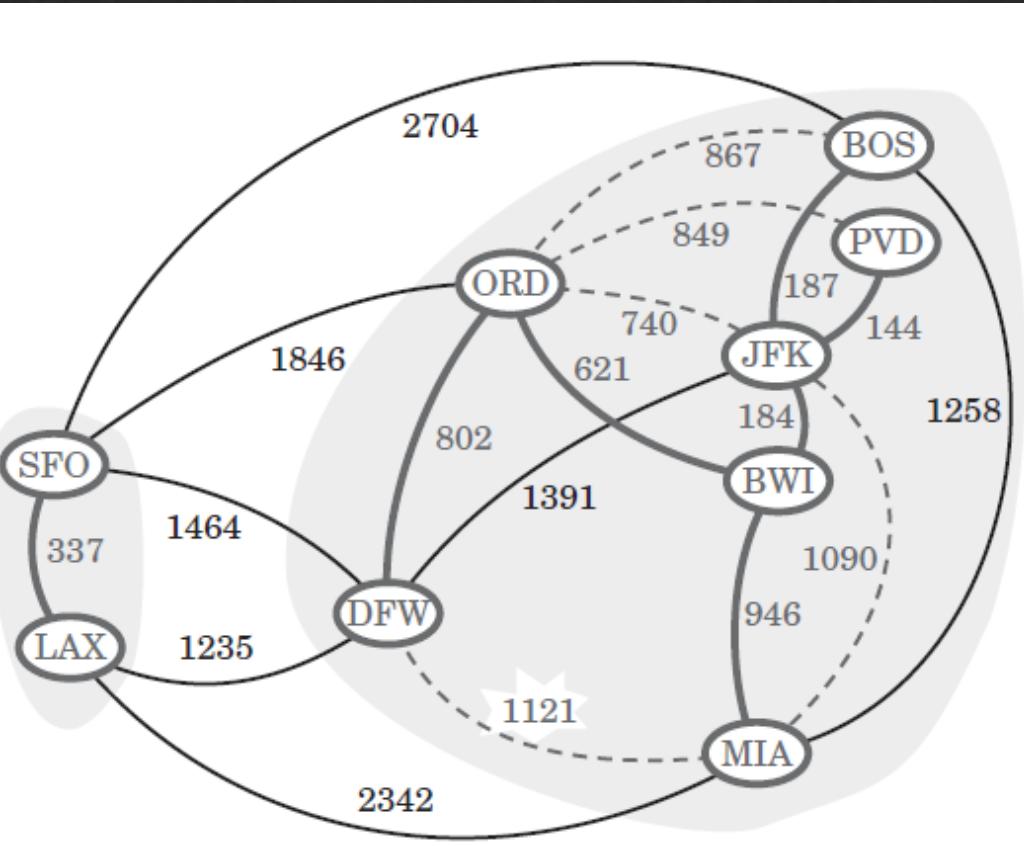
Kruskal for Minimum Spanning Tree

{SFO, LAX} {DFW, ORD, PVD, JFK, BWI, BOS, MIA}

❖ Priority Q

Edge (Weights)

(JFK, PVD)	144
(BWI, JFK)	184
(BOS, JFK)	187
(LAX, SFO)	337
(BWI, ORD)	621
(JFK, ORD)	740
(DFW, ORD)	802
(ORD, PVD)	849
(BOS, ORD)	867
(BWI, MIA)	946
(JFK, MIA)	1090
(DFW, MIA)	1121
(DFW, LAX)	1235
(BOS, MIA)	1258
(DFW, SFO)	1464
(LAX, MIA)	2342
(BOS, SFO)	2704



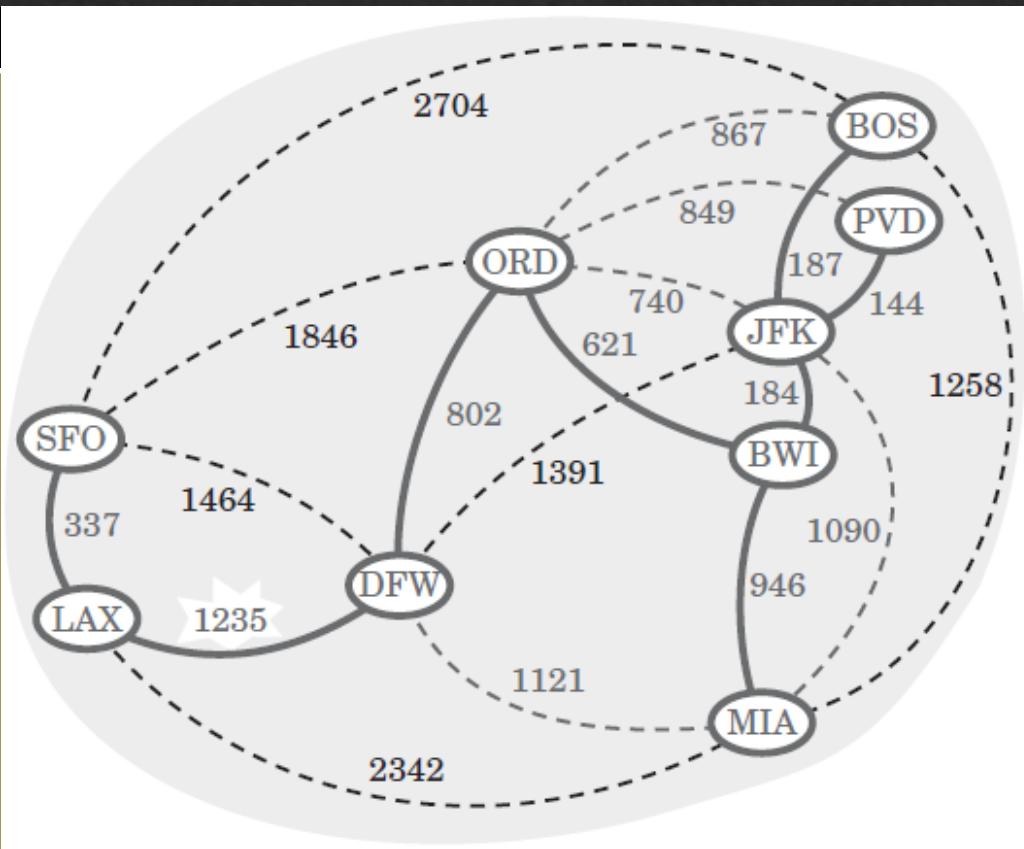
$$\begin{cases} T[0] = (\text{JFK}, \text{PVD}) \\ T[1] = (\text{BWI}, \text{JFK}) \\ T[2] = (\text{BOS}, \text{JFK}) \\ T[3] = (\text{LAX}, \text{SFO}) \\ T[4] = (\text{BWI}, \text{ORD}) \\ T[5] = (\text{DFW}, \text{ORD}) \\ T[6] = (\text{BWI}, \text{MIA}) \end{cases}$$

Kruskal for Minimum Spanning Tree

{SFO, LAX, DFW, ORD, PVD, JFK, BWI, BOS, MIA}

❖ Priority Q

Edge (Weights)
(JFK, PVD) 144
(BWI, JFK) 184
(BOS, JFK) 187
(LAX, SFO) 337
(BWI, ORD) 621
(JFK, ORD) 740
(DFW, ORD) 802
(ORD, PVD) 849
(BOS, ORD) 867
(BWI, MIA) 946
(JFK, MIA) 1090
(DFW, MIA) 1121
(DFW, LAX) 1235
(BOS, MIA) 1258
(DFW, SFO) 1464
(LAX, MIA) 1342
(BOS, SFO) 2704



$T[0]$	= (JFK, PVD)
$T[1]$	= (BWI, JFK)
$T[2]$	= (BOS, JFK)
$T[3]$	= (LAX, SFO)
$T[4]$	= (BWI, ORD)
$T[5]$	= (DFW, ORD)
$T[6]$	= (BWI, MIA)
$T[7]$	= (DFW, LAX)

Analyzing Kruskal's Algorithm With Sample Implementation

Heap
Initialize in $O(m \log m)$

Algorithm KruskalMST(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

 Define an elementary cluster $C(v) \leftarrow \{v\}$.

 Let Q be a priority queue storing the edges in G , using edge weights as keys

$T \leftarrow \emptyset$ // T will ultimately contain the edges of the MST

while T has fewer than $n - 1$ edges **do**

$(u, v) \leftarrow Q.\text{removeMin}()$

 Let $C(v)$ be the cluster containing v

 Let $C(u)$ be the cluster containing u

if $C(v) \neq C(u)$ **then**

 Add edge (v, u) to T

 Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$

return tree T

Analyzing Kruskal's Algorithm

Algorithm KruskalMST(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

 Define an elementary cluster $C(v) \leftarrow \{v\}$.

Let Q be a priority queue storing the edges in G , using edge weights as keys

$T \leftarrow \emptyset$ // T will ultimately contain the edges of the MST

while T has fewer than $n - 1$ edges **do**

$(u, v) \leftarrow Q.\text{removeMin}()$

 Let $C(v)$ be the cluster containing v

 Let $C(u)$ be the cluster containing u

if $C(v) \neq C(u)$ **then**

 Add edge (v, u) to T

 Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$

return tree T

$O(\log m)$

G is simple ~
 $O(\log n)$

Simple: No loop,
multiple edges

Analyzing Kruskal's Algorithm

Algorithm KruskalMST(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

 Define an elementary cluster $C(v) \leftarrow \{v\}$.

Let Q be a priority queue storing the edges in G , using edge weights as keys

$T \leftarrow \emptyset$ // T will ultimately contain the edges of the MST

while T has fewer than $n - 1$ edges **do**

$(u, v) \leftarrow Q.\text{removeMin}()$

 Let $C(v)$ be the cluster containing v

 Let $C(u)$ be the cluster containing u

if $C(v) \neq C(u)$ **then**

 Add edge (v, u) to T

 Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$

return tree T

$O(l)$

unordered linked
list

Analyzing Kruskal's Algorithm

Algorithm KruskalMST(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

 Define an elementary cluster $C(v) \leftarrow \{v\}$.

Let Q be a priority queue storing the edges in G , using edge weights as keys

$T \leftarrow \emptyset$ // T will ultimately contain the edges of the MST

while T has fewer than $n - 1$ edges **do**

$(u, v) \leftarrow Q.\text{removeMin}()$

 Let $C(v)$ be the cluster containing v

 Let $C(u)$ be the cluster containing u

if $C(v) \neq C(u)$ **then**

 Add edge (v, u) to T

return tree T

$O(\min\{|C(u)|, |C(v)|\})$

unordered linked
list

Total time spent merging clusters is $O(n \log n)$

Each merge \rightarrow The size of the cluster got merged into at most doubled.

$t(v)$: Number of times vertex v is moved to a new cluster.

$$t(v) \leq \log n$$

Total time spent merging clusters

$$\sum_{v \in G} t(v) \leq n \log n$$

Kruskal's Running Time

$O((n+m)\log n)$ which can be simplified to $O(m\log n)$ since G is simple & connected

Maximum Spanning Tree

Suppose you are a manager in the IT department for the government of a corrupt dictator, who has a collection of computers that need to be connected together to create a communication network for his spies. You are given a weighted graph, G , such that each vertex in G is one of these computers and each edge in G is a pair of computers that could be connected with a communication line. It is your job to decide how to connect the computers. Suppose now that the CIA has approached you and is willing to pay you various amounts of money for you to choose some of these edges to belong to this network (presumably so that they can spy on the dictator). Thus, for you, the weight of each edge in G is the amount of money, in U.S. dollars, that the CIA will pay you for using that edge in the communication network. Describe an efficient algorithm, therefore, for finding a maximum spanning tree in G , which would maximize the money you can get from the CIA for connecting the dictator's computers in a spanning tree.



Ring Algebra

19th century German = "Association"

Denoted by

$(R, +, \circ, 0_n, 1_n)$

In algebra, **ring theory** is the study of **rings** (algebraic structures) in which addition and multiplication are defined and have similar properties to those operations defined for the integers.

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

- I. Rings: Algebraic structures
 - I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying
 - I. R is an abelian group under addition
 - I. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

- I. Rings: Algebraic structures
 - I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying
 - I. R is an abelian group under addition
 - I. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)
 - 2. $a+b = b+a$ for all a,b in R (commutative)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

- I. Rings: Algebraic structures
 - I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying
 - I. R is an abelian group under addition
 - I. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)
 - 2. $a+b = b+a$ for all a,b in R (commutative)
 - 3. $a+0 = a$ for all a in R (0 is the additive identity)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

- I. Rings: Algebraic structures
 - I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying
 - I. R is an abelian group under addition
 - 1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)
 - 2. $a+b = b+a$ for all a,b in R (commutative)
 - 3. $a+0 = a$ for all a in R (0 is the additive identity)
 - 4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

- I. Rings: Algebraic structures
 - I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying
 - I. R is an abelian group under addition
 - 1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)
 - 2. $a+b = b+a$ for all a,b in R (commutative)
 - 3. $a+0 = a$ for all a in R (0 is the additive identity)
 - 4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)
 - 2. R is a monoid under multiplication
 - 1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

- I. Rings: Algebraic structures
 - I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying
 - I. R is an abelian group under addition
 1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)
 2. $a+b = b+a$ for all a,b in R (commutative)
 3. $a+0 = a$ for all a in R (0 is the additive identity)
 4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)
 - 2. R is a monoid under multiplication
 - 1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)
 - 2. An element 1 in R s.t. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

I. Rings: Algebraic structures

1. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying

1. R is an abelian group under addition

1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

2. $a+b = b+a$ for all a,b in R (commutative)

3. $a+0 = a$ for all a in R (0 is the additive identity)

4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

2. R is a monoid under multiplication

1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

2. An element 1 in R s.t. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

3. Multiplication is distributive w.r.t addition

1. $a \circ (b+c) = (a \circ b) + (a \circ c)$ (left distributivity)

2. $(b+c) \circ a = (b \circ a) + (c \circ a)$ (right distributivity)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

Associative is not required.

i. Rings: Algebraic structures

i. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying

i. R is an abelian group under addition

1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

2. $a+b = b+a$ for all a,b in R (commutative)

3. $a+0 = a$ for all a in R (0 is the additive identity)

4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

2. R is a monoid under multiplication

1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

2. An element 1 in R s.t. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

3. Multiplication is distributive w.r.t addition

1. $a \circ (b+c) = (a \circ b) + (a \circ c)$ (left distributivity)

2. $(b+c) \circ a = (b \circ a) + (c \circ a)$ (right distributivity)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

Associative is not required.

Multiplicative inverse is not required.

i. Rings: Algebraic structures

i. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying

i. R is an abelian group under addition

i. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

2. $a+b = b+a$ for all a,b in R (commutative)

3. $a+0 = a$ for all a in R (0 is the additive identity)

4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

2. R is a monoid under multiplication

i. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

2. An element 1 in R s.t. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

3. Multiplication is distributive w.r.t addition

i. $a \circ (b+c) = (a \circ b) + (a \circ c)$ (left distributivity)

2. $(b+c) \circ a = (b \circ a) + (c \circ a)$ (right distributivity)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

I. Rings: Algebraic structures

1. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying

1. R is an abelian group under addition

1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

2. $a+b = b+a$ for all a,b in R (commutative)

3. $a+0 = a$ for all a in R (0 is the additive identity)

4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

2. R is a monoid under multiplication

1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

2. An element 1 in R s.t. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

3. Multiplication is distributive w.r.t addition

1. $a \circ (b+c) = (a \circ b) + (a \circ c)$ (left distributivity)

2. $(b+c) \circ a = (b \circ a) + (c \circ a)$ (right distributivity)

Ring Algebra

Denoted by

$(R, +, \circ, 0_n, 1_n)$

I. Rings: Algebraic structures

1. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying

1. R is an abelian group under addition

1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

2. $a+b = b+a$ for all a,b in R (commutative)

3. $a+0 = a$ for all a in R (0 is the additive identity)

4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

2. R is a monoid under multiplication

1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

2. An element 1 in R s.t. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

3. Multiplication is distributive w.r.t addition

1. $a \circ (b+c) = (a \circ b) + (a \circ c)$ (left distributivity)

2. $(b+c) \circ a = (b \circ a) + (c \circ a)$ (right distributivity)

3. If $ab = ba \rightarrow$ commutative ring

Ring Algebra

I. Rings: Algebraic structures

I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying

1. R is an abelian group under addition

1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

2. $a+b = b+a$ for all a,b in R (commutative)

3. $a+0 = a$ for all a in R (0 is the additive identity)

4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

2. R is a monoid under multiplication

1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

2. An element 1 in R s.t. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

3. Multiplication is distributive w.r.t addition

1. $a \circ (b+c) = (a \circ b) + (a \circ c)$ (left distributivity)

2. $(b+c) \circ a = (b \circ a) + (c \circ a)$ (right distributivity)

3. If $ab = ba \rightarrow$ commutative ring

Called the "Unity" and is unique



Denoted by $(R, +, \circ, 0, 1)$

Ring Algebra

Denoted by

$(R, +, \circ, 0, 1)$

Distributive laws, are the only axioms
that connect $+$ and \circ .

I. Rings: Algebraic structures

I. Definition: A set R equipped with 2 binary operations ($+$ and \circ) satisfying

I. R is an abelian group under addition

1. $(a+b)+c = a+(b+c)$ for all a,b,c in R (associative)

2. $a+b = b+a$ for all a,b in R (commutative)

3. $a+0 = a$ for all a in R (0 is the additive identity)

4. $a+(-a) = 0$ ($-a$ is the additive inverse of a)

2. R is a monoid under multiplication

1. $(a \circ b) \circ c = a \circ (b \circ c)$ for all a,b,c in R (associative)

2. An element 1 in R st. $a \circ 1 = a$ and $1 \circ a = a$ (multiplicative identity)

3. Multiplication is distributive w.r.t addition

1. $a \circ (b+c) = (a \circ b) + (a \circ c)$ (left distributivity)

2. $(b+c) \circ a = (b \circ a) + (c \circ a)$ (right distributivity)

3. If $ab = ba \rightarrow$ commutative ring

Examples

- $(C, +, \times, 0, 1)$: C is set of complex numbers, $+$ and \times are complex number addition and multiplication (commutative ring). Same for Z, R, Q
- $(\{0, 1, 2, \dots, m-1\}, +_m, *_m, 0, 1)$ where $+_m$ and $*_m$ are addition and multiplication modulo m (commutative ring)
- $(M_n, +, *, 0_n, I_n)$ where M_n is the set of all $n \times n$ matrices with elements from a ring, $+$, $*$ are matrix addition and multiplication and 0_n is zero matrix and I_n is identity matrix
- The set $M_2(R)$ of all 2×2 matrices over R is a ring using matrix addition and multiplication.

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

Matrix Algebra

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

Abbreviated as $A = [a_{ij}]$

Matrix Algebra

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \quad \text{Abbreviated as} \quad A = [a_{ij}]$$

R denotes a commutative ring.

$M_{mn}(R)$: Set of all $m \times n$ matrices with entries from R .

$A + B + C$ in $M_{mn}(R)$:

$$A+B = B+A \text{ and } A + (B+C) = (A+B) + C$$

Matrix Algebra

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \quad \text{Abbreviated as} \quad A = [a_{ij}]$$

R denotes a commutative ring.

$M_{mn}(R)$: Set of all $m \times n$ matrices with entries from R .

$A + B + C$ in $M_{mn}(R)$:

$$A+B = B+A \text{ and } A + (B+C) = (A+B) + C$$

Matrix Algebra

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \quad \text{Abbreviated as} \quad A = [a_{ij}]$$

Zero matrix of size $m \times n$: 0 or 0_{mn}

For all A in $M_{mn}(R)$

$$A + (-A) = 0$$

$$A + 0 = A$$

Matrix Algebra

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

Abbreviated as $A = [a_{ij}]$

$$A + B = [a_{ij} + b_{ij}]$$

$$A - B = A + (-B) = [a_{ij} - b_{ij}]$$

-> The additive arithmetic in $M_{mn}(R)$ is entirely analogous to numerical arithmetic

Example

If A, B in $M_{2,3}(R)$ find X in $M_{2,3}(R)$ such that $X + A = B$

$$A = \begin{bmatrix} 1 & -1 & 0 \\ 5 & 7 & -2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 7 & -1 \\ 0 & 1 & 6 \end{bmatrix}$$

Example

If A, B in $M_{2,3}(R)$ find X in $M_{2,3}(R)$ such that $X + A = B$

$$A = \begin{bmatrix} 1 & -1 & 0 \\ 5 & 7 & -2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 7 & -1 \\ 0 & 1 & 6 \end{bmatrix}$$

$$X = B - A = \begin{bmatrix} 3 & 7 & -1 \\ 0 & 1 & 6 \end{bmatrix} - \begin{bmatrix} 1 & -1 & 0 \\ 5 & 7 & -2 \end{bmatrix} = \begin{bmatrix} 2 & 8 & -1 \\ -5 & -6 & 8 \end{bmatrix}$$

Matrix Multiplication

The dot product of a row matrix and a column matrix:

$$V = [v_1 \ v_2 \ \cdots \ v_k], W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{bmatrix}$$
$$VW = [v_1 \ v_2 \ \cdots \ v_k] \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{bmatrix} = v_1w_1 + v_2w_2 + \cdots + v_kw_k$$

Matrix Multiplication

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,k} \end{bmatrix}$$

(i,j) entry of the product of these two matrices: go across the i th row of A and down j th column of B .

Matrix Multiplication

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,k} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,k} \end{bmatrix}$$

(i,j) entry of the product of these two matrices: go across the i th row of A and down j th column of B .

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Matrix Multiplication Example

$$\begin{bmatrix} 3 & -1 & 2 \\ 0 & 1 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 1 \\ 0 & 2 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 6+0-2 & 3-2+0 \\ 0+0-4 & 0+2+0 \end{bmatrix} = \begin{bmatrix} 4 & 1 \\ -4 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 \\ 0 & 2 \\ -1 & 0 \end{bmatrix} \times \begin{bmatrix} 3 & -1 & 2 \\ 0 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 6+0 & -2+1 & 4+4 \\ 0+0 & 0+2 & 0+8 \\ -3+0 & 1+0 & -2+0 \end{bmatrix} = \begin{bmatrix} 6 & -1 & 8 \\ 0 & 2 & 8 \\ -3 & 1 & -2 \end{bmatrix}$$

(i,j) entry of the product of these two matrices: go across the i th row of A and down j th column of B .

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Identity Matrix (\mathbf{I})

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \times \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}$$

Square Matrices & Inverses

$M_n(R) = M_{nn}(R)$ for any $n \geq 2$

Let A, B, C be matrices in $M_n(R)$. Then

1. $A + B = B + A$
2. $(A+B) + C = A + (B+C)$
3. $A + O_n = A$
4. $A + (-A) = O$
5. $(AB)C = A(BC)$
6. $AI = A = IA$
7. $A(B+C) = AB + AC$ and $(B+C)A = BA + CA$
8. Set $M_n(R)$ over R is a ring using matrix addition and multiplication.

Matrix Multiplication

$O(n^3)$ time (measured in ring operation + and \times . Why?

Matrix Multiplication

$O(n^3)$ time (measured in ring operation + and \times . Why?

Each column in B

Multiply through n rows in A

Multiply & Sum (i,j) pair \rightarrow n multiplication & n-1 summations

$\rightarrow n^2$

n column in B

$\rightarrow n^3$

Partitioned Matrices

$$C = \begin{bmatrix} 1 & -2 & 4 & 1 & 3 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 3 & 2 & -1 & 2 \\ 4 & 6 & 2 & 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \left[\begin{array}{ccc|cc} 1 & -2 & 4 & 1 & 3 \\ 2 & 1 & 1 & 1 & 1 \\ \hline 3 & 3 & 2 & -1 & 2 \\ 4 & 6 & 2 & 2 & 4 \end{array} \right]$$

$$B = \begin{bmatrix} -1 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 4 & 1 \end{bmatrix}$$

$$B = (\mathbf{b}_1 \ \mathbf{b}_2 \ \mathbf{b}_3) = \left[\begin{array}{c|cc} -1 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 4 & 1 \end{array} \right]$$

Block Multiplication

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,k} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n,1} & B_{n,2} & \cdots & B_{n,k} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,k} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ C_{m,1} & C_{m,2} & \cdots & C_{m,k} \end{bmatrix}$$

(i,j) entry of the product of these two matrices: go across the i th row of A and down j th column of B .

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

Block Multiplication

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ \hline 3 & 3 & 2 & 2 \end{array} \right]$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ \hline 3 & 1 & 1 & 1 \\ 3 & 2 & 1 & 2 \end{array} \right]$$

$$\left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ \hline 3 & 3 & 2 & 2 \end{array} \right] \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ \hline 3 & 1 & 1 & 1 \\ 3 & 2 & 1 & 2 \end{array} \right] = \left[\begin{array}{cc|cc} 8 & 6 & 4 & 5 \\ 10 & 9 & 6 & 7 \\ \hline 18 & 15 & 10 & 12 \end{array} \right]$$

Strassen's Algorithm for Matrix Multiplication

$$AB = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Instead of 8 multiplications and 4 additions as above

There are only 7 multiplications and 18 additions/subtractions

Strassen's Algorithm for Matrix Multiplication

$$AB = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Instead of 8 multiplications and 4 additions as above
There are only 7 multiplications and 18 additions/subtractions

$$m_1 = (a_{12} - a_{22})(b_{21} + b_{22})$$

$$m_2 = (a_{11} + a_{22})(b_{11} + b_{22})$$

$$m_3 = (a_{11} - a_{21})(b_{11} + b_{12})$$

$$m_4 = (a_{11} + a_{12})b_{22}$$

$$m_5 = a_{11}(b_{11} - b_{22})$$

$$m_6 = a_{22}(b_{21} - b_{11})$$

$$m_7 = (a_{21} + a_{22})b_{11}$$

$$C_{11} = m_1 + m_2 - m_4 + m_6$$

$$C_{12} = m_2 + m_5$$

$$C_{21} = m_6 + m_7$$

$$C_{12} = m_2 - m_3 + m_5 - m_7$$

Strassen's Algorithm for Matrix Multiplication

$$AB = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Main idea: Divide-and-conquer
 $n/2 \times n/2$ sub matrices

$$T(n) = 7T(n/2) + 18(n/2)^2 \quad n \geq 2$$
$$T(1) = 1$$

$T(n)$ is $O(n^{\log_2 7})$ ring operations.

Instead of 8 multiplications and 4 additions as above
There are only 7 multiplications and 18 additions/subtractions

$$\left\{ \begin{array}{l} m_1 = (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 = (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_3 = (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 = (a_{11} + a_{12})b_{22} \\ m_5 = a_{11}(b_{21} - b_{22}) \\ m_6 = a_{22}(b_{11} - b_{12}) \\ m_7 = (a_{21} + a_{22})b_{11} \\ C_{11} = m_1 + m_2 - m_4 + m_6 \\ C_{12} = m_2 + m_5 \\ C_{21} = m_6 + m_7 \\ C_{12} = m_2 - m_3 + m_5 - m_7 \end{array} \right.$$

Areas to Study Rings

Homomorphisms, Isomorphism, decomposition, subrings, prime.... (Abstract Algebra)

Introduction to Homomorphisms:

R: A set with structure of an additive abelian group & a multiplicative monoid, together with the distributive laws.

Structure-preserving mappings: $\theta: R \rightarrow S$ (S is another ring)

Areas to Study Rings

Homomorphisms:

R: A set with structure of an additive abelian group & a multiplicative monoid, together with the distributive laws.

Structure-preserving mappings: $\theta: R \rightarrow S$ (S is another ring)

Ring homomorphism if, for all r and r_i in R :

$$1. \theta(r+r_i) = \theta(r)+\theta(r_i) \quad (\text{Preserve addition})$$

$$2. \theta(rr_i) = \theta(r)\theta(r_i) \quad (\text{Preserve multiplication})$$

$$3. \theta(1_R) = 1_S \quad (\text{Preserves the unity})$$

Theorem. Ring homomorphism

θ preserves zero, negatives, \mathbb{Z} -multiplication, powers

Rational Expression

$r^2su^5 - 3su^{-2}r + 2$ is a rational expression.

$$\theta: R \rightarrow S$$

If $\theta(x) = \bar{x}$ for every $x \in R$ then

$$\theta(r^2su^5 - 3su^{-2}r + 2) = \bar{r}^2\bar{s}\bar{u}^5 - 3\bar{s}\bar{u}^{-2}\bar{r} + \bar{2}$$

Example

If $\theta: R \rightarrow S$ is a ring homomorphism, show that $\bar{\theta}: M_2(R) \rightarrow M_2(S)$ is also a ring homomorphism where

$$\bar{\theta} \begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} \theta(r) & \theta(s) \\ \theta(t) & \theta(u) \end{bmatrix} \quad \text{for all } \begin{bmatrix} r & s \\ t & u \end{bmatrix} \text{ in } M_2(R)$$

Example

If $\theta: R \rightarrow S$ is a ring homomorphism, show that $\bar{\theta}: M_2(R) \rightarrow M_2(S)$ is also a ring homomorphism where

$$\bar{\theta} \begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} \theta(r) & \theta(s) \\ \theta(t) & \theta(u) \end{bmatrix} \quad \text{for all } \begin{bmatrix} r & s \\ t & u \end{bmatrix} \text{ in } M_2(R)$$

You task: verifies the preservation of addition and unity.

The preservation of multiplication: We write $\theta(r) = \bar{r}, \forall r \in R$

$$\bar{\theta} \left(\begin{bmatrix} r & s \\ t & u \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \right) = \begin{bmatrix} \overline{ra+sc} & \overline{rb+sd} \\ \overline{ta+uc} & \overline{tb+ud} \end{bmatrix} = \begin{bmatrix} \overline{ra} + \overline{sc} & \overline{rb} + \overline{sd} \\ \overline{ta} + \overline{uc} & \overline{tb} + \overline{ud} \end{bmatrix} = \begin{bmatrix} \bar{r} & \bar{s} \\ \bar{t} & \bar{u} \end{bmatrix} \begin{bmatrix} \bar{a} & \bar{b} \\ \bar{c} & \bar{d} \end{bmatrix} = \bar{\theta} \left(\begin{bmatrix} r & s \\ t & u \end{bmatrix} \right) \cdot \bar{\theta} \left(\begin{bmatrix} a & b \\ c & d \end{bmatrix} \right)$$

Example

◇ Show that $x^3 - 5x^2 - x - 17 = 0$ has no solution in \mathbb{Z} .

Example

>Show that $x^3 - 5x^2 - x - 17 = 0$ has no solution in \mathbb{Z} .

Consider the homomorphism $\theta: \mathbb{Z} \rightarrow \mathbb{Z}_5$ given by $\theta(k) = \bar{k}$.

Suppose n in \mathbb{Z} is a solution. Apply θ gives: $\bar{n}^3 - 5\bar{n}^2 - \bar{n} - \bar{17} = \bar{0}$ in \mathbb{Z}_5

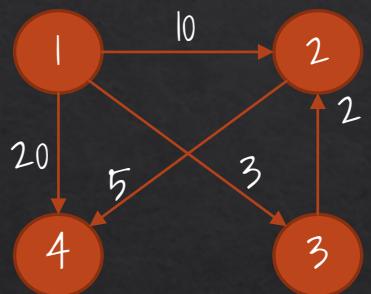
That is: $\bar{n}^3 - \bar{n} - \bar{2} = \bar{0} \text{ (I)}$

But \bar{n} can only take one of the value $\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}$ in \mathbb{Z}_5 . None of these satisfies (I). Hence, no solution of the original equation could exist in \mathbb{Z} .

Fermat's Last Theorem

Field	Number theory
Statement	For any integer $n > 2$, the equation $a^n + b^n = c^n$ has no positive integer solutions.
First stated by	Pierre de Fermat
First stated in	c. 1637
First proof by	Andrew Wiles
First proof in	Released 1994 Published 1995

Warshall-Floyd Algorithm – Find the Shortest Paths



Vertex	1	2	3	4
1	0	10	3	20
2	inf	0	inf	5
3	inf	2	0	inf
4	inf	inf	inf	0

What name should we give this algorithm?

```
function giveMeAName(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

What name should we give this algorithm?

```
function giveMeAName(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

Step k	a	b	t	Equation	Quotient and remainder
0	27	4	4	$27 = q_0 4 + r_0$	$q_0 = 6$ and $r_0 = 3$

What name should we give this algorithm?

```
function giveMeAName(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

Euclidean divisor (i.e. division with remainder)

Step k	a	b	t	Equation
0	27	4	4	$27 = q_0 4 + r_0$

Quotient and remainder

$$q_0 = 6 \text{ and } r_0 = 3$$

What name should we give this algorithm?

```
function giveMeAName(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

Step k	a	b	t	Equation	Quotient and remainder
0	27	4	4	$27 = q_0 4 + r_0$	$q_0 = 6$ and $r_0 = 3$
1	4	3	3	$4 = q_1 3 + r_1$	$q_1 = 1$ and $r_1 = 1$

What name should we give this algorithm?

```
function giveMeAName(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

Step k	a	b	t	Equation	Quotient and remainder
0	27	4	4	$27 = q_0 4 + r_0$	$q_0 = 6$ and $r_0 = 3$
1	4	3	3	$4 = q_1 3 + r_1$	$q_1 = 1$ and $r_1 = 1$
2	3	1	1	$3 = q_2 1 + r_2$	$q_2 = 3$ and $r_2 = 0$
3	1	0 (end)			

What name should we give this algorithm?

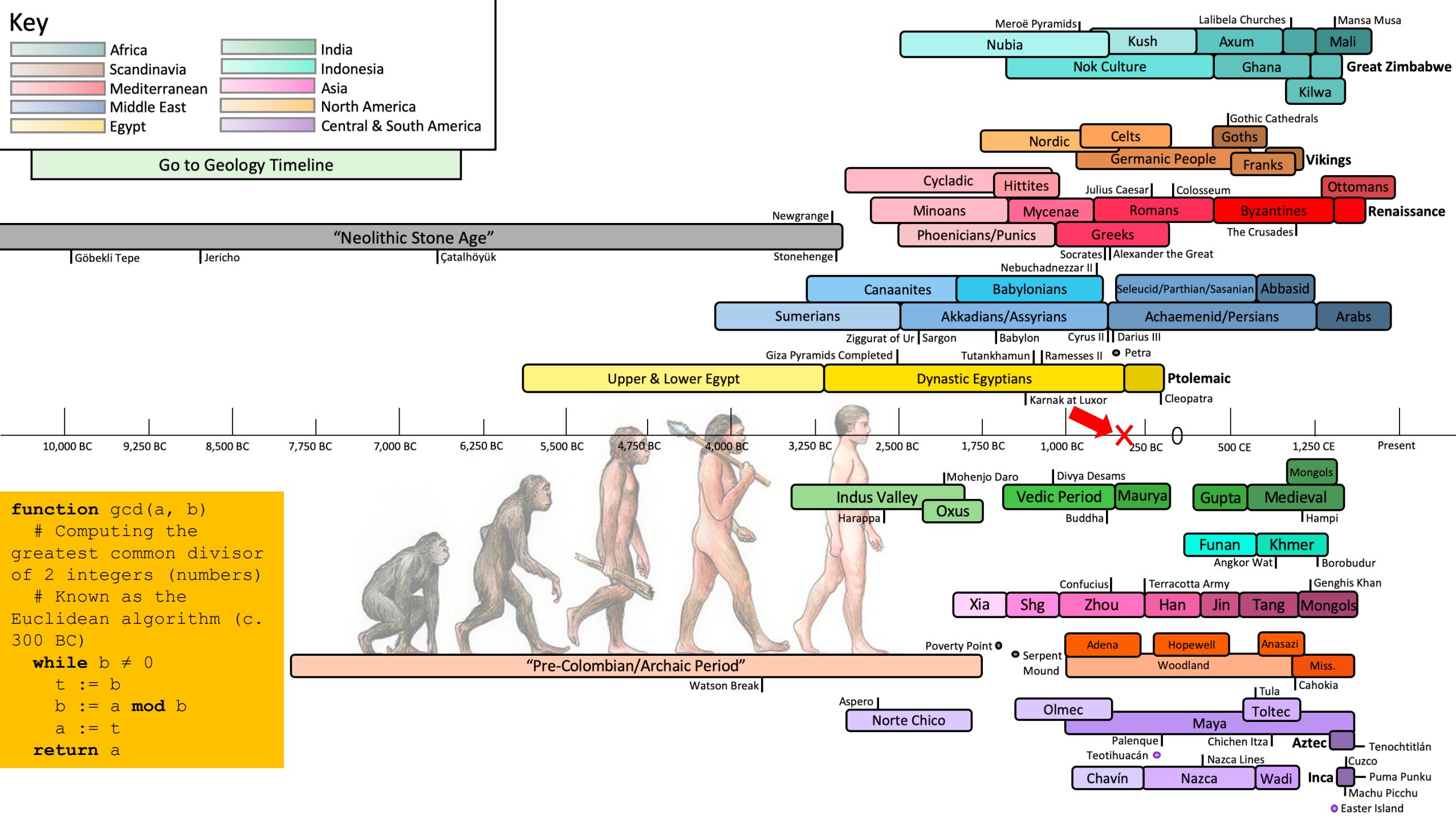
```
function giveMeAName(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

Step k	a	b	t	Equation	Quotient and remainder
0	27	18	18	$27 = q_0 \cdot 18 + r_0$	$q_0 = 1$ and $r_0 = 9$

What name should we give this algorithm?

```
function giveMeAName(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

Step k	a	b	t	Equation	Quotient and remainder
0	$27 = 3 \times 3 \times 3$	$18 = 3 \times 3 \times 2$	18	$27 \text{ mod } 18 = 9 \rightarrow 27 = q_0 18 + r_0$	$q_0 = 1$ and $r_0 = 9$
1	18	9	9	$18 = q_1 9 + r_1$	$q_1 = 2$ and $r_1 = 0$
2	9		0 (end)		



Euclidean Algorithm

(Ancient Greek Mathematician Euclid c. 300 B.C.)

$\gcd(a,b) = 1 \rightarrow a, b$ are said to be coprime (or relatively prime)

Chinese Remainder Theorem

There are certain things whose number is unknown.

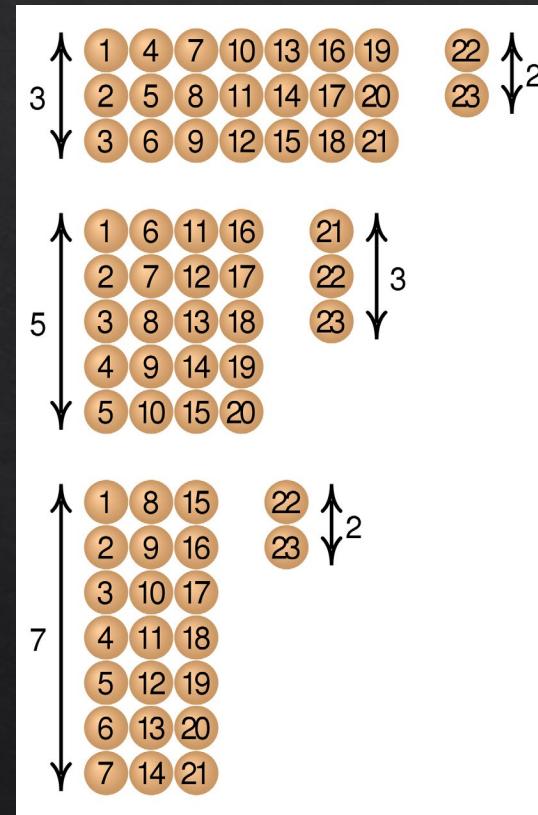
If we count them by threes, we have two left over;

by fives, we have three left over;

and by sevens, two are left over.

How many things are there?

Chinese Mathematician Sun-tzu 3rd century A.D.



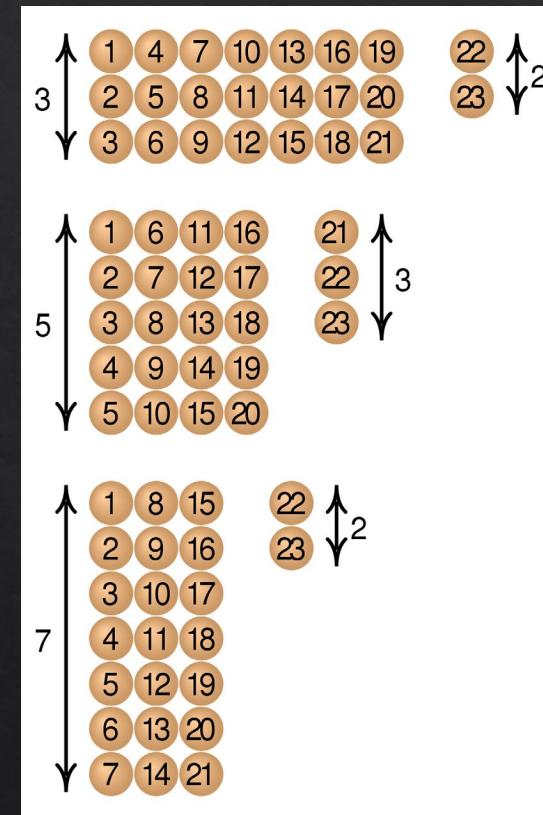
Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}$$

$$x \equiv 3 \pmod{5}$$

$$x \equiv 2 \pmod{7}$$

$$x = ?$$



Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}$$

$$x \equiv 3 \pmod{5}$$

$$x \equiv 2 \pmod{7}$$

$$x = ?$$

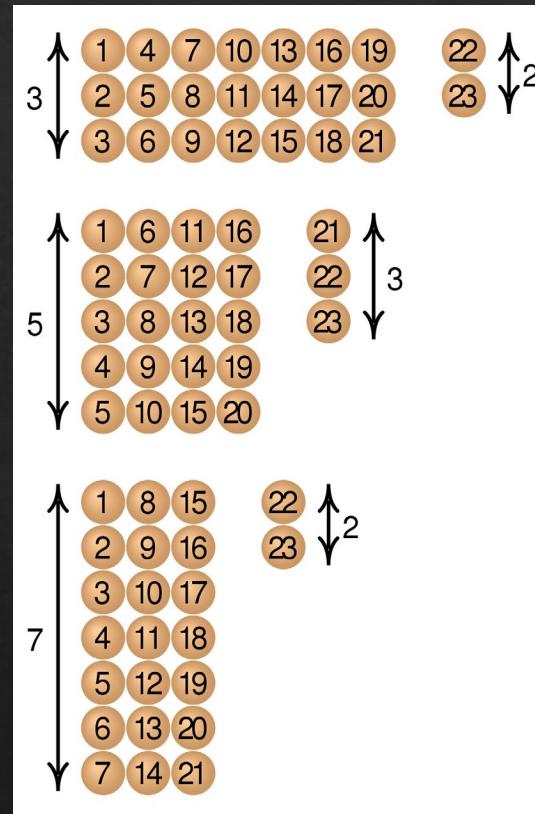
Suppose y is our solution ($x = y$)

Then,

$y + 3 \times 5 \times 7 = y + 105$ is also a solution.

→ So we only need to look for solutions mod 105.

By brute force, we find the solution $x = 23 \pmod{105}$



Chinese Remainder Theorem

- ◆ Theorem: Let p, q be coprime. Then the system of equations

$$x = a \pmod{p}$$

$$x = b \pmod{q}$$

Has a unique solution for x modulo pq

Chinese Remainder Theorem

◇ Theorem: Let p, q be coprime. Then the system of equations

$$x \equiv a \pmod{p}$$

$$x \equiv b \pmod{q}$$

Has a unique solution for x modulo pq

Proof: Let $p_1 = p^{-1} \pmod{q}$ and $q_1 = q^{-1} \pmod{p}$. These must exist because p, q are coprime.

Chinese Remainder Theorem

◇ Theorem: Let p, q be coprime. Then the system of equations

$$x \equiv a \pmod{p}$$

$$x \equiv b \pmod{q}$$

Has a unique solution for x modulo pq

Proof: Let $p_1 = p^{-1} \pmod{q}$ and $q_1 = q^{-1} \pmod{p}$. These must exist because p, q are coprime.

$$\text{E.g.: } p = 5, q = 7 \rightarrow p_1 = 5^{-1} \pmod{7} \rightarrow 5p_1 = 1 \pmod{7} \rightarrow p_1 = 3 \pmod{7}$$

$$\rightarrow q_1 = 7^{-1} \pmod{5} \rightarrow 7q_1 = 1 \pmod{5} \rightarrow 2q_1 = 1 \pmod{5} \rightarrow q_1 = 3 \pmod{5}$$

Chinese Remainder Theorem

◇ Theorem: Let p, q be coprime. Then the system of equations

$x = a \pmod{p}$, $x = b \pmod{q}$ has a unique solution for x modulo pq

Proof: Let $p_1 = p^{-1} \pmod{q}$ and $q_1 = q^{-1} \pmod{p}$. These must exist because p, q are coprime.

Then, we claim that if $y \in \mathbb{Z}$ s.t.

$$y = aqq_1 + bpp_1 \pmod{pq}$$

Then y satisfies both equations.

Chinese Remainder Theorem

◇ Theorem: Let p, q be coprime. Then the system of equations

$$x \equiv a \pmod{p}, x \equiv b \pmod{q} \text{ has a unique solution for } x \text{ modulo } pq$$

Proof: Let $p_1 = p^{-1} \pmod{q}$ and $q_1 = q^{-1} \pmod{p}$. These must exist because p, q are coprime.

Then, we claim that if $y \in \mathbb{Z}$ s.t.

$$y = aqq_1 + bpp_1 \pmod{pq}$$

Then y satisfies both equations.

E.g.: $p = 5, q = 7 \rightarrow p_1 = 3, q_1 = 3$

If $a = 2, b = 3$, we have: $y = (2 \times 7 \times 3 + 3 \times 5 \times 3) \pmod{35} = (42 + 45) \pmod{35} = 87 \pmod{35} = 17$

Chinese Remainder Theorem for Several Equations

Given : Let p_0, p_1, \dots, p_{k-1} be k relatively prime numbers. Denote residues $(u_0, u_1, \dots, u_{k-1})$ w.r.t to p_0, p_1, \dots, p_{k-1}

Then

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \text{ modulo } p$$

where

$$p = p_0 * p_1 * \dots * p_{k-1} \text{ and } c_i = p / p_i, \text{ and } d_i = c_i^{-1} \bmod p_i$$

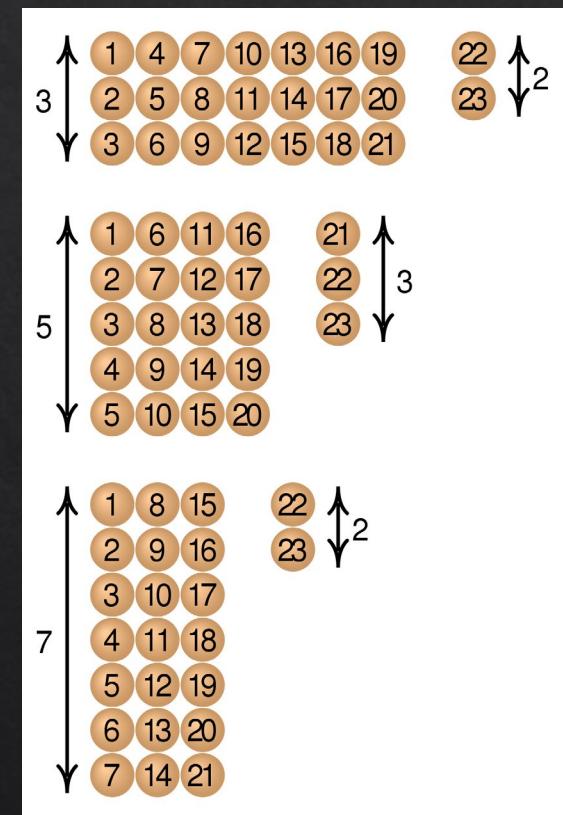
We write: $u \leftrightarrow (u_0, u_1, \dots, u_{k-1})$

Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

i	p_i	$c_i = p/p_i$	$d_i = c_i^{-1} \pmod{p_i}$	$c_i d_i u_i$
3	3	1	1	1
5	5	1	2	2

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{p}$$



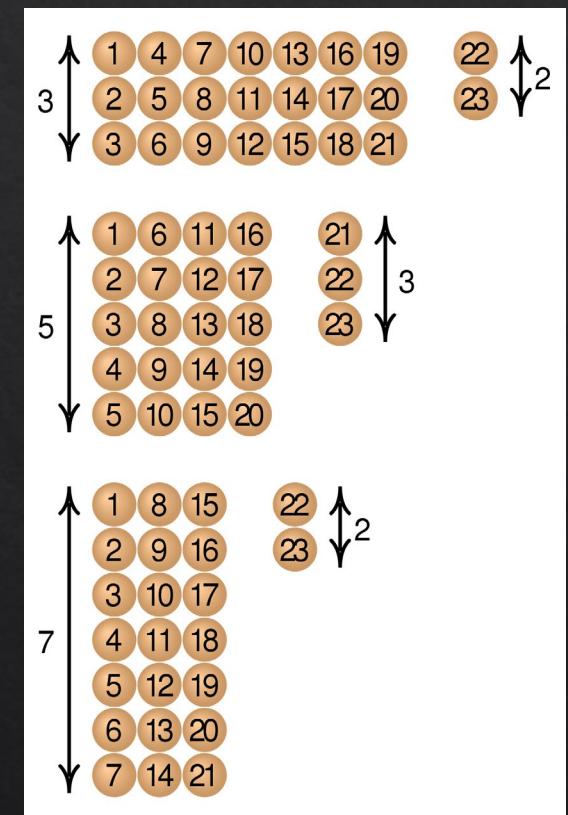
Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

$$P = P_0 P_1 P_2 = 3 \times 5 \times 7 = 105$$

i	0	1	2
0			
1			
2			

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \text{ modulo } p$$



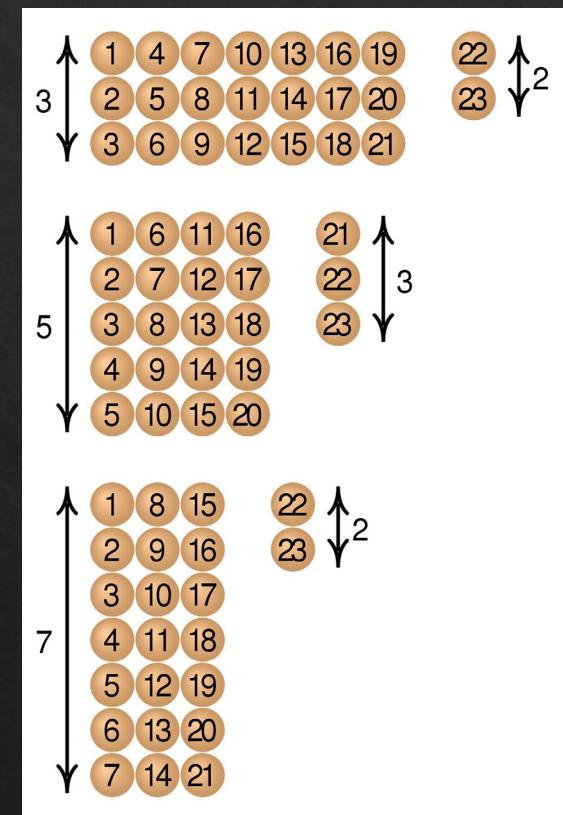
Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

$$P = P_0 P_1 P_2 = 3 \times 5 \times 7 = 105$$

i	P_i	$c_i = P/P_i$	$d_i = c_i^{-1} \pmod{P_i}$	$c_i d_i u_i$
0	3	35		
1	5	21		
2	7	15		

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{P}$$



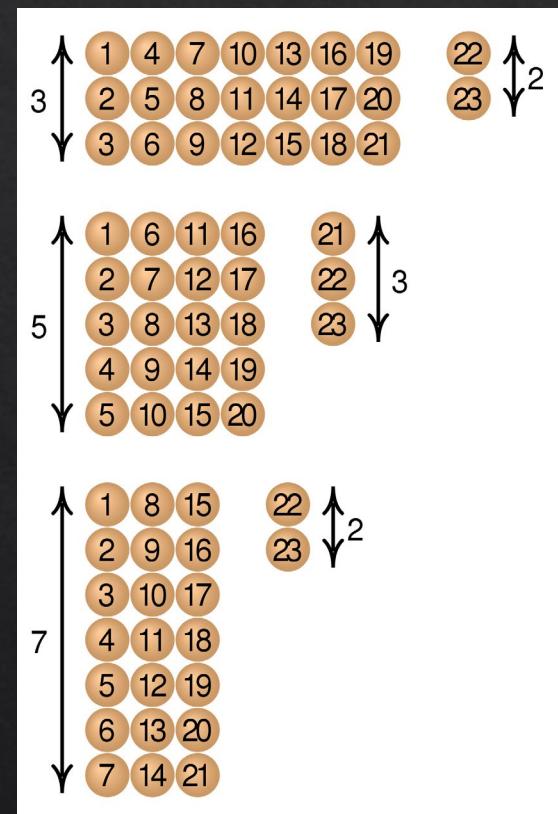
Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

$$P = P_0 P_1 P_2 = 3 \times 5 \times 7 = 105$$

i	P_i	$c_i = P/P_i$	$d_i = c_i^{-1} \pmod{P_i}$	$c_i d_i u_i$
0	3	35	$35d_0 \equiv 1 \pmod{3}$ $2d_0 \equiv 1 \pmod{3}$ $d_0 \equiv 2$	
1	5	21		
2	7	15		

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{P}$$



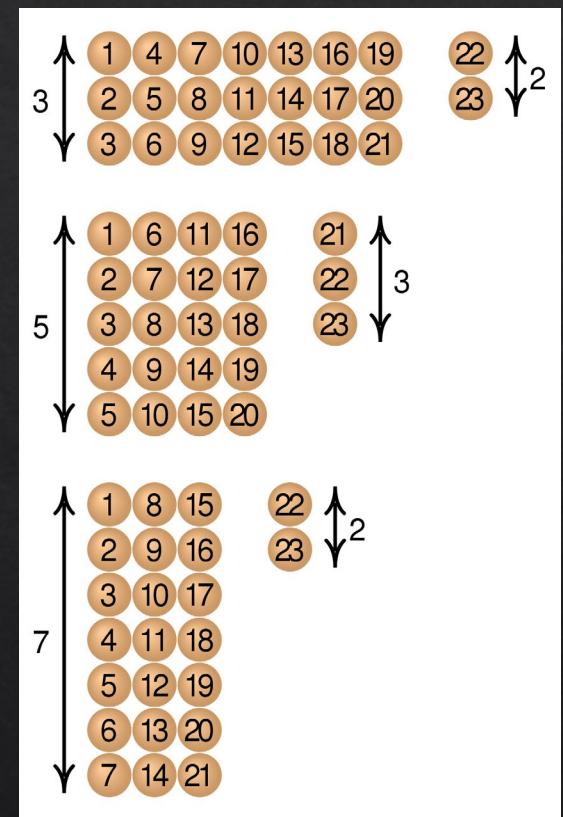
Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

$$P = P_0 P_1 P_2 = 3 \times 5 \times 7 = 105$$

i	P_i	$c_i = P/P_i$	$d_i = c_i^{-1} \pmod{P_i}$	$c_i d_i u_i$
0	3	35	$35d_0 \equiv 1 \pmod{3}$ $2d_0 \equiv 1 \pmod{3}$ $d_0 = 2$	
1	5	21	$21d_1 \equiv 1 \pmod{5}$ $d_1 = 1$	
2	7	15	$15d_2 \equiv 1 \pmod{7}$ $d_2 = 1$	

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{P}$$



Chinese Remainder Theorem

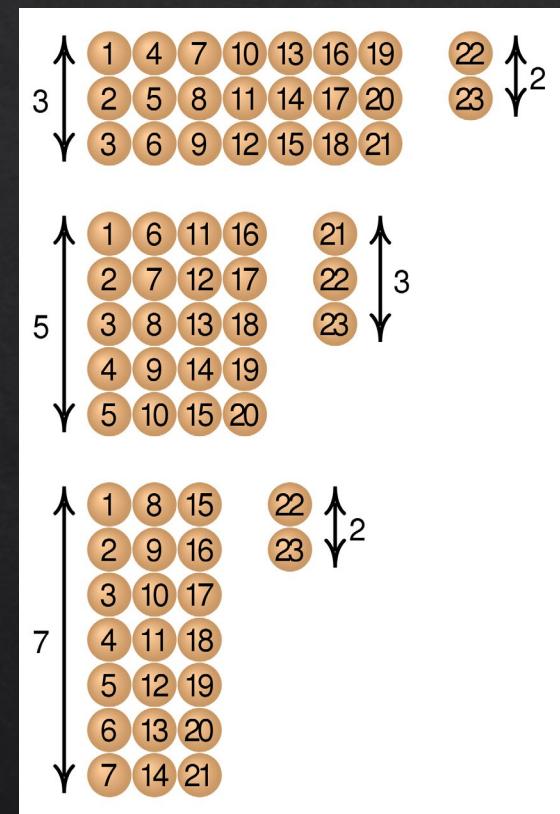
$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

$$P = P_0 P_1 P_2 = 3 \times 5 \times 7 = 105$$

i	P_i	$c_i = P/P_i$	$d_i = c_i^{-1} \pmod{P_i}$	$c_i d_i u_i$
0	3	35	$35d_0 \equiv 1 \pmod{3}$ $2d_0 \equiv 1 \pmod{3}$ $d_0 = 2$	$35 \times 2 \times 2 = 140$
1	5	21	$21d_1 \equiv 1 \pmod{5}$ $d_1 = 1$	
2	7	15	$15d_2 \equiv 1 \pmod{7}$ $d_2 = 1$	

$$140 + 63 + 30$$

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{P}$$



Chinese Remainder Theorem

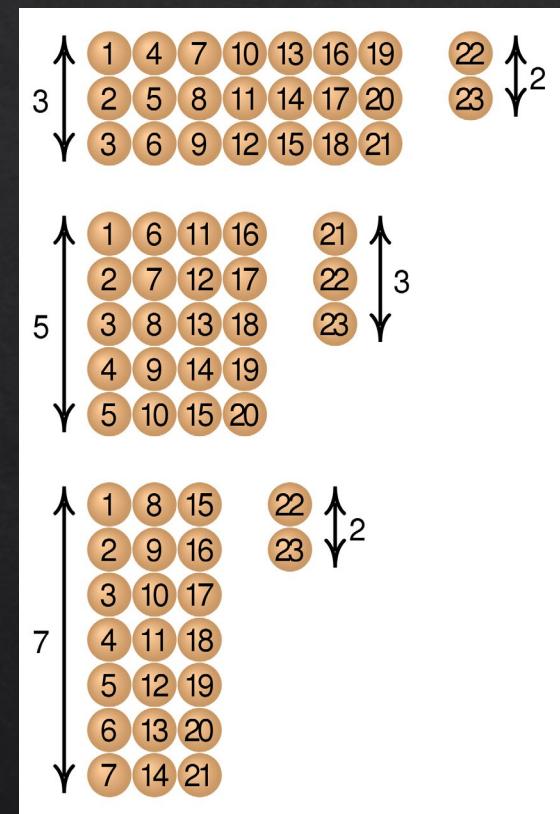
$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

$$P = P_0 P_1 P_2 = 3 \times 5 \times 7 = 105$$

i	P_i	$c_i = P/P_i$	$d_i = c_i^{-1} \pmod{P_i}$	$c_i d_i u_i$
0	3	35	$35d_0 \equiv 1 \pmod{3}$ $2d_0 \equiv 1 \pmod{3}$ $d_0 = 2$	$35 \times 2 \times 2 = 140$
1	5	21	$21d_1 \equiv 1 \pmod{5}$ $d_1 = 1$	$21 \times 1 \times 3 = 63$
2	7	15	$15d_2 \equiv 1 \pmod{7}$ $d_2 = 1$	$15 \times 1 \times 2 = 30$

$$140 + 63 + 30 = 233 \pmod{105} = 23$$

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{P}$$



Chinese Remainder Theorem

$$x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$$

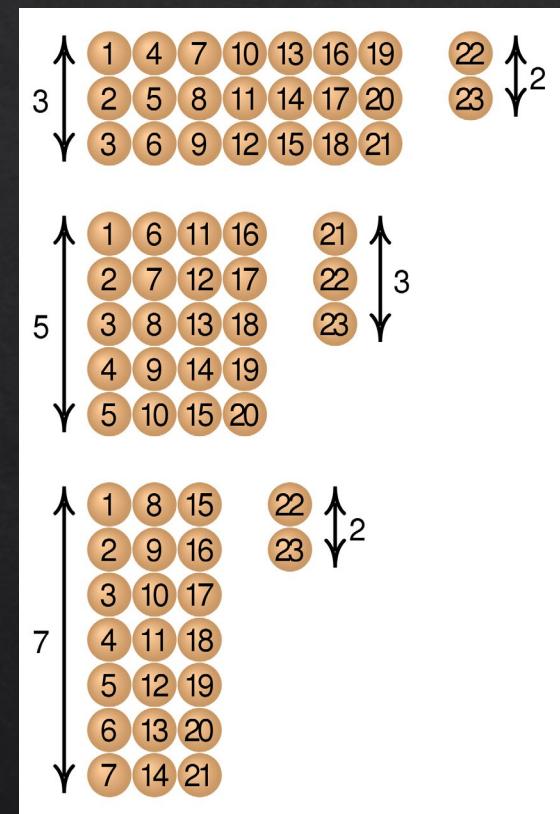
$$P = P_0 P_1 P_2 = 3 \times 5 \times 7 = 105$$

i	P_i	$c_i = P/P_i$	$d_i = c_i^{-1} \pmod{P_i}$	$c_i d_i u_i$
0	3	35	$35d_0 \equiv 1 \pmod{3}$ $2d_0 \equiv 1 \pmod{3}$ $d_0 = 2$	$35 \times 2 \times 2 = 140$
1	5	21	$21d_1 \equiv 1 \pmod{5}$ $d_1 = 1$	$21 \times 1 \times 3 = 63$
2	7	15	$15d_2 \equiv 1 \pmod{7}$ $d_2 = 1$	$15 \times 1 \times 2 = 30$

$$140 + 63 + 30 = 233 \pmod{105} = 23$$

Complexity = $O(k)$ integer multiplications

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{P}$$



Chinese Remainder Theorem

$p_0 = 2, p_1 = 3, p_2 = 5$ and $p_3 = 7$ and $(u_0, u_1, u_2, u_3) = (1, 2, 4, 3)$. What is u such that $u \leftrightarrow (1, 2, 4, 3)$?

$$P = P_0 P_1 P_2 P_3 = 2 \times 3 \times 5 \times 7 = 210$$

i	P_i	$c_i = P/P_i$	$d_i = c_i^{-1} \pmod{P_i}$	$c_i d_i u_i$
0	2	$15 \times 7 = 105$	$d_0 = 1/105 \pmod{2}$ $105d_0 = 1 \pmod{2}$ $d_0 = 1$	$105 \times 1 \times 1 = 105$

1	3	70	$70d_1 = 1 \pmod{3}$ $d_1 = 1$	$70 \times 1 \times 2 = 140$
2	5	42	$42d_2 = 1 \pmod{5}$ $2d_2 = 1 \pmod{5}$ $d_2 = 3$	$42 \times 3 \times 4 = 504$

3	7	30	$30d_3 = 1 \pmod{7}$ $2d_3 = 1 \pmod{7}$ $d_3 = 4$	$30 \times 4 \times 3 = 360$
---	---	----	--	------------------------------

$$u = \sum_{i=0}^{k-1} c_i d_i u_i \pmod{P}$$

$$1109 \pmod{210}$$

$$69 \pmod{210}$$

Google's Page Rank – In the Beginning

1996: Search engine BackRub: Studying the importance of links with his own homepage on Stanford website.

-> Later renamed "Page-Rank" after Larry Page. Two big ideas:

1. (A rough measure of) a site authority: The # of times it was linked to others
2. Automate and sanctify the search process and cope with the ever-increasing number of sites.

Aug 96: google.Stanford.edu

Talked w/ David Filo -> Formed their own company.

"Others assumed large servers were the fastest way to handle massive amounts of data. Google found networked PCs to be faster," – Google Info Web page.

Sep 99, beta label came off Google.com

Main Idea behind Page Rank

- ❖ Treat the web as a directed graph $G=(V,E)$
- ❖ Perform random walks by following 2 rules:
 - ❖ $d \times 100\%$ of the time, user follows (outgoing) links at random.
 - ❖ $(1-d) \times 100\%$ of the time, user picks a random page (not links to current web-page)
- ❖ Fundamental question: What is the probability that a given page will be visited? \rightarrow the "importance" (i.e. "rank") of the page.

Ranking Web-pages

$$PR(A) = \frac{1-d}{n} + d \left(\frac{PR(T_1)}{C(T_1)} + \dots + \frac{PR(T_m)}{C(T_m)} \right),$$

n: Total number of pages in the corpus (web)

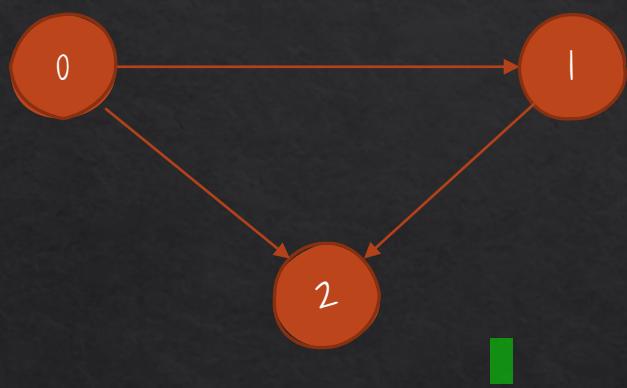
T_i : page pointing to page A

$C(T_i)$: The number of outgoing links of T_i

$PR(A)$ can be interpreted as the probability of visiting web-page A during the random walk obtained through the application following 2 rules we learnt.

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix

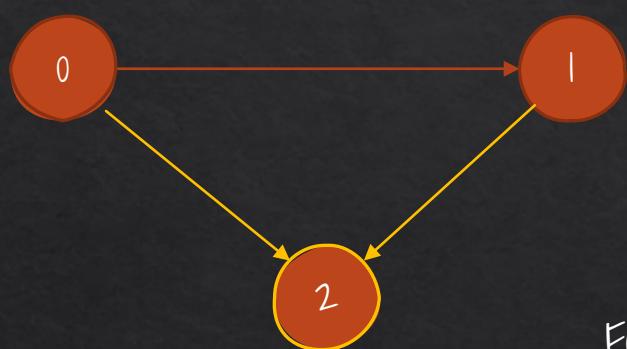
0	1	1
0	0	1
0	0	0

Adjacency Matrix Transposed

0	0	0
1	0	0
1	1	0

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix

0	1	1
0	0	1
0	0	0

Adjacency Matrix Transposed

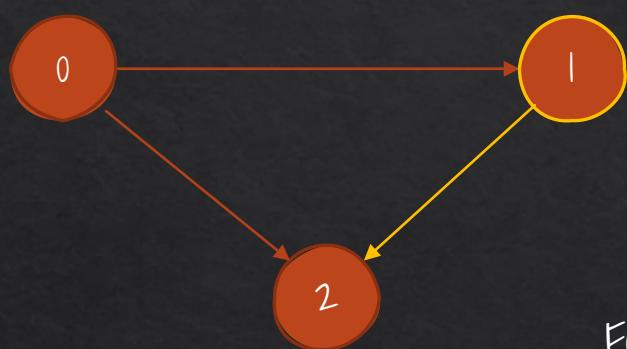
0	0	0
1	0	0
1	1	0

For every link (ij) in E that point to document d_j

$$a(d_j) = \sum_{(ij) \in E} h(d_i)$$

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix

0	1	1
0	0	1
0	0	0

Adjacency Matrix Transposed

0	0	0
1	0	0
1	1	0

For every link (ij) in E that point to document d_j

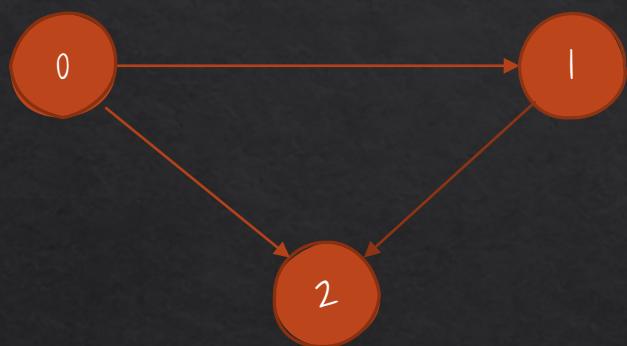
$$a(d_j) = \sum_{(ij) \in E} h(d_i)$$

For every link (jk) in E of documents d_k pointed by d_j

$$h(d_j) = \sum_{(jk) \in E} a(d_k)$$

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix A

0	1	1
0	0	1
0	0	0

Adjacency Matrix Transposed A^T

0	0	0
1	0	0
1	1	0

$$\begin{aligned}a(0) &= 0 \\a(1) &= h(0) \\a(2) &= h(0) + h(1)\end{aligned}$$

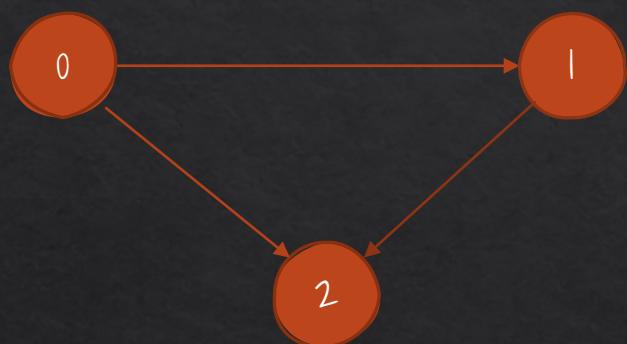
$$\begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} = A^T * \begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} \rightarrow a = A^T * h$$

$$\begin{aligned}h(0) &= a(1) + a(2) \\h(1) &= a(2) \\h(2) &= 0\end{aligned}$$

$$\begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} = A * \begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} \rightarrow h = A * a$$

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix A

0 1 1
0 0 1
0 0 0

Adjacency Matrix Transposed A^T

0 0 0
1 0 0
1 1 0

$$\begin{aligned}a(0) &= 0 \\a(1) &= h(0) \\a(2) &= h(0) + h(1)\end{aligned}$$

$$\begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} = A^T * \begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} \rightarrow a = A^T * h$$

$$\begin{aligned}h(0) &= a(1) + a(2) \\h(1) &= a(2) \\h(2) &= 0\end{aligned}$$

$$\begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} = A * \begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} \rightarrow h = A * a$$

Initialize $h: 1 1 1$, $a: 1 1 1$

Step 1: $a: 0 1 2$

$h: 3 2 0$

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix A

0	1	1
0	0	1
0	0	0

Adjacency Matrix Transposed A^T

0	0	0
1	0	0
1	1	0

$$\begin{aligned}a(0) &= 0 \\a(1) &= h(0) \\a(2) &= h(0) + h(1)\end{aligned}$$

$$\begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} = A^T * \begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} \rightarrow a = A^T * h$$

$$\begin{aligned}h(0) &= a(1) + a(2) \\h(1) &= a(2) \\h(2) &= 0\end{aligned}$$

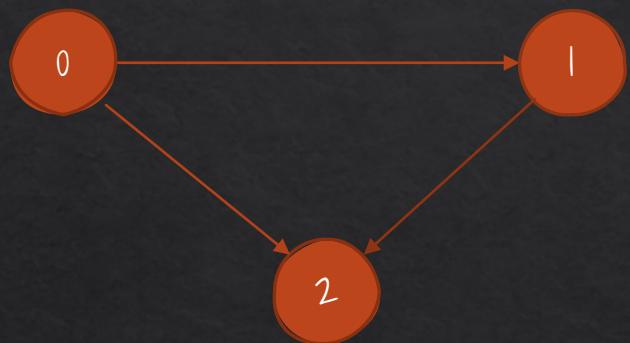
$$\begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} = A * \begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} \rightarrow h = A * a$$

Let's rewrite the formula to track iterations:

$$\begin{cases} a^{(i)} = A^T h^{(i-1)} \\ h^{(i)} = A a^{(i)} \end{cases}$$

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix A

0	1	1
0	0	1
0	0	0

Adjacency Matrix Transposed A^T

0	0	0
1	0	0
1	1	0

$$\begin{aligned} a(0) &= 0 \\ a(1) &= h(0) \\ a(2) &= h(0) + h(1) \end{aligned}$$

$$\begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} = A^T * \begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} \rightarrow a = A^T * h$$

$$\begin{aligned} h(0) &= a(1) + a(2) \\ h(1) &= a(2) \\ h(2) &= 0 \end{aligned}$$

$$\begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} = A * \begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} \rightarrow h = A * a$$

Let's rewrite the formula to track iterations:

$$\begin{aligned} a^{(i)} &= A^T h^{(i-1)} \\ h^{(i)} &= A a^{(i)} \end{aligned}$$

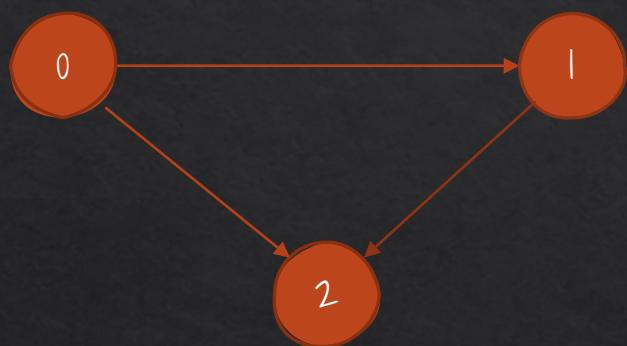
Power method:

$$\begin{aligned} a^{(1)} &= A^T h^{(0)} \\ a^{(2)} &= A^T h^{(1)} \rightarrow a^{(1)} = A^T A a^{(1)} \\ a^{(i)} &= A^T h^{(i-1)} \rightarrow a^{(i)} = A^T A a^{(i-1)} = (A^T A)^{i-1} a^{(1)} \end{aligned}$$

NJIT

Matrix Theory To The Rescue

understand how we can use a simple iterative algorithm, PR(A) can be computed in a few iterations



Adjacency List

0: 1 2
1: 2
2: -

Adjacency Matrix A

0	1	1
0	0	1
0	0	0

Adjacency Matrix Transposed A^T

0	0	0
1	0	0
1	1	0

$$\begin{aligned}a(0) &= 0 \\a(1) &= h(0) \\a(2) &= h(0) + h(1)\end{aligned}$$

$$\begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} = A^T * \begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} \rightarrow a = A^T * h$$

$$\begin{aligned}h(0) &= a(1) + a(2) \\h(1) &= a(2) \\h(2) &= 0\end{aligned}$$

$$\begin{bmatrix} h(0) \\ h(1) \\ h(2) \end{bmatrix} = A * \begin{bmatrix} a(0) \\ a(1) \\ a(2) \end{bmatrix} \rightarrow h = A * a$$

Doing the same for $h^{(i)}$, we have:

$$a^{(i)} = (A^T A)^{-1} a^{(i)}$$

$$h^{(i)} = A a^{(i)} = A A^T h^{(i-1)} = (A A^T)^{-1} h^{(i)}$$

Avoid a, h grows large by scaling to ensure they always stay within 0,1

What does convergence mean?

PageRank Algorithm

PageRank(G,V,E) //The code performs a SYNCHRONOUS Rank update

```
1. for all vertices u in V /* Initialization Step */  
2.   Src[u] = 1/n;  
3. small = something-small;  
4. while (convergence-distance > small) {  
5.   for all v in V  
6.     D[v]=0;  
7.     for(i=0;i<|V|;i++) {  
8.       Read-Adjacency-List(u,m,k1,k2,...,km); /*k1,k2,...,km: endpoints-of-outgoing edges */  
9.       for(j=1;j<=m;j++) /* m : out degrees of vertex u*/  
10.         D[kj] = D[kj] + Src[u]/m  
11.     }  
12.     for all v in V  
13.       D[v] = d * D[v] + (1-d)/n  
14.     convergence-distance = ||Src-D|| /* Euclidean distance */  
15.     Src=D;  
16. }
```

PageRank's Intricacy

Intricacy: Google's Technology page explains how the process gets more complicated:

PageRank relies on the uniquely democratic nature of the web by using its vast link structure as an indicator of an individual page's value. In essence, Google interprets a link from page A to page B as a vote, by page A, for page B. But Google looks at considerably more than the sheer volume of votes, or links the page receives. For example, it also analyzes the page that casts the vote. Votes cast by pages that are themselves "important" weigh more heavily and help to make other pages "important." Using these and other factors, Google provides its views on the pages' relative importance. And that's still only part of the protocol.

Performance: It's almost impossible to fathom, but PageRank considers more than 500 variables and 3 billion terms and still manages to deliver results in fractions of a second. Yet there also is a certain simplicity to the search process.

Searching Problems

Dictionary ADT

- Stores associations between keys and items (also called associative map)
- Operations :
 - (a) **insertElement(k,e)** – insert association between a key and item; replace if necessary
 - (b) **removeElement(k)** – remove association between key k and its element if it exists, else return NO_SUCH_KEY error
 - (c) **findElement(k)** – find the element associated with key k if it exists, else return EMPTY_ELEMENT
- Keys are from a set which may or may not be totally ordered but keys checked for “equality”.

Dictionary implementations

- Store keys as sequence (unordered) with new elements inserted at end.
 - **insertElement(k,e)** – O(1) time worst-case
 - **removeElement(k)** – O(n) time worst-case
 - **findElement(k)** – O(n) time worst-case
- **Hash tables**
 - Define a function $f : S \rightarrow \{0,1,\dots N-1\}$ where S is the set of keys and N is the table size.
 - Use this hash function to identify the index in the table where the key is stored. (similar to direct access in an array)
 - Since it is not 1-1 function, more than one key may map into the same index in the table causing collision.
 - Ideally f should distribute keys evenly in the table.

Hash tables

- Hash function f is composed of two functions
 $f_1 : S \rightarrow Z$ and $f_2 : Z \rightarrow \{0, 1, \dots, N-1\}$ where S is set of integers.
 f_1 is called “**hashcode**” function (defined in Java Object) and f_2 is called “compression map”
- **Hashcode** function should be representative of all fields of an object and for a primitive type (e.g. integer) representative of all bits in integer. It is defined independent hash table size.
- Polynomial hash codes:

$$x_{k-1} a^{k-1} + x_{k-2} a^{k-2} + \dots + x_1 a + x_0$$

where x_{k-1}, x_{k-2}, \dots are integer representations of components of key object and a is a constant not equal to 1

-- 33, 37, 39, 41 are found to be good choices for character strings.

Example hashCode in Java

```
public class Key implements Comparable<Key> {  
    private final String firstName, lastName;  
    public Key(String fName, String lName) { .. }
```

```
@Override
```

```
public int hashCode() {  
    int hash = 17 + firstName.hashCode();  
    hash = hash * 31 + lastName.hashCode();  
    return hash;  
}
```

```
@Override
```

```
public boolean equals(Object obj) {....}
```

- $a.equals(b) \Rightarrow a.hashCode() == b.hashCode()$ but not vice versa.

Hash tables (contd.)

Compression Maps

- **Division** method:

$h(k) = |k| \bmod N$, k is hash code and choose N as a prime number so as to distribute hash values evenly among table indices.

- **MAD** (Multiply, Add and Divide) method:

$h(k) = |ak+b| \bmod N$ where a and b are integers randomly chosen .s.t. $a,b \geq 0$ and $a \bmod N \neq 0$

-- provides close to “ideal” hash function where
Prob(two keys hash into same value) $\approx 1/N$

Collision resolution

- Each location in hash table is called “bucket”.
- (a) Chaining, (b) Open addressing
- Chaining – keep colliding keys in a list or sequence called “chain” in the same bucket
 - in `findElement(k)`, after hashing need to search for keys in the chain
 - in `insertElement(k,e)` and `removeElement(k)`, need to hash into bucket and then insert/remove element in/from chain
- Load factor – n/N (n number of items in hash table)
preferably $n/N < 1$
- Expected time complexity – $O(\lceil n/N \rceil)$ ($O(1)$ if n is $O(N)$)

Open Addressing

- Open Addressing – no chaining of keys that have same hash value
- Probes other locations for the key
 - Suppose $i = h(k)$. Probe sequence of locations $(i+f(j)) \bmod N$, $j = 0, 1, 2, \dots$ until the key is found for search/remove or until an empty slot is found for insert
 - if $f(j) = j$ for all j , it is called “linear probing”
 - if $f(j) = j^2$, it is called “quadratic probing”
 - if $f(j) = j \cdot g(k)$ where $g(k)$ is another hashing function, it is called “double hashing”
 - Faster than chaining for search/insert but removal is complicated as there should not be any “holes” in sequence for a particular $h(k)$
 - Tends to introduce clusters of keys in the table for linear and quadratic probing.

Universal Hashing

- A family of hash functions that minimizes expected number of collisions
- Stated formally, let H be a subset of functions from $[0, M-1]$ to $[0, N-1]$ satisfying the property that for any randomly chosen function h from H and for any two integers j, k in $[0, M-1]$,
$$\Pr(h(j) = h(k)) \leq 1/N$$

- Implies that $E[\# \text{ of collisions between } j \text{ and "n" integers from } [0, M-1]] \leq n/N$
- Set of hash functions of form $(ak + b \bmod p) \bmod N$

where $0 < a < p$, $0 \leq b < p$ where p is a prime number with $M \leq p < 2M$ (M is number of hash codes)

can be shown to be “universal”.

- All Dictionary ADT operations can be done in expected time $O([n/N])$ using a randomly chosen hash function from this set and chaining.

FFT computations

CS 610 Spring 2019
Instructor : Ravi Varadarajan

April 25, 2019

Remainder theorem: The value of a polynomial $p(x)$ at a point, i.e. $p(a)$ is equal to the remainder when $p(x)$ is divided by $x - a$.

Note $\text{DFT}([a_0, a_1, a_2, \dots, a_{n-1}]) = [p(\omega^0), p(\omega^1), \dots, p(\omega^{n-1})]$ where ω is a principal n -th root of unity in a commutative ring and $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$.

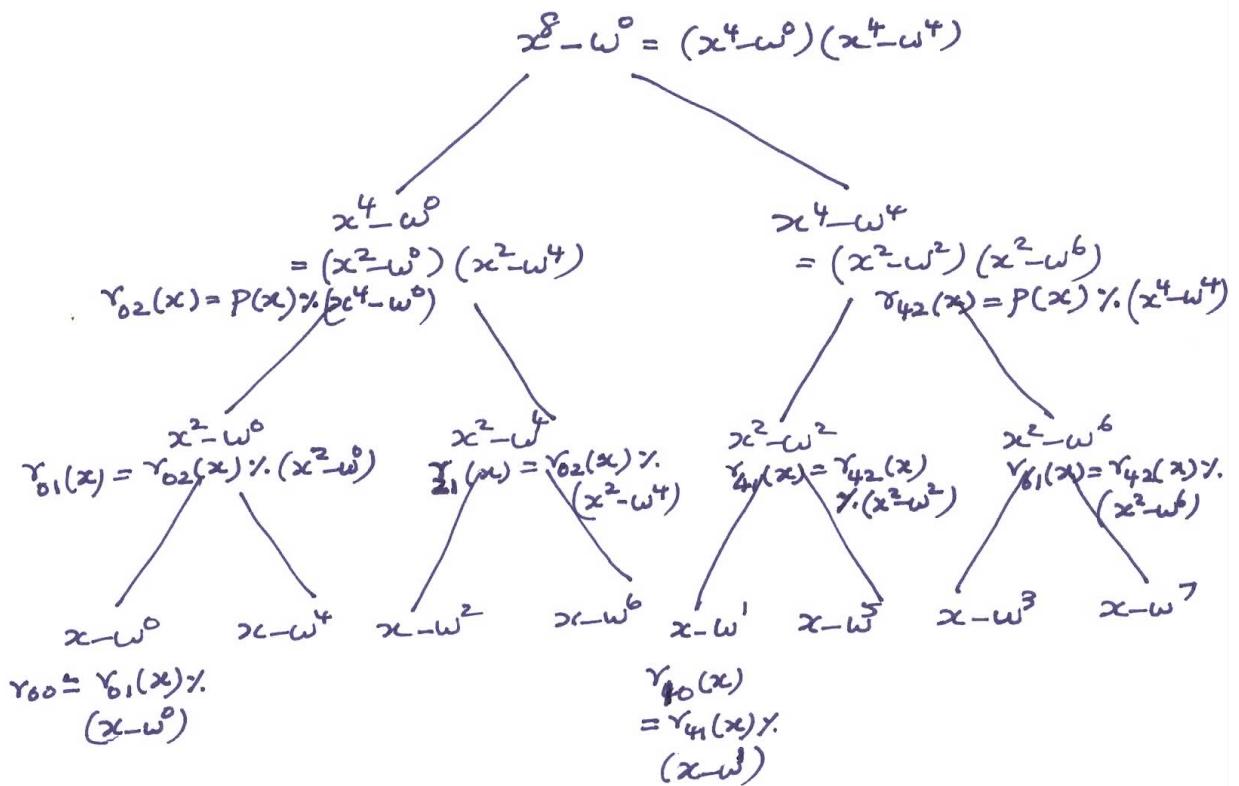
By using remainder theorem, to compute DFT, we just need to compute remainders of $p(x)$ when divided by $(x - \omega^0), (x - \omega^1), (x - \omega^2), \dots, (x - \omega^{n-1})$ respectively. We can do this cleverly by using the following fact :

Suppose $r(x)$ is the remainder polynomial when $p(x)$ is divided by the product polynomial $d_1(x)d_2(x)$. Then the remainder polynomial $r_1(x)$ when $p(x)$ is divided by $d_1(x)$ is the same as remainder polynomial when $r(x)$ is divided by $d_1(x)$.

As an example we use the following tree of product polynomials for successive divisions to get remainder polynomials. In the tree, the parent polynomial is the product of children polynomials. For example, $(x - \omega^0)(x - \omega^4) = (x^2 - \omega^0)$ and $(x^2 - \omega^0)(x^2 - \omega^4) = (x^4 - \omega^0)$.

Product + Remainder polynomials

Notation: Remainder polynomial when $p(x)$ is divided by $g(x)$ denoted as $p(x) \text{ mod } g(x)$ (or $p(x) \% g(x)$)



Let's us illustrate for the case of $n = 8$. Let $r_{02}(x), r_{42}(x)$ be the remainder polynomials when $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_7x^7$ is divided by $(x^4 - w^0)$ and $(x^4 - w^4)$ respectively. It can be shown that

$$r_{02}(x) = (a_0 + \omega^0 a_4) + (a_1 + \omega^0 a_5)x + (a_2 + \omega^0 a_6)x^2 + (a_3 + \omega^0 a_7)x^3$$

and

$$r_{42}(x) = (a_0 + \omega^4 a_4) + (a_1 + \omega^4 a_5)x + (a_2 + \omega^4 a_6)x^2 + (a_3 + \omega^4 a_7)x^3.$$

If we order coefficients of these polynomials, we get $(a_0 + \omega^0 a_4), (a_1 + \omega^0 a_5), (a_2 + \omega^0 a_6), (a_3 + \omega_0 a_7), (a_0 + \omega^4 a_4), (a_1 + \omega^4 a_5), (a_2 + \omega^4 a_6), (a_3 + \omega_4 a_7)$. This is exactly what we get after the first stage of computation in the butterfly network by pairing elements in positions which differ in the $k = 3$ -th bit (i.e. most significant position). The powers of ω to be used in each position is given by left shifting by $k - 1 = 2$ bits after bit reversal of bits representing that position. For example for $6 = (110)_2$, we use $(100)_2 = 4$. Let these values be denoted by b_0, b_1, \dots, b_7 .

Now if $r_{01}(x), r_{21}(x)$ are the remainder polynomials when $r_{02}(x)$ is divided by $(x^2 - \omega^0)$ and $(x^2 - \omega^4)$ respectively, then we get

$$r_{01}(x) = (b_0 + \omega^0 b_2) + (b_1 + \omega^0 b_3)x$$

and

$$r_{21}(x) = (b_0 + \omega^4 b_2) + (b_1 + \omega^4 b_3)x$$

Also if $r_{41}(x), r_{61}(x)$ are the remainder polynomials when $r_{42}(x)$ is divided by $(x^2 - \omega^2)$ and $(x^2 - \omega^6)$ respectively, then we get

$$r_{41}(x) = (b_4 + \omega^2 b_6) + (b_5 + \omega^2 b_7)x$$

and

$$r_{61}(x) = (b_4 + \omega^6 b_6) + (b_5 + \omega^6 b_7)x$$

If we order coefficients of $r_{01}(x), r_{21}(x), r_{41}(x), r_{61}(x)$, we get $(b_0 + \omega^0 b_2), (b_1 + \omega^0 b_3), (b_0 + \omega^4 b_2), (b_1 + \omega^4 b_3), (b_4 + \omega^2 b_6), (b_5 + \omega^2 b_7), (b_4 + \omega^6 b_6), (b_5 + \omega^6 b_7)$. This is precisely what we get after the second stage of computation in the butterfly network by pairing elements in positions which differ in the $k - 1 = 2$ -th bit. The powers of ω to be used in each position is given by left shifting by $k - 2 = 1$ bit after bit reversal of bits representing that position. For example for $7 = (111)_2$, we use $(110)_2 = 6$.

Finally we get remainders r_{00}, r_{40} when r_{01} is divided by $(x - w^0)$ and $(x - w^4)$,

remainders r_{20}, r_{60} when r_{21} is divided by $(x - w^2)$ and $(x - w^6)$,

remainders r_{10}, r_{50} when r_{41} is divided by $(x - w^1)$ and $(x - w^5)$, and

remainders r_{30}, r_{70} when r_{61} is divided by $(x - w^3)$ and $(x - w^7)$.

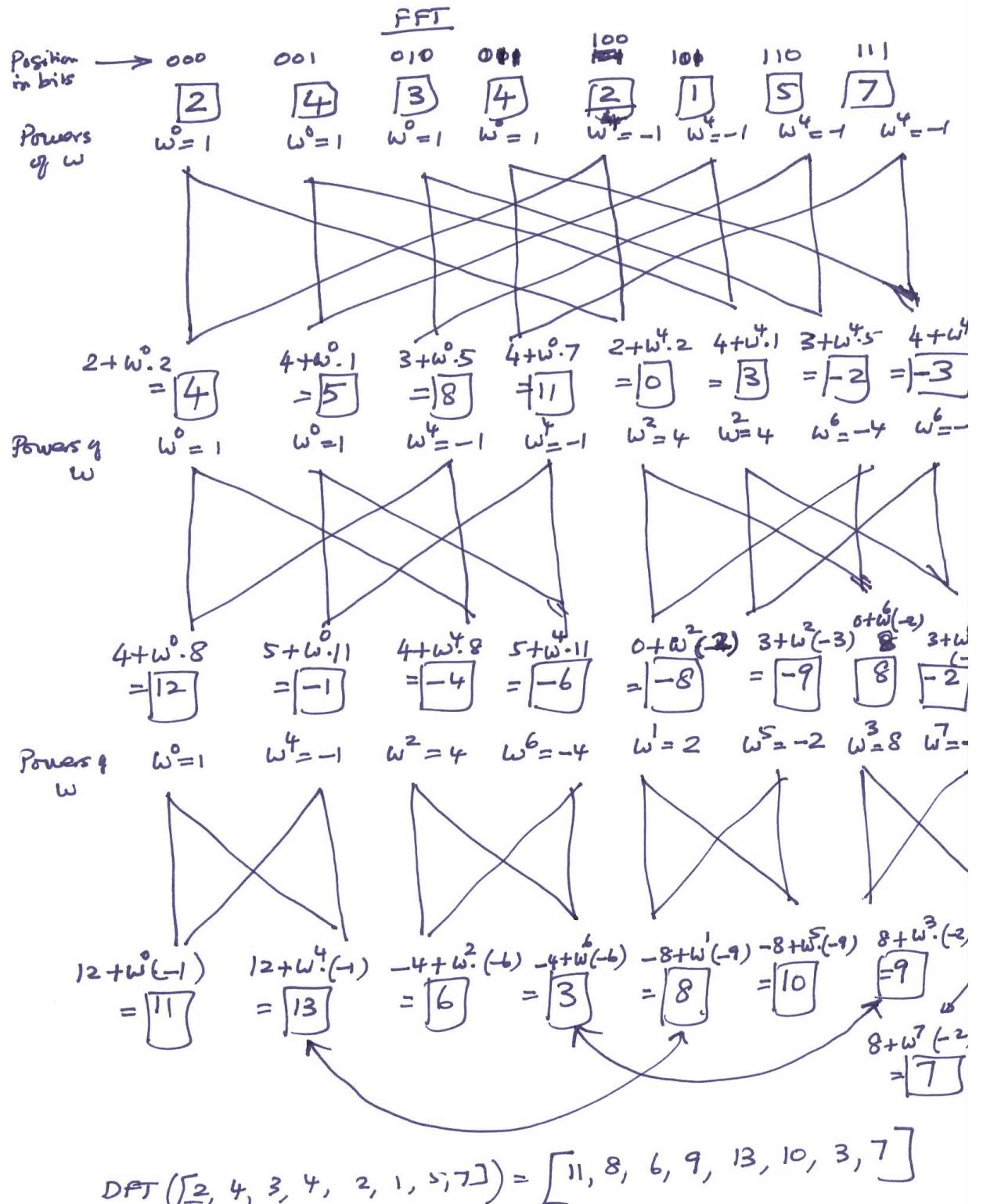
These remainders are exactly the values of DFT in positions 0, 4, 2, 6, 1, 5, 3, 7 permuted in bit reversal order.

Example : Compute DFT of $[2, 4, 3, 4, 2, 1, 5, 7]$ in commutative ring $[0, 1, 2, \dots, 16]$ where $+$ and $*$ operations are modulo 17. Note that $16 \equiv -1, 15 \equiv -2, \dots, 1 \equiv -16$

First let us determine ω , 8-th principal root of unity in this ring, i.e. $\omega^8 \equiv 1 \pmod{17}$. Hence $\omega = 2$. Thus $\omega^0 = 1, \omega^1 = 2, \omega^2 = 4, \omega^3 = 8, \omega^4 = 16 \equiv -1 \pmod{17}, \omega^5 = -2, \omega^6 = -4, \omega^7 = -8$

In the butterfly network, there are $\log n$ stages, where at each stage we pair elements to do computations. For a pair with coefficients (b_i, b_j) , the new coefficients to be used for next stage at positions i and j are computed as $c_i = b_i + \omega^{p_i} b_j$ and $c_j = b_i + \omega^{p_j} b_j$ where the powers of ω , namely p_i and p_j are obtained by reversing the bits of i and j and left shifting $k - m - 1$ bits for stage $0 \leq m < \log n$ computation.

In the diagram below we show 3 stages, $m = 0, 1, 2$. The powers of ω to be used in these positions are indicated below the elements.



In the first stage, i.e., when $m = 0$, computations are done by pairing elements in positions that differ in the most significant bit, i.e. 0 with 4, 1 with 5, 2 with 6 and 3 with 7. For example, new values for position 0 = $a_0 + \omega^0 a_4 = 2 + 2 = 4$ and for position 4 = $a_0 + \omega^4 a_4 = 2 - 2 = 0$.

In the second stage, i.e, when $m = 1$, computations are done by pairing elements in positions that differ in the 2nd bit, i.e. 0 with 2, 1 with 3, 4 with 6 and 5 with 7. For example, new values for position $0 = 4 + \omega^0 * 8 = 12$ and for position $2 = 4 + \omega^4 * 8 = 4 - 8 = -4$. New values for position $4 = 0 + \omega^2 * (-2) = -8$ and for position $6 = 0 + \omega^6 * (-2) = 8$.

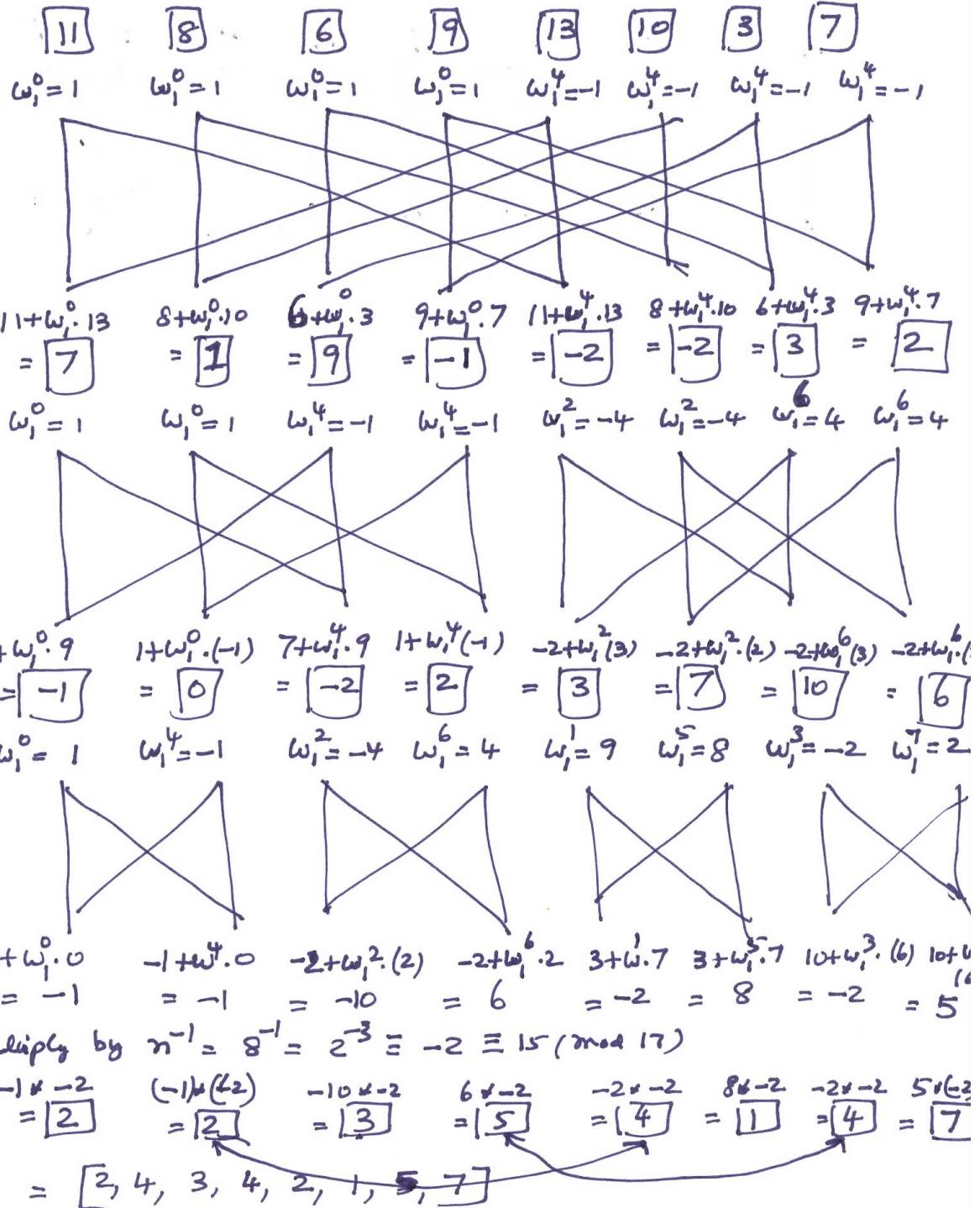
In the third stage, i.e, when $m = 2$, computations are done by pairing elements in positions that differ in the 3rd bit, i.e. 0 with 1, 2 with 3, 4 with 5 and 6 with 7. For example, new values for position $0 = 12 + \omega^0 * (-1) = 11$ and for position $1 = 12 + \omega^4 * (-1) = 13$. New values for position $6 = 8 + \omega^3 * (-2) = -8$ and for position $7 = 8 + \omega^7 * (-2) = 7$.

The DFT vector is given by $[11, 8, 6, 9, 13, 10, 3, 7]$. Note that elements in positions 1 and 4 were switched, as well positions in 3 and 6.

Let's compute inverse DFT of the DFT vector to see if we get back the original vector. For inverse FFT, $\omega_1 = \omega^{-1} = 2^{-1} \equiv 9 \pmod{17} \equiv -8 \pmod{17}$. The powers of this root are : $\omega_1^0 = 1, \omega_1^1 = 9, \omega_1^2 = -4, \omega_1^3 = -2, \omega_1^4 = -1, \omega_1^5 = 8, \omega_1^6 = 4, \omega_1^7 = 2$. We do the exat type of computations as in FFT but at the end we need to multiply the values by $n^{-1} = 8^{-1} = 15 \equiv -2 \pmod{17}$. In the diagram below at the end, we got back the original vector for which we computed the DFT before.

Inverso PPT

$$\omega_1 = \omega^{-1} = 2^{-1} \equiv 9 \pmod{17} \equiv -8 \pmod{17}$$



Graph algorithms

Graphs

- A directed graph $G = (V, E)$, V is set of vertices and E is set of edges, i.e. subset of $V \times V$
- A digraph defines a binary relation on V .
- Useful representation in many applications (e.g. web page links, transportation routes, semantic, social networks)
- $\text{InEdges}(v) = \{ (u, v) \mid (u, v) \in E \}$ and $\text{outEdges}(v) = \{ (v, u) \mid (v, u) \in E \}$
- $\text{indeg}(v) = |\text{InEdges}(v)|$ $\text{outDeg}(v) = |\text{outEdges}(v)|$
- $\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = |E|$
- $|E| \leq n^2$ where n is number of vertices and $|E| \leq n(n-1)$ if there are no self-cycles.

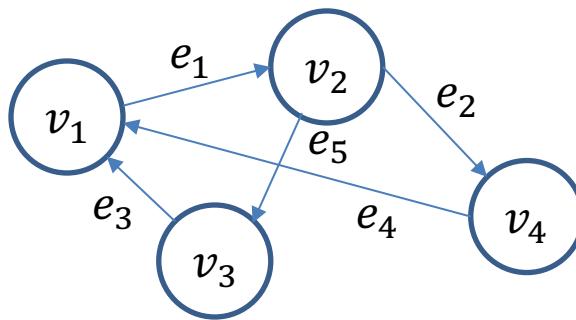
Undirected graphs

- In undirected graph, no edge orientation.
 $(u,v) \in E \rightarrow (v,u) \in E$
- Defines a symmetric relation on V . Not reflexive i.e. (u,u) not in E .
- $\text{incidentedges}(v) = \{ (u,v) \mid (u,v) \in E \text{ or } (v,u) \in E \}$
- $\deg(v) = |\text{incidentedges}(v)|$
- $\sum_{v \in V} \deg(v) = 2 |E|$
- $|E| \leq n(n-1)/2$

Graph data Structures

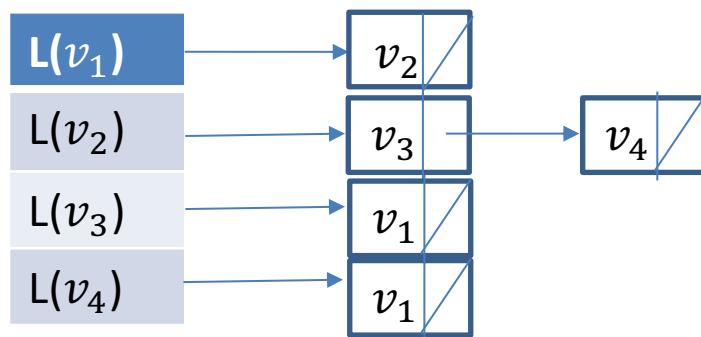
- Adjacency matrix A – $n \times n$ boolean matrix where $A(i,j) = 1$ iff $(v_i, v_j) \in E$ where $V = \{v_1, v_2, \dots, v_n\}$
 - Useful for matrix operations to solve all-pair problems
- Incidence matrix B – $n \times m$ matrix where $B(i,j) = 1$ if $e_j = (v_i, v_k)$ for some k and $B(i,j) = -1$ if $e_j = (v_k, v_i)$ for some k . We assume $V = \{v_1, v_2, \dots, v_n\}$ and $E = \{e_1, e_2, \dots, e_m\}$ – Useful for analysis of some cycle problems
- Adjacency List – Array $L[0..n-1]$ of lists where $L[i]$ contains the list of indices of vertices v_j such that $(v_i, v_j) \in E$
Space is $O(n+m)$ as opposed to $O(n^2)$ for adjacency matrix. Useful for many efficient graph algorithms.

Graph representations (example)



$$A = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} e_1 & e_2 & e_3 & e_4 & e_5 \\ 1 & 0 & -1 & -1 & 0 \\ -1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 \end{bmatrix}$$

Adj lists:



Graph Search

- Two approaches:
 - (a) Breadth-first search - From start node s , get vertices immediately reachable from s and fan-out from each of those nodes and so on

BFS(G):

Input : $G=(V,E)$ as adjacency list

Output: List of nodes visited in BFS order

$q \leftarrow$ empty queue; $bfsList \leftarrow \{\}$

for $i \leftarrow 0$ to $n-1$

$visited[i] \leftarrow$ false

for $i \leftarrow 0$ to $n-1$

 if $\neg visited[v_i]$

$q.enqueue(v_i)$

$visited[s] \leftarrow$ true

 while $\neg q.empty()$

$v \leftarrow q.dequeue()$

$bfsList.append(v)$

 for each w in $G.adjList[v]$

 if $\neg visited[w]$

$visited[w] \leftarrow$ true

$q.enqueue(w)$

Return $bfsList$

Graph Search (contd.)

- BFS takes $O(n)$ additional space and $O(n+m)$ time primitive queue, list operations, assignment and boolean operations
- Depth-first search is a recursive traversal exploring a path before back-tracking by a step and trying a different path from that vertex.

DFSMain(G, s):

Input : $G=(V,E)$ as adjacency list and s start vertex index

Output: List of nodes visited in DFS order

```
dfsList ← {}
for i ← 0 to n-1
    visited[i] ← false
for i ← 0 to n-1
    if !visited( $v_i$ )
        DFS( $G, v_i, \text{dfsList}$ )
```

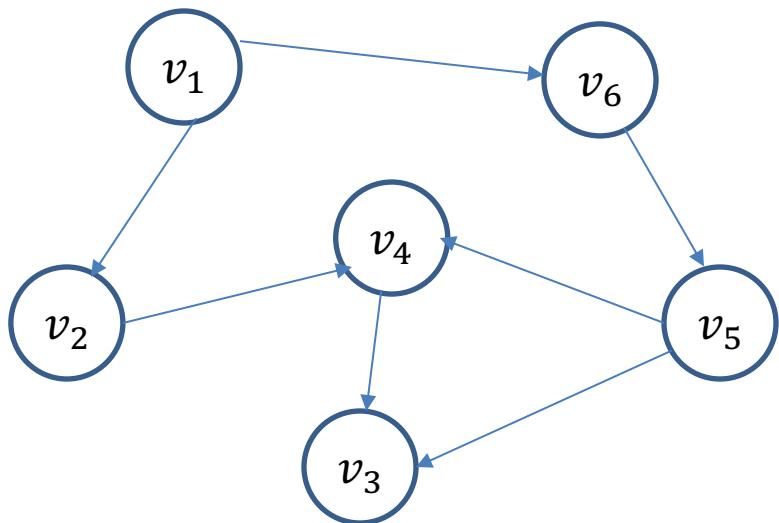
DFS($G, v, \text{dfsList}$):

```
visited[v] ← true
dfsList.append(v)
for each w in  $G.\text{adjList}[v]$ 
    if !visited[w]
        DFS( $G, w, \text{dfsList}$ )
```

Return dfsList

- Takes $O(L)$ additional stack space (L - longest path length from v) and $O(n+m)$ time

Graph search example



Graph search example - BFS

- Start vertex : 1; $Q = \{v_1\}$; $bfslist = []$
- Remove v_1 from Q, $bfslist=[v_1]$; $Q = \{v_2, v_6\}$
- Remove v_2 , $bfslist=[v_1, v_2]$; $Q = \{v_6, v_4\}$
- Remove v_6 , $bfslist=[v_1, v_2, v_6]$; $Q = \{v_4, v_5\}$
- Remove v_4 , $bfslist=[v_1, v_2, v_6, v_4]$; $Q = \{v_5, v_3\}$
- Remove v_5 , $bfslist=[v_1, v_2, v_6, v_4, v_5]$; $Q = \{v_3\}$
- Remove v_3 , $bfslist=[v_1, v_2, v_6, v_4, v_5, v_3]$; $Q = \{\}$

Graph search example - DFS

- Start vertex : v_1 ; dfslist = {}

DFS(v_1) dfslist = { v_1 }

DFS(v_2) dfslist = { v_1, v_2 }

DFS(v_6) dfslist = { v_1, v_2, v_4, v_3, v_6 }

DFS(v_4) dfslist = { v_1, v_2, v_4 } DFS(v_5) dfslist = { $v_1, v_2, v_4, v_3, v_6, v_5$ }

DFS(v_3) dfslist = { v_1, v_2, v_4, v_3 }

Undirected graph connectivity

- A graph is connected if there is a path between every pair of vertices in G.
- A connected component is a maximal subgraph of G which is connected.
- Can use BFS and DFS to find connected components of a G in $O(n+m)$ time.
- A spanning tree of a connected graph G is a subgraph of G which is connected and has minimum number of edges. It has $n-1$ edges where $n = |V|$
- BFS and DFS provide spanning trees.
- A spanning forest is a collection of spanning trees of a graph that contains all the vertices.

Cycles in graphs

- A cycle is a path that starts and ends with the same vertex. A simple cycle is a cycle which itself does not contain any cycles.
- A tree does not have any cycle. It contradicts minimal subgraph property.
- A directed graph which does not have any directed cycles is called a directed acyclic graph (DAG)
- Examples of DAG : task precedence graph, PERT charts used in project management, block-chains used in crypto-currency technologies
- **Topological sorting** in DAG : Assign unique number to each of the vertices such that there is no edge from higher indexed vertex to lower indexed vertex.

Topological sorting of DAG

- There exists at least one vertex (source) whose indegree = 0; Symmetrically, at least one vertex (sink) whose outdegree = 0. Why ?
- Idea:
 - (i) We can first number all source vertices.
 - (ii) If we remove source vertices from graphs along with their outgoing edges, resulting graph is still a DAG and should have at least one source vertex. These edges will be bound to be directed from a lower numbered to higher numbered vertex.
 - (iii) Repeat steps (i) and (ii) until we have an empty graph.

This can be done in $O(n+m)$ time with additional space complexity of $O(n)$

K-connectedness

- An undirected graph is k-connected if it requires the removal of at least k vertices to make the graph disconnected.
- For example, computer networks are less resistant to failure of a gateway if it has higher connectivity.
- A tree is a 1-connected graph
- A bi-connected graph is 2-connected. This means that for every pair of vertices (u,v) , there is a cycle that includes u and v
- A bi-connected component of a graph is a maximal bi-connected subgraph of a graph. Note that bi-connected components create a partition of edge set.

Graph algorithms

K-connectedness

- An undirected graph is k-connected if it requires the removal of at least k vertices to make the graph disconnected.
- For example, computer networks are less resistant to failure of a gateway if it has higher connectivity.
- A tree is a 1-connected graph
- A bi-connected graph is 2-connected. This means that for every pair of vertices (u,v) , there is a cycle that includes u and v
- A bi-connected component of a graph is a maximal bi-connected subgraph of a graph. Note that bi-connected components create a partition of edge set.

Bi-connectivity algorithm

- An articulation point (separation vertex) of a graph G is a vertex whose removal splits G into at least 2 components. A bi-connected graph has no articulation points
- In DFS of a (undirected) graph there must be either forward edges (of spanning tree) or back edges from a descendant to an ancestor.
- If there is a cycle in graph involving vertex v then there must be a **back-edge** (x,y) from the depth-first sub tree of v where x is a descendant of v and y is an ancestor of v .
- Define $DF(u)$ be depth-first number of vertex u .
- Define $LOW(v) = \min\{ DF(v) \cup \{DF(y) \mid \exists (x,y) \in E \text{ s.t. } x \text{ is a descendant of } v \text{ and } y \text{ is an ancestor of } v \text{ in DFS tree}\} \}$

Bi-connectivity alg. proof

Claim :

v is an articulation point or a separation vertex iff either

(a) v is the root and v has more than one child

or

(b) v is not the root and has a child w such that $\text{LOW}(w) \geq \text{DF}(v)$.

Proof:

(i) (if-part) If root v has two tree edges (v,x) and (v,y) then there is no edge between subtrees T_1 and T_2 rooted at x and y .

→ every path from a vertex in T_1 to a vertex in T_2 has to have vertex v
→ v is an articulation point

(only-if) We use proof by contradiction. Suppose v is the root and an articulation point but has only one child x .

→ Removal of v along with edge (v,x) does not disconnect the graph of vertices reachable from v .

→ This is a contradiction as v is an articulation point.

Hence v must have more than one child.

Bi-connectivity alg. Proof (contd.)

(ii)

(if-part) v is not the root and v has a child w such that $\text{LOW}[w] \geq \text{DF}(v)$.

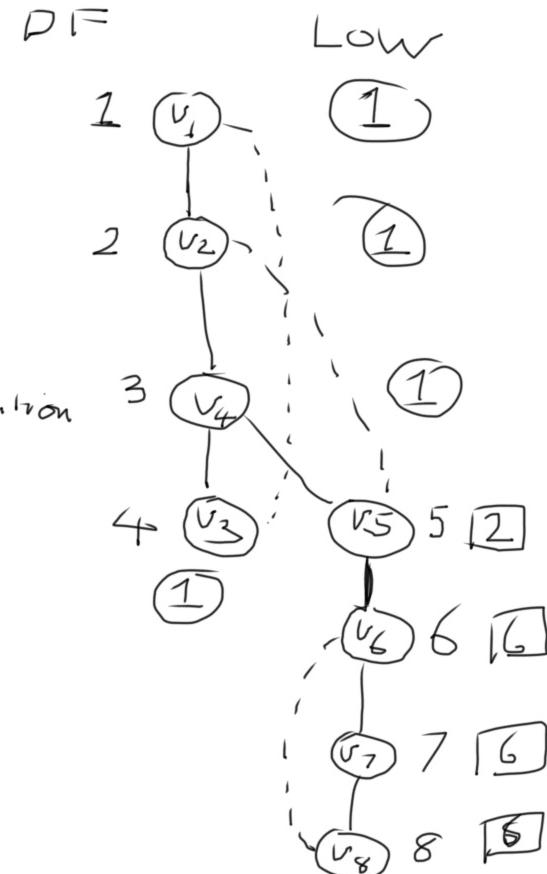
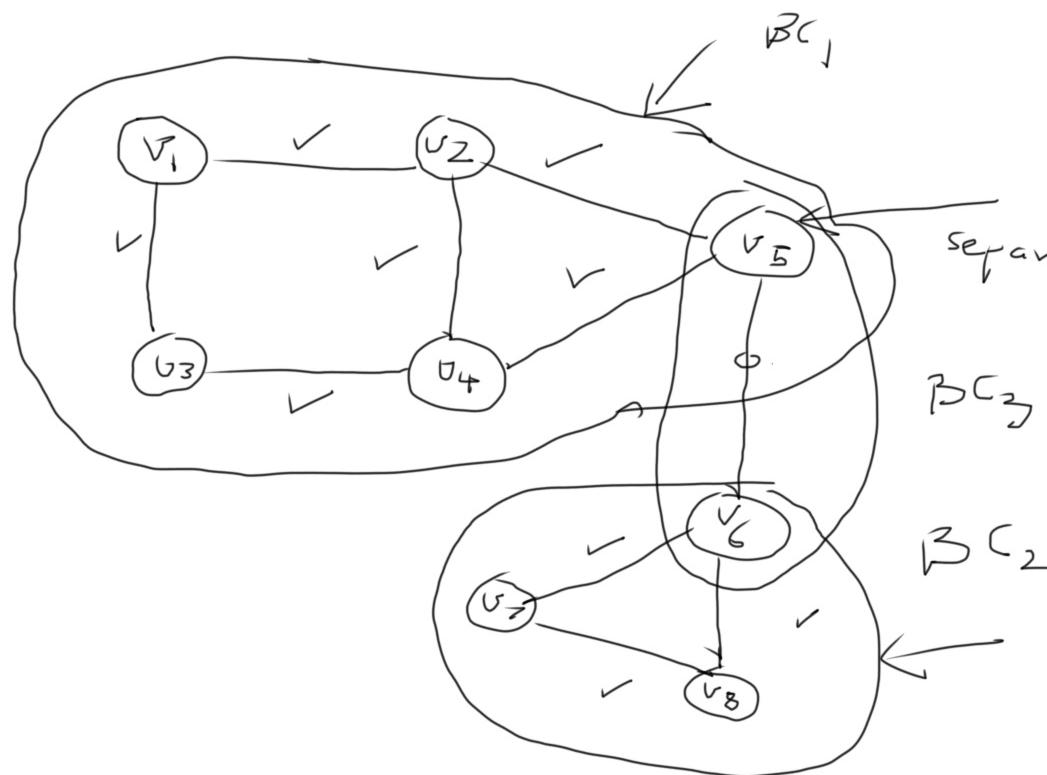
Let's prove by contradiction. Suppose v is not articulation point.
→ there is a path from w to a proper ancestor of v , say y that bypasses v .

→ \exists a back edge (x,y) where $\text{DF}(x) \geq \text{DF}(w)$ and $\text{DF}(y) < \text{DF}(v)$
→ $\text{LOW}(w) \leq \text{DF}(y) < \text{DF}(v)$ which contradicts defn. of LOW

Hence v is an articulation point

(only-if) Left as exercise.

Example



Since v_5 has child v_6 with $LOW(v_6) = 6 \geq DF(v_5) = 2$, v_5 is an articulation point
 Since v_6 has child v_7 with $LOW(v_7) = 6 \geq DF(v_6) = 6$, v_6 is an articulation point

Bi-connectivity algorithm

DFSB(G,v,count,isRoot,S):

Output: Articulation points of G and max DF value in subtree of v

visited[v] \leftarrow true; DF[v] \leftarrow count

LOW[v] \leftarrow DF[v]

nChildren \leftarrow 0

for each vertex w in G.adjList[v]

if !visited[w]

parent[w] \leftarrow v;

nChildren \leftarrow nChildren + 1

count \leftarrow DFSB(G,w,count+1,false,S)

if (isRoot and nChildren > 1) or (!isRoot and LOW[w] \geq DF[v])

S.add(v) // v is an articulation point

LOW[v] \leftarrow min(LOW[v], LOW[w])

else if w != parent[v]

LOW[v] \leftarrow min(LOW[v], DF[w])

Return count

Bi-connectivity algorithm (contd.)

Biconnected(G):

Input : Graph $G=(V,E)$ with adjacency list data structure

Output: Set of articulation points

for $v \leftarrow 0$ to $n-1$

$\text{visited}[v] \leftarrow \text{false}$

$\text{count} \leftarrow 0$; $S \leftarrow \{\}$

 for $v \leftarrow 0$ to $n-1$

 if $\text{!visited}[v]$

$\text{count} \leftarrow \text{DFSB}(G,v,\text{count}+1,\text{true},S)$

Return S

Time complexity : $O(n+m)$ and additional space complexity $O(L)$

Graph algorithms

Digraph strong connectivity

- A digraph $G=(V,E)$ is strongly connected if there is a directed path between every pair of vertices (u,v)
- A strongly connected component is a maximal subgraph that is strongly connected.
- Unlike bi-connected components, strongly connected components do not partition the edges but only vertices.
- In a DFS tree of a directed graph, there are 3 types of non-tree edges (u,v)
 - (a) back edge from a descendant to an ancestor ($DF(v) < DF(u)$)
 - (b) forward edge from an ancestor to a descendant ($DF(v) > DF(u)$)
 - (c) cross edge (to left in tree) where neither of u and v is a descendant of another ($DF(v) < DF(u)$)

Strong connectivity algorithm basis

- We use DFS and LOW computation similar to bi-connectivity algorithm.

- Define

$$\text{LOWLINK}(v) = \min\{ \text{DF}(v) \cup \{\text{DF}(y) \mid \exists (x,y) \in E \text{ s.t. } x \text{ is a descendent of } v \text{ and } (a) \text{ either } (x,y) \text{ is a back-edge or } (b) (x,y) \text{ is a cross edge and root of the strongly connected component containing } y \text{ is an ancestor of } v \} \}$$

- **Claim:** v is root of a strongly connected component in DFS tree, iff $\text{LOWLINK}(v) = \text{DF}(v)$

Strong connectivity algorithm basis

Proof:

- **If part :** proof by contradiction. Suppose $\text{LOWLINK}(v) = \text{DF}(v)$ but v is not the root of a strongly connected component in DFS tree

→ let r be a proper ancestor of v in DFS tree and be the root of component containing v

→ \exists a directed path from v to r in the graph and let w be first vertex in this path such that w is a descendant of v and (w,x) is a non-tree edge with $\text{DF}(x) < \text{DF}(v)$

Then there is a path from r to v and then v to r that contains w and w must be in the same component as v and r

→ $\text{LOWLINK}(v) \leq \text{DF}(x) < \text{DF}(v)$, contradiction

- **Only-if part:** left as an exercise.

$O(n+m)$ Strong connectivity alg.

DFSB(G,v,count):

Output: strongly connected components detected during DFS of v

visited[v] \leftarrow true; DF[v] \leftarrow count

LOWLINK[v] \leftarrow DF[v]

st.push(v); inStack[v] \leftarrow true

for each vertex w in G.adjList[v]

if !visited[w]

 count \leftarrow **DFSB**(G,w,count+1)

 LOWLINK[v] \leftarrow min(LOWLINK[v], LOWLINK[w])

else if DF[w] < DF[v] and inStack[w]

 LOWLINK[v] \leftarrow min(LOWLINK[v], DF[w])

if LOWLINK[v] = DF[v]

 print “beginning of a strongly connected component”

 while !st.isEmpty()

 w \leftarrow st.pop(); print w ; inStack[w] \leftarrow false

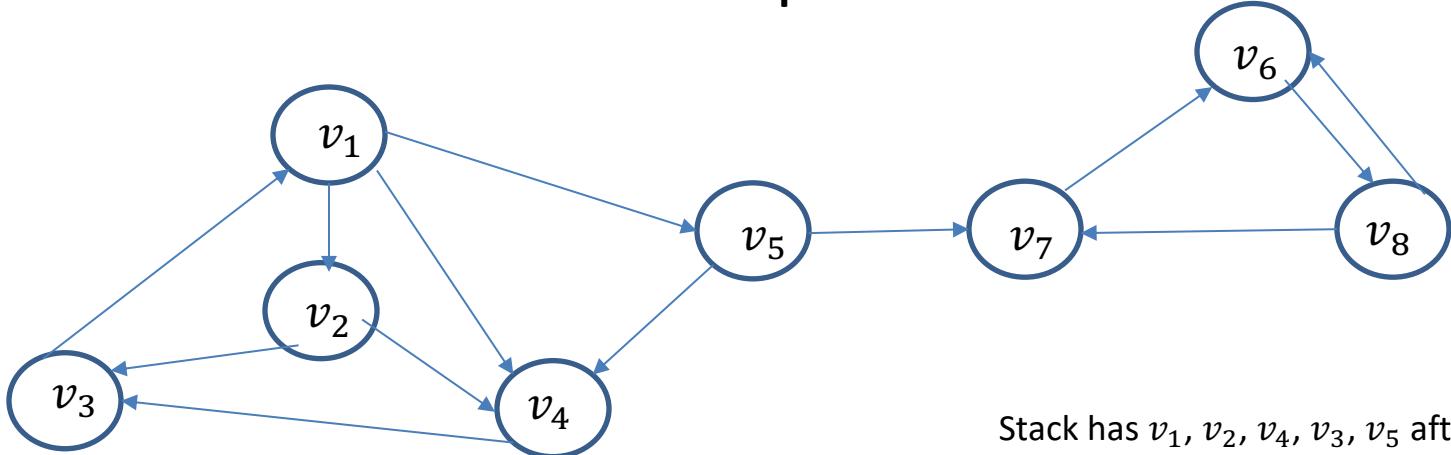
 if w = v

 break

 print “end of a strongly connected component”

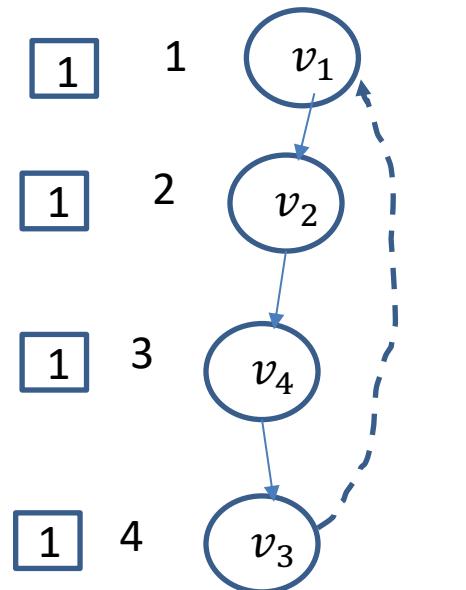
Return count

Example

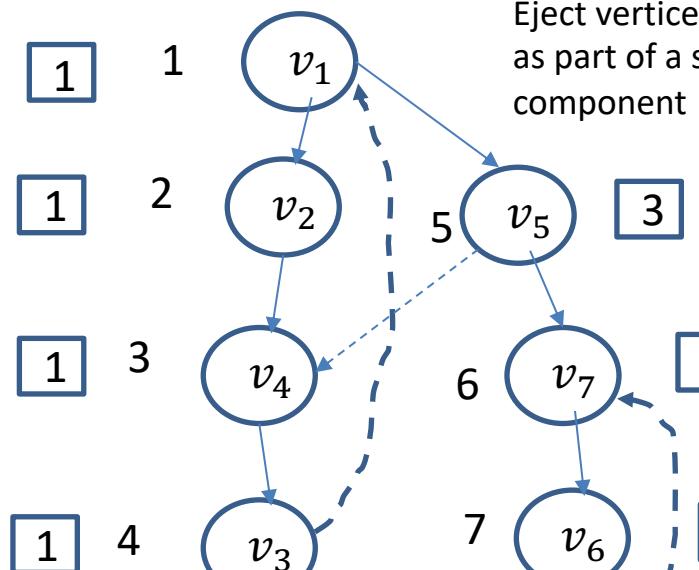


DF

LOWLINK (in square)



Stack(not recursion stack) has v_1, v_2, v_4, v_3 (v_3 on top) after $\text{DFS}(v_2)$ is completed

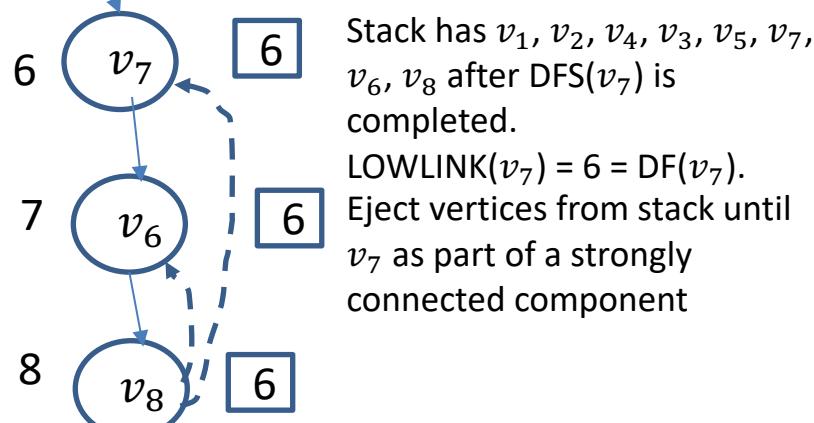


Cross edge (v_5, v_4) used as v_4 still in stack

Stack has v_1, v_2, v_4, v_3, v_5 after $\text{DFS}(v_1)$ is completed.

$$\text{LOWLINK}(v_1) = 1 = \text{DF}(v_1).$$

Eject vertices from stack until v_1 as part of a strongly connected component



Stack has $v_1, v_2, v_4, v_3, v_5, v_7, v_6, v_8$ after $\text{DFS}(v_7)$ is completed.

$$\text{LOWLINK}(v_7) = 6 = \text{DF}(v_7).$$

Eject vertices from stack until v_7 as part of a strongly connected component

Minimum Edit Distance - DP example

CS 435 Spring 2021
Instructor : Ravi Varadarajan

February 11, 2021

We need to transform the string $X = x_1x_2\dots x_m$ into the string $Y = (y_1y_2\dots y_n)$. Let's keep a pair of cursors one for X and another for Y which indicate the transformation remaining to be done. For example when the cursor pair is (i, j) , it remains to transform $x_i\dots x_m$ into $y_j\dots y_n$. This cursor pair defines a state and we move from state to state as we make decisions to keep a character, insert a character from Y , delete a character from X or replace a character in X from Y in these cursor positions.

Let us define $F(i, j)$ as the minimum number of operations required to transform $x_i\dots x_m$ into $y_j\dots y_n$. i.e. it is optimization function for a subproblem when we are in state (i, j) . So what we require ultimately is $F(1, 1)$. Let us focus on the subproblem defined by the state (i, j) .

(a) When $x_i = y_j$, then the optimum number of operations required to transform $x_i\dots x_m$ into $y_j\dots y_n$ should be the same as the optimum number of operations required to transform $x_{i+1}\dots x_m$ into $y_{j+1}\dots y_n$; in this case we move to cursor pair state $(i + 1, j + 1)$. So $F(i, j) = F(i + 1, j + 1)$.

(b) When $x_i \neq y_j$, we have 3 choices :

(i) insert y_j before x_i in which case we move to state $(i, j + 1)$ as what remains is to transform $x_i\dots x_m$ into $y_{j+1}\dots y_n$; here the cost will be 1 unit for insert + minimum number of operations required to transform $x_i\dots x_m$ into $y_{j+1}\dots y_n$, i.e. $1 + F(i, j + 1)$,

(ii) delete x_i in which case we move to state $(i + 1, j)$ as what remains is to transform $x_{i+1}\dots x_m$ into $y_j\dots y_n$; cost here is $1 + F(i + 1, j)$,

(iii) replace x_i by y_j in which case we move to state $(i + 1, j + 1)$; here cost is $1 + F(i + 1, j + 1)$. Obviously we like to take the choice that gives the minimum. This means that $F(i, j) = \min(1 + F(i + 1, j), 1 + F(i, j + 1), 1 + F(i + 1, j + 1))$.

This is the recursive formulation of DP. But we do not compute it recursively but iteratively in an order which guarantees that when we compute $F(i, j)$, we have already computed $F(i + 1, j), F(i, j + 1), F(i + 1, j + 1)$. What order guarantees that ?

We start from the following boundary conditions first :

(a) For $1 \leq i \leq m$, $F(i, n + 1)$ i.e. for the problem of transforming $x_i\dots x_m$ into empty string as $n + 1$ -th cursor position in Y moves past the end of the string. It is easy to see $F(i, n + 1) = m - i + 1$ as we just have to delete these $m - i + 1$ characters from X .

(b) similarly for $1 \leq j \leq n$, $F(m + 1, j)$ i.e. for the problem of transforming empty string into $y_j\dots y_n$ as $m + 1$ -th cursor position in X moves past the end of the string. It is easy to see $F(m + 1, j) = n - j + 1$ as we just have to insert these $n - j + 1$ characters of Y into X .

(c) $F(m+1, n+1) = 0$ as all transformation is done here.

Let's have two $(m+1) \times (n+1)$ tables, one to store $F(i, j)$, $1 \leq i \leq m+1, 1 \leq j \leq n+1$ and another to store $A(i, j)$ for optimum action in that state which has one of the values O, I, D, R corresponding to the four actions of keeping x_i when $x_i = y_j$, inserting y_j , deleting x_i and replacing x_i with y_j

We fill first the bottom row and right most column for boundary conditions, then we fill the rows from bottom to top and from right to left in each row. Final entry to be filled is $F(1, 1)$. The algorithm steps are follows:

```

for  $i \leftarrow 1$  to  $m+1$  do
|    $F[i, n+1] \leftarrow m - i + 1$ 
|   end
|   for  $j \leftarrow 1$  to  $n+1$  do
|   |    $F[m+1, j] \leftarrow n - j + 1$ 
|   |   end
|   for  $i \leftarrow m$  downto 1 do
|   |   for  $j \leftarrow n$  downto 1 do
|   |   |   if  $X.elementAtRank(i) = Y.elementAtRank(j)$  then
|   |   |   |    $(F[i, j], A[i, j]) \leftarrow (F[i+1, j+1], O')$ 
|   |   |   |   end
|   |   |   else
|   |   |   |    $F[i, j] = \min(1 + F[i+1, j], 1 + F[i, j+1], 1 + F[i+1, j+1])$ 
|   |   |   if  $F[i, j] = 1 + F[i+1, j]$  then
|   |   |   |    $A[i, j] = D'$ 
|   |   |   |   end
|   |   |   |   if  $F[i, j] = 1 + F[i, j+1]$  then
|   |   |   |   |    $A[i, j] = I'$ 
|   |   |   |   |   end
|   |   |   |   if  $F[i, j] = 1 + F[i+1, j+1]$  then
|   |   |   |   |    $A[i, j] = R'$ 
|   |   |   |   |   end
|   |   |   |   end
|   |   end
|   end
|   // get optimum solution
|   print 'minimum edit distance = ',  $F[1, 1]$ 
|    $(i, j) \leftarrow (1, 1)$ 

```

```

while  $i \leq m$  and  $j \leq n$  do
    if  $A[i, j] = 'O'$  then
        print 'Keep ', X.elementAtRank(i)
         $(i, j) \leftarrow (i + 1, j + 1)$ 
    end if
    if  $A[i, j] = 'I'$  then
        print 'Insert ', Y.elementAtRank(j)
         $(i, j) \leftarrow (i, j + 1)$ 
    end if
    if  $A[i, j] = 'D'$  then
        print 'Delete ', X.elementAtRank(i)
         $(i, j) \leftarrow (i + 1, j)$ 
    end if
    if  $A[i, j] = 'R'$  then
        print 'Replace ', X.elementAtRank(i), ' by ', Y.elementAtRank(j)
         $(i, j) \leftarrow (i + 1, j + 1)$ 
    end if
end while
while  $i \leq m$  do
    print 'Delete ', , X.elementAtRank(i)
     $i \leftarrow i + 1$ 
end while
while  $j \leq n$  do
    print 'Insert ', , Y.elementAtRank(j)
     $j \leftarrow j + 1$ 
end while

```

Try to compute these tables for X='AMAZING' and Y='HORIZONS' and get optimal solution.

Note optimal actions are indicated by \textcircled{O} - keep symbol in X, \textcircled{I} - insert symbol from Y in X, \textcircled{D} - delete symbol in X, \textcircled{R} - replace symbol in X by symbol in Y. Table after filling in boundary conditions:

	$j \rightarrow$	H	O	R	I	Z	O	N	S	
	$i \downarrow$	1	2	3	4	5	6	7	8	9
A	1									7 \textcircled{D}
M	2									6 \textcircled{D}
A	3									5 \textcircled{D}
Z	4									4 \textcircled{D}
I	5									3 \textcircled{D}
N	6									2 \textcircled{D}
G	7									1 \textcircled{D}
	8	8 \textcircled{I}	7 \textcircled{I}	6 \textcircled{I}	5 \textcircled{I}	4 \textcircled{I}	3 \textcircled{I}	2 \textcircled{I}	1 \textcircled{I}	0

Table after filling in the row $i = 7$ from right left:

	$j \rightarrow$	H	O	R	I	Z	O	N	S	
	$i \downarrow$	1	2	3	4	5	6	7	8	9
A	1									7 (D)
M	2									6 (D)
A	3									5 (D)
Z	4									4 (D)
I	5									3 (D)
N	6									2 (D)
G	7	8 (R)	7 (R)	6 (R)	5 (R)	4 (R)	3 (R)	2 (I)	1 (R)	1 (D)
	8	8 (I)	7 (I)	6 (I)	5 (I)	4 (I)	3 (I)	2 (I)	1 (I)	0

For example entry for the cell $(7, 8)$, namely for the subproblem of transforming 'G' to 'S' is given by $F(7, 8) = \min(1 + F(8, 9), 1 + F(7, 9), 1 + F(8, 8)) = 1 + F(8, 9) = 1$. The corresponding action here is replace 'G' by 'S' and we will be done with transformation as we move to state $(8, 9)$.

On the other hand for the cell $(7, 7)$, for the subproblem of transforming 'G' to 'NS', $F(7, 7) = \min(1 + F(8, 8), 1 + F(7, 8), 1 + F(8, 7))$ and both $1 + F(8, 8)$ and $1 + F(7, 8)$ give the same min value of 2 and we arbitrarily pick $1 + F(7, 8)$ with the corresponding action of inserting 'N' before 'G' and then move state to $(7, 8)$.

Table after filling in the row $i = 6$ from right left:

	$j \rightarrow$	H	O	R	I	Z	O	N	S	
	$i \downarrow$	1	2	3	4	5	6	7	8	9
A	1									7 (D)
M	2									6 (D)
A	3									5 (D)
Z	4									4 (D)
I	5									3 (D)
N	6	7 (I)	6 (I)	5 (I)	4 (I)	3 (I)	2 (I)	1 (O)	2 (R)	2 (D)
G	7	8 (R)	7 (R)	6 (R)	5 (R)	4 (R)	3 (R)	2 (I)	1 (R)	1 (D)
	8	8 (I)	7 (I)	6 (I)	5 (I)	4 (I)	3 (I)	2 (I)	1 (I)	0

For example for the cell $(6, 7)$, the subproblem is transforming 'NG' to 'NS' and since the symbols are same at the cursor positions, $F(6, 7) = F(7, 8) = 1$ and we move to state $(7, 8)$.

Final table values are given below:

	$j \rightarrow$	H	O	R	I	Z	O	N	S	
	$i \downarrow$	1	2	3	4	5	6	7	8	9
A	1	6 (R)	5 (R)	5 (R)	5 (D)	6 (R)	6 (R)	6 (D)	7 (R)	7 (D)
M	2	6 (R)	5 (R)	4 (R)	4 (R)	5 (R)	5 (R)	5 (D)	6 (R)	6 (D)
A	3	6 (R)	5 (R)	4 (R)	3 (R)	3 (D)	4 (R)	4 (D)	5 (R)	5 (D)
Z	4	6 (R)	5 (R)	4 (R)	3 (I)	2 (O)	3 (R)	3 (D)	4 (R)	4 (D)
I	5	6 (I)	5 (I)	4 (I)	3 (O)	3 (R)	2 (R)	2 (D)	3 (R)	3 (D)
N	6	7 (I)	6 (I)	5 (I)	4 (I)	3 (I)	2 (I)	1 (O)	2 (R)	2 (D)
G	7	8 (R)	7 (R)	6 (R)	5 (R)	4 (R)	3 (R)	2 (I)	1 (R)	1 (D)
	8	8 (I)	7 (I)	6 (I)	5 (I)	4 (I)	3 (I)	2 (I)	1 (I)	0

Optimal value is given by $F(1, 1) = 6$. We can get the optimal sequence of actions by starting from (1, 1) and using optimal action in the cell to move to next state. The optimal sequence of states : (1, 1) \rightarrow (2, 2) \rightarrow (3, 3) \rightarrow (4, 4) \rightarrow (4, 5) \rightarrow (5, 6) \rightarrow (6, 7) \rightarrow (7, 8) \rightarrow (8, 9).

The corresponding sequence of actions : (R) (replace 'A' by 'H') \rightarrow (R) (replace 'M' by 'O') \rightarrow (R) (replace 'A' by 'R') \rightarrow (I) (insert 'I') \rightarrow (O) (keep 'Z') \rightarrow (R) (replace 'I' by 'O') \rightarrow (O) (keep 'N') \rightarrow (R) (replace 'G' by 'S')

Numerical Problems

Ring Algebra

- $(S, +, \circ, 0, 1)$ is a **ring** if
 - (a) $(S, +, 0)$ is a monoid
 - (b) $(S, \circ, 1)$ is a monoid
 - (c) $+$ is commutative ($a + b = b + a$)
 - (d) \circ distributes over $+$ ($a \circ (b+c) = (a \circ b) + (a \circ c)$)
 - (e) every element a in S has additive inverse b , i.e. $a + b = b + a = 0$ (we call it $-a$)
- If operation \circ is also commutative, it is a **commutative ring**

Examples of Rings

- Set of real numbers with + and * operations (commutative ring)
- $(C, +, *, 0, 1)$ where C is set of complex numbers, + and * are complex number addition and multiplication (commutative ring)
- $(\{0,1,2,\dots,m-1\}, +_m, *_m, 0, 1)$ where $+_m$ and $*_m$ are addition and multiplication modulo m (commutative ring)
- $(M_n, +, *, 0_n, I_n)$ where M_n is the set of all $n \times n$ matrices with elements from a ring, + , * are matrix addition and multiplication and 0_n is zero matrix and I_n is identity matrix

Fourier Transforms

- Fourier transform is used in signal processing to convert a signal from time domain to frequency domain.
- Discrete Fourier Transform (DFT) is used in digital signal processing to examine frequency spectrum of a sampled signal (e.g. audio) over a period of time
- Let $(S, +, \circ, 0, 1)$ be a commutative ring. An element ω in S is called a principal n -th root of unity if

- (a) $\omega \neq 1$
- (b) $\omega^n = 1$ and
- (c) $\sum_{j=0}^{n-1} \omega^{jp} = 0, 1 \leq p < n$

Root of equation $x^n - 1 = (x - 1)(1 + x^2 + x^3 + \dots + x^{n-1}) = 0$
which is not 1.

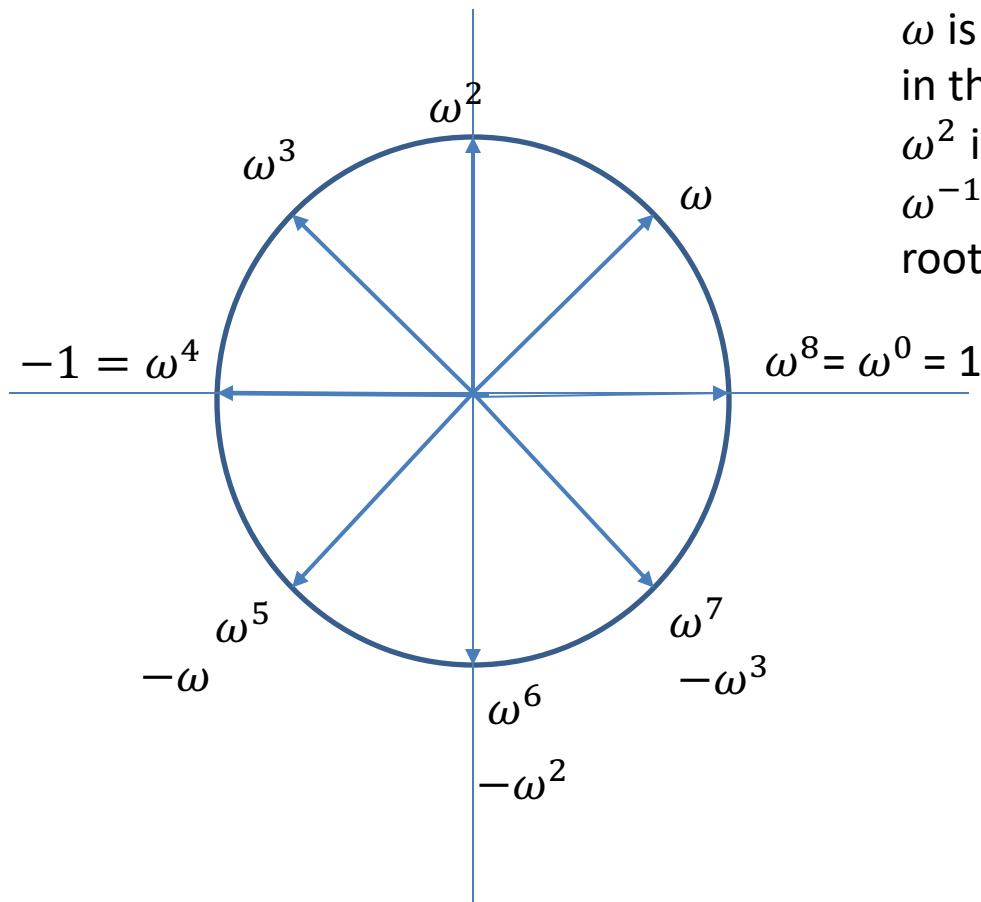
- $\omega^0, \omega^1, \dots, \omega^{n-1}$ are roots of unity, all except 1 are principal roots.

DFT

- In complex number ring $(C, +, *, 0, 1)$, a principal n-th root of unity is given by $e^{2\pi i/n}$ where $i = \sqrt{-1} = \cos 2\pi i/n + \sin 2\pi i/n$
- In $(\{0,1,2,\dots,16\}, +, *, 0, 1)$ where $+, *$ are modulo 17, 2 is the 8-th root of unity
- **DFT definition:**

Let $a = [a_0, a_1, \dots, a_{n-1}]$ be a n-element column vector
A is a $n \times n$ matrix with $A(i,j) = \omega^{ij}$ where ω is principal n-th root of unity

- $F(a) = A * a$ is a n-element transform vector
- Inverse A^{-1} exists, i.e. $A^{-1}(F(a)) = a$
 - If n has multiplicative inverse, $A^{-1}(i,j) = n^{-1} \omega^{-ij}$. Note ω^{-1} is also a principal n-th root of unity.



ω is the 8-th root of unity in the ring
 ω^2 is the 4-th root of unity
 $\omega^{-1} = \omega^7$ is also the 8-th root of unity.

In Complex number ring , $\omega = e^{\frac{2\pi i}{8}} = \cos \frac{2\pi}{8} + i \sin \frac{2\pi}{8}$

$$\omega^{-1} = \cos \frac{2\pi}{8} - i \sin \frac{2\pi}{8}$$

In modulo 17 ring $\{0,1,2\dots 16\}$, $\omega = 2$, $\omega^2 = 4$, $\omega^3 = 8$, $\omega^4 = 16 \equiv -1 \pmod{17}$,
 $\omega^5 = (-1) * 2 = -2 \equiv 15 \pmod{17}$, $\omega^6 = -4 \equiv 13 \pmod{17}$, $\omega^7 = -8 \equiv 9 \pmod{17}$
 $\omega^{-1} = \omega^7 = -8 \equiv 9 \pmod{17}$

DFT and polynomials

- Let $p(x)$ be the polynomial $= a_0 + a_1 x + \dots + a_{n-1} x^{n-1}$
- Then i-th element of $F(a)$ (DFT)
 $= \sum_{j=0}^{n-1} a_j \omega^{ij} = p(\omega^i)$ – evaluated at root ω^i
- $F(a)$ computes **evaluation** of $p(x)$ at roots of unity
- Inverse-DFT does polynomial **interpolation** (determining coefficients) from values at roots of unity
- Multiplication of polynomials $p(x) = a_0 + a_1 x + \dots + a_{n-1} x^{n-1}$ and $q(x) = b_0 + b_1 x + \dots + b_{n-1} x^{n-1}$ is given by
$$p(x) \cdot q(x) = c_0 + c_1 x + \dots + c_{2n-2} x^{2(n-1)}$$
where $c_k = \sum_{j=0}^k a_j b_{k-j}$
- $[c_0, c_1, \dots, c_{2n-2}]$ vector is called “convolution” of two vectors $[a_0, a_1, \dots, a_{n-1}]$ and $[b_0, b_1, \dots, b_{n-1}]$ commonly used in signal processing.

One way to do convolution of two n-1 element vectors is to

- (a) evaluate each of them at $2n$ -th roots of unity after padding each vector with 0's to length $2n$ – DFT of each vector
- (b) do pointwise multiplication of evaluated (DFT) values
- (c) Inverse-DFT of the result to get convolution vector

Numerical Problems (contd.)

DFT and polynomials

- Let $p(x)$ be the polynomial $= a_0 + a_1 x + \dots + a_{n-1} x^{n-1}$
- Then i-th element of $F(a)$ (DFT)
 $= \sum_{j=0}^{n-1} a_j \omega^{ij} = p(\omega^i)$ – evaluated at root ω^i
- $F(a)$ computes **evaluation** of $p(x)$ at roots of unity
- Inverse-DFT does polynomial **interpolation** (determining coefficients) from values at roots of unity
- Multiplication of polynomials $p(x) = a_0 + a_1 x + \dots + a_{n-1} x^{n-1}$ and $q(x) = b_0 + b_1 x + \dots + b_{n-1} x^{n-1}$ is given by
$$p(x) \cdot q(x) = c_0 + c_1 x + \dots + c_{2n-2} x^{2(n-1)}$$
where $c_k = \sum_{j=0}^k a_j b_{k-j}$
- $[c_0, c_1, \dots, c_{2n-2}]$ vector is called “convolution” of two vectors $[a_0, a_1, \dots, a_{n-1}]$ and $[b_0, b_1, \dots, b_{n-1}]$ commonly used in signal processing.

One way to do convolution of two n-1 element vectors is to

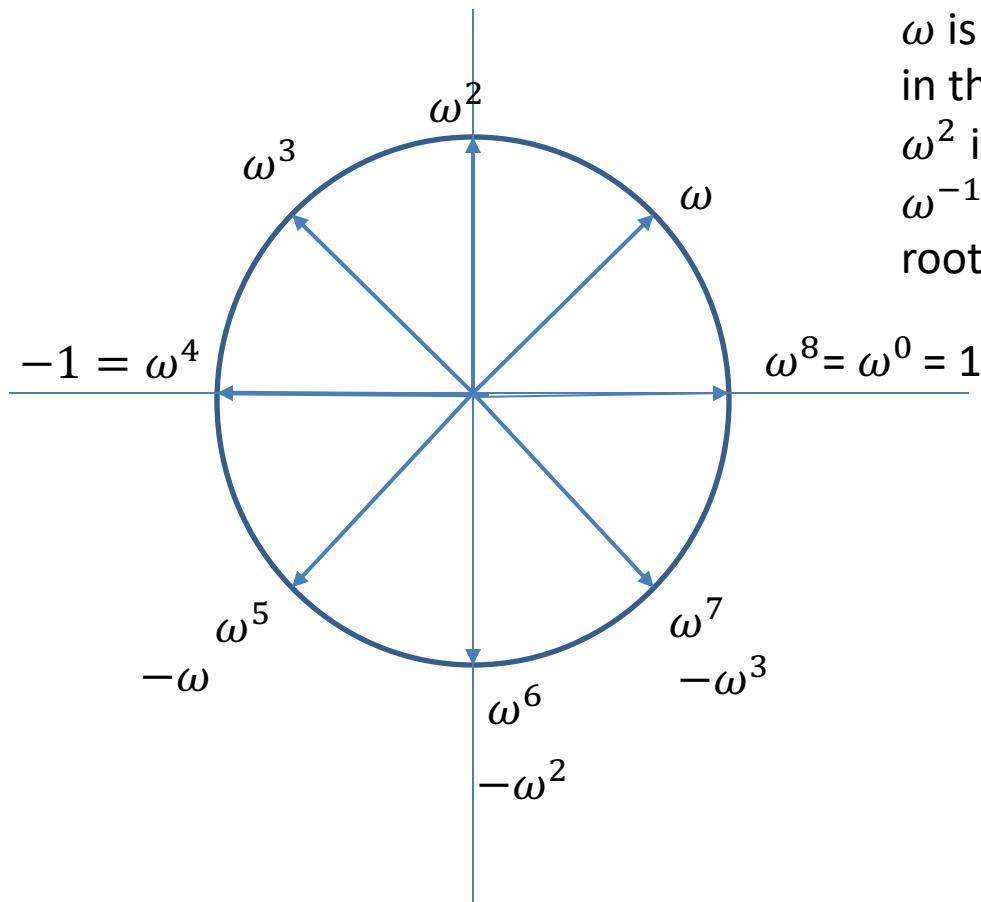
- (a) evaluate each of them at $2n$ -th roots of unity after padding each vector with 0's to length $2n$ – DFT of each vector
- (b) do pointwise multiplication of evaluated (DFT) values
- (c) Inverse-DFT of the result to get convolution vector

Fast Fourier Transform

- Evaluation of a $n-1$ degree polynomial at a point takes $O(n)$ time (ring operations) using Horner's rule:

$$p(x) = ((\dots (a_{n-1} x + a_{n-2}) x + a_{n-3}) x + \dots a_1) x + a_0$$

- If we evaluate $p(x)$ at each point $x = \omega^i$, $0 \leq i < n$ separately, DFT computation takes $O(n^2)$ time.
- FFT algorithm is much faster, takes $O(n \log n)$; requires n to be a power of 2. Relies on 2 results:
- Reduction property:** If ω is n -th root of unity when n is even, ω^2 is the $n/2$ -th root of unity in the same ring.
- Reflexive property :** If ω is n -th root of unity,
 $\omega^{n/2} = -1$ (additive inverse of 1 in the ring)



ω is the 8-th root of unity in the ring
 ω^2 is the 4-th root of unity
 $\omega^{-1} = \omega^7$ is also the 8-th root of unity.

In Complex number ring , $\omega = e^{\frac{2\pi i}{8}} = \cos \frac{2\pi}{8} + i \sin \frac{2\pi}{8}$

$$\omega^{-1} = \cos \frac{2\pi}{8} - i \sin \frac{2\pi}{8}$$

In modulo 17 ring $\{0,1,2\dots 16\}$, $\omega = 2$, $\omega^2 = 4$, $\omega^3 = 8$, $\omega^4 = 16 \equiv -1 \pmod{17}$,
 $\omega^5 = (-1) * 2 = -2 \equiv 15 \pmod{17}$, $\omega^6 = -4 \equiv 13 \pmod{17}$, $\omega^7 = -8 \equiv 9 \pmod{17}$
 $\omega^{-1} = \omega^7 = -8 \equiv 9 \pmod{17}$

Basis of FFT algorithm

- Suppose $p(x) = a_0 + a_1 x + \dots a_{n-1} x^{n-1}$ where $n = 2k$

Then $p(x)$ can be written as $p_{even}(x^2) + x \cdot p_{odd}(x^2)$

where

$$p_{even}(y) = a_0 + a_2 y + a_4 y^2 + \dots + a_{2(k-1)} y^{(k-1)}$$

$$p_{odd}(y) = a_1 + a_3 y + a_5 y^2 + \dots + a_{2(k-1)+1} y^{(k-1)}$$

- If $b = \text{FFT}([a_0, \dots, a_{n-1}])$, for $0 \leq i \leq k = n/2$,

$$\begin{aligned} b_i &= p(\omega^i) = p_{even}((\omega^2)^i) + \omega^i p_{odd}((\omega^2)^i) \\ &= c_i + \omega^i d_i \end{aligned}$$

where $c = \text{FFT}([a_0, a_2, \dots, a_{2(k-1)}])$, $d = \text{FFT}([a_1, a_3, \dots, a_{2(k-1)+1}])$ evaluated at $k=n/2$ -th root of unity ω^2

- Also note that $b_{k+i} = c_i + \omega^{k+i} d_i = c_i - \omega^i d_i$ as $\omega^k = \omega^{n/2} = -1$

Recursive FFT algorithm

FFT(a, ω):

Input : $a = [a_0, a_1, \dots, a_{n-1}]$ from a commutative ring, where $n=2^k$, $k \geq 0$ and ω is n-th root of unity in the ring

Output: $b = F.a$ is the FFT of a

if $n = 1$

 return a

$c \leftarrow \text{FFT}([a_0, a_2, \dots, a_{n-2}], \omega^2)$

$d \leftarrow \text{FFT}([a_1, a_3, \dots, a_{n-1}], \omega^2)$

$x \leftarrow \omega$

for $i \leftarrow 0$ to $n/2 - 1$

$b_i \leftarrow c_i + x * d_i$

$b_{\frac{n}{2}+i} \leftarrow c_i - x * d_i$

$x \leftarrow x * \omega$

return b

More on FFT

- Time complexity (primitive ring operations):
 $T(n) \leq 2 T(n/2) + k n$, $n \geq 2$, $T(1) = d \rightarrow T(n)$ is $O(n \log n)$
- Another way to look at FFT is division by polynomials
- For a polynomial $p(x)$, value at $x = a$
 $p(a)$ = remainder when $p(x)$ is divided by $x - a$ (remainder theorem)
- To compute values of $p(x)$ at $\omega^0, \omega^1, \dots, \omega^{n-1}$, we need remainders when $p(x)$ is divided by
 $x - \omega^0, x - \omega^1, \dots, x - \omega^{n-1}$
- To find out remainders of $p(x)$ when we divide by $q_1(x)$ and $q_2(x)$, we can first find remainder polynomial $r(x)$ of $p(x)$ when divided by $q_1(x) * q_2(x)$ and then
take remainders of $r(x)$ when divided by $q_1(x)$ and $q_2(x)$
- If we pair $x - \omega^0, x - \omega^1, \dots, x - \omega^{n-1}$ in a particular order and successively multiply products to get final product polynomial and then find remainders successively by the product polynomials we can find evaluations efficiently.

FFT Butterfly network

- Iterative bottom-up approach amenable to parallel processing
- For $n = 2^k$, there are k stages
- At each stage m , $0 \leq m < k$, we have 2^m smaller butterfly networks of size 2^{k-m}
- In a butterfly network at stage m , for each $0 \leq i \leq n - 1$, value at position i namely $v(i)$ paired with value at position j $v(j)$ where i and j differ in $(k - m)$ -th bit position
- Assuming $i < j$, value for next stage at $i = v(i) + \omega^{i_m} v(j)$ and value for next stage at $j = v(i) + \omega^{j_m} v(j)$
 i_m - integer resulting from reversing bits of integer i and shifting by $(k - m - 1)$ bits to the left

Numerical Problems (contd.)

Wrapped convolution

- Convolution of two n-length vectors results in 2n-length vector and hence FFT algorithm requires padding input vectors with zeros for the remaining n elements
- Wrapped convolution results in n-length vector and does not require padding with zeros.
- Positive wrapped convolution (useful in many integer algorithms):

$$c_i = \sum_{j=0}^i a_i b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}, \quad 0 \leq i \leq n-1$$

- Negative wrapped convolution:

$$d_i = \sum_{j=0}^i a_i b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}, \quad 0 \leq i \leq n-1$$

- Positive wrapped convolution **c** of **a** and **b** = $F^{-1}(F(\mathbf{a}) * F(\mathbf{b}))$
- Let $g(\mathbf{a}) = [a_0, \Psi a_1, \dots, \Psi^{n-1} a_{n-1}]$ where $\Psi^2 = \omega$. Then $g(\mathbf{d}) = F^{-1}(F(g(\mathbf{a})) * F(g(\mathbf{b})))$ where **d** is the negative wrapped convolution of **a** and **b**

Polynomial vs Integer arithmetic

- N-bit integer $u = \sum_{i=0}^{n-1} u_i 2^i$ is like a polynomial
- Integer product $u.v$ can be computed using convolution. In fact we do positive wrapped convolution by doing computation of uv modulo $2^n + 1$ using FFT.
- Schönhage-Strassen integer multiplication algorithm uses this approach to achieve time complexity of $O(n \log n \log \log n)$ bit operations.
- Polynomial division and squaring take same order of time as polynomial multiplication \rightarrow integer operations of same kind take same order of time as integer multiplication
- Finding $a \bmod b \leftrightarrow$ polynomial evaluation at a point
- Constructing integer from residues (Chinese remaindering) \leftrightarrow polynomial interpolation

Chinese Remainder Theorem

- Two integers a and b are relatively prime if $\gcd(a,b) = 1$
- Let p_0, p_1, \dots, p_{k-1} be k relatively prime numbers. Then any integer r between 0 and $p_0 * p_1 * \dots * p_{k-1} - 1$ can be represented as $(r_0, r_1, \dots, r_{k-1})$ (called “residues”) where $r_j = r \pmod{p_j}$. We write it as $r \leftrightarrow (r_0, r_1, \dots, r_{k-1})$
- This representation is unique due to Chinese Remainder Theorem
- **Given** : residues $(u_0, u_1, \dots, u_{k-1})$ w.r.t to p_0, p_1, \dots, p_{k-1}
Then $u = \sum_{i=0}^{k-1} c_i d_i u_i$ modulo p where
$$p = p_0 * p_1 * \dots * p_{k-1}$$
 and $c_i = p / p_i$, and $d_i = c_i^{-1} \pmod{p_i}$
- **Example:** $p_0 = 2, p_1 = 3, p_2 = 5$ and $p_3 = 7$ and $(u_0, u_1, u_2, u_3) = (1, 2, 4, 3)$. What is u such that $u \leftrightarrow (1, 2, 4, 3)$

Polynomial vs Integer arithmetic complexity

Problem	Integer (bit operations)	Polynomial (ring ops)
Addition/Subtraction	$O(n)$	$O(n)$
Multiplication	$O(n \log n \log \log n)$	$O(n \log n)$
Modulo $m = w^p + 1$ or modulo $x^t - c$	$O(p \log w)$	$O(t)$
Division	$O(n \log n \log \log n)$	$O(n \log n)$
Evaluation at n points		$O(n \log^2 n)$
Interpolation from n points		$O(n \log^2 n)$
Residues for “ k ” relatively prime numbers (n - bit)/polynomials (n - degree)	$O(nk \log k \log nk \log \log nk)$	$O(nk \log k \log nk)$
Chinese Remaindering (with preconditioning)	$O(nk \log k \log nk \log \log nk)$	$O(nk \log k \log nk)$
GCD	$O(n \log^2 n \log \log n)$	$O(n \log^2 n)$

Linear Recurrence Relations

Why need them ?

- Need for time and space complexity analysis
 - Natural for recursive algorithms but useful for non-recursive iterative algorithms also
 - Example: BinarySearch
- $T(n)$ – worst case complexity of while loop when $high - low = n$

$$T(0) = 1$$

$$T(n) = T(\lfloor n/2 \rfloor) + 8 \text{ for } n > 0$$

$S(n)$ – Space complexity is $O(n)$ and $O(1)$ for additional space

Complexity for recursion

factorial(n):

 Input : An integer $n \geq 0$

 Output: factorial of n

 if $n = 0$

 return 1

 else

 return $n * \text{factorial}(n-1)$

- Time complexity: $T(n) = 4 + T(n-1)$, $n > 0$ and $T(0) = 2$
- Space complexity $S(n) = 1 + S(n-1)$, $n > 0$ and $S(0) = 1$

Solving linear recurrences

- Expand and derive by inspection – use series summation
- Use characteristic equations – solve polynomial equation for general solution
- Use generating functions – $T(n) \rightarrow f(z)$ and convert recurrence to equation in $f(z)$, solve for $f(z)$, convert $f(z)$ solution back to $T(n)$ – Will not discuss it here.

Expansion & Inspection examples

$$1. \quad T(n) = T\left(\frac{n}{2}\right) + 1, n > 1 \text{ and } n \text{ a power of 2}$$
$$= 1, n = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + 1$$
$$= (T\left(\frac{n}{2^2}\right) + 1) + 1 = T\left(\frac{n}{2^2}\right) + 2$$
$$= (T\left(\frac{n}{2^3}\right) + 1) + 2 = T\left(\frac{n}{2^3}\right) + 3$$

Generalizing it by inspection,

$$T(n) = T\left(\frac{n}{2^i}\right) + i, i \geq 1$$

For what value of i , we can reach boundary condition, i.e. $n=1$

$$\frac{n}{2^i} = 1 \Rightarrow i = \log_2 n$$

Hence $T(n) = T(1) + \log_2 n = 1 + \log_2 n$. i.e. $T(n)$ is $\Theta(\log_2 n)$

Expansion & Inspection examples

2. $T(n) = 3 T\left(\frac{n}{2}\right) + c n, n > 1, n \text{ a power of } 2 \text{ and some } c > 0$
 $= k, n = 1 \text{ for some } k > 0$

$$\begin{aligned}T(n) &= 3 T\left(\frac{n}{2}\right) + cn \\&= 3 \left(3 T\left(\frac{n}{2^2}\right) + c \frac{n}{2}\right) + cn = 3^2 T\left(\frac{n}{2^2}\right) + cn \left(1 + \frac{3}{2}\right) \\&= 3^2 \left(3 T\left(\frac{n}{2^3}\right) + c \frac{n}{2^2}\right) + cn \left(1 + \frac{3}{2}\right) \\&= 3^3 T\left(\frac{n}{2^3}\right) + cn \left[1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2\right]\end{aligned}$$

Generalizing it by inspection for $i > 1$,

$$T(n) = 3^i T\left(\frac{n}{2^i}\right) + cn \sum_{j=0}^{i-1} \left(\frac{3}{2}\right)^j$$

Again setting $i = \log_2 n$,

$$\text{Hence } T(n) = 3^{\log_2 n} T(1) + cn \sum_{j=0}^{(\log_2 n-1)} \left(\frac{3}{2}\right)^j$$

Expansion & Inspection examples

Again setting $i = \log_2 n$,

$$\text{Hence } T(n) = 3^{\log_2 n} T(1) + cn \sum_{j=0}^{(\log_2 n-1)} \left(\frac{3}{2}\right)^j$$

Note that first term = $3^{\log_2 n}$ $k = n^{\log_2 3}$ k which is $\Theta(n^{\log_2 3})$

$$\text{Second term} = cn \sum_{j=0}^{(\log_2 n-1)} \left(\frac{3}{2}\right)^j$$

$$= cn \left[\frac{\left(\frac{3}{2}\right)^{\log_2 n-1}}{\frac{3}{2}-1} \right] \text{(using geometric series summation)}$$

$$\leq 2cn \left(\frac{3}{2}\right)^{\log_2 n} = c n^{\log_2 3} \text{ which is } O(n^{\log_2 3}), \text{ in fact it is } \Theta(n^{\log_2 3})$$

Hence $T(n)$ is $\Theta(n^{\log_2 3})$

Using characteristic equations

- Solving higher order (i.e. having more than one recurrence term on RHS) recurrences with expansion is difficult.
- Consider homogenous recurrence equation

$$T(n) = a_1 T(n - 1) + a_{m-1} T(n - 2) + \dots + a_m T(n - m)$$

We can show that $T(n) = b^n$ satisfies the above equation for some value of b .

$$b^n - a_1 b^{n-1} - a_2 b^{n-2} - \dots - a_m b^{n-m} = 0$$

$$b^{n-m}(b^m - a_1 b^{m-1} - a_2 b^{m-2} - \dots - a_m) = 0$$

$$\Rightarrow (b^m - a_1 b^{m-1} - a_2 b^{m-2} - \dots - a_m) = 0$$

There are m roots of the equation.

Simple case : all m roots b_1, b_2, \dots, b_m are distinct

General solution : $k_1 b_1^n + k_2 b_2^n + \dots + k_m b_m^n$

Use m boundary conditions to solve for constants k_1, \dots, k_m

Characteristic equations examples

1. Fibonacci sequence

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

$$F(1) = 1, F(0) = 0$$

$$F(n) - F(n - 1) - F(n - 2) = 0$$

Characteristic equation :

$$b^2 - b - 1 = 0 ; \text{ quadratic equation in } b$$

$$\text{Roots are : } b_1 = \frac{1+\sqrt{5}}{2} \text{ and } b_2 = \frac{1-\sqrt{5}}{2}$$

$$\text{General solution : } k_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + k_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$F(0) = k_1 + k_2 = 0$$

$$F(1) = k_1 \left(\frac{1+\sqrt{5}}{2}\right) + k_2 \left(\frac{1-\sqrt{5}}{2}\right) = 1$$

$$\text{Solving for } k_1, k_2 \text{ we get } k_1 = \frac{1}{\sqrt{5}} \text{ and } k_2 = -\frac{1}{\sqrt{5}}$$

$$\text{Final solution : } F(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Characteristic equations examples

- Equal roots of characteristic equation:

Example:

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

$$F(0) = 1 \text{ and } F(1) = d + 1$$

$$\Rightarrow F(n) - 2F(n - 1) + F(n - 2) = 0$$

$$\Rightarrow \text{Characteristic equation: } b^2 - 2b + 1 = 0$$

$$(b - 1)^2 = 0 . \text{ Equal roots } b = 1$$

$$\text{Here general solution : } F(n) = (k_1 n + k_2) b^n = (k_1 n + k_2)$$

$$F(0) = k_2 = 1$$

$$F(1) = (k_1 + k_2) = d + 1 \Rightarrow k_1 = d$$

$$\text{Final solution : } F(n) = n + d$$

Characteristic equation method for non-homogeneous recurrences

$$a_1 T(n) + a_2 T(n - 1) + a_{m-1} T(n - 2) + \dots + a_m T(n - m) = f(n)$$

- Two approaches
- 1. convert it into higher order homogeneous recurrence:

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

$$T(n) - T(n-1) = d$$

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

$$\text{Hence } T(n) - T(n-1) = T(n-1) - T(n-2)$$

$$T(n) - 2T(n-1) + T(n-2) = 0 \text{ (previous example)}$$

- 2. Solution : $T(n) = G(n) + H(n)$ (homogeneous + particular solution)

Transforming into linear recurrences

- $T(n) = 3 T\left(\frac{n}{2}\right) + c n, n > 1, n \text{ a power of } 2$
 $= k, n = 1$

Setting $n = 2^l$ and letting $S(l) = T(n)$

We get

$$S(l) = 3 S(l - 1) + c 2^l$$

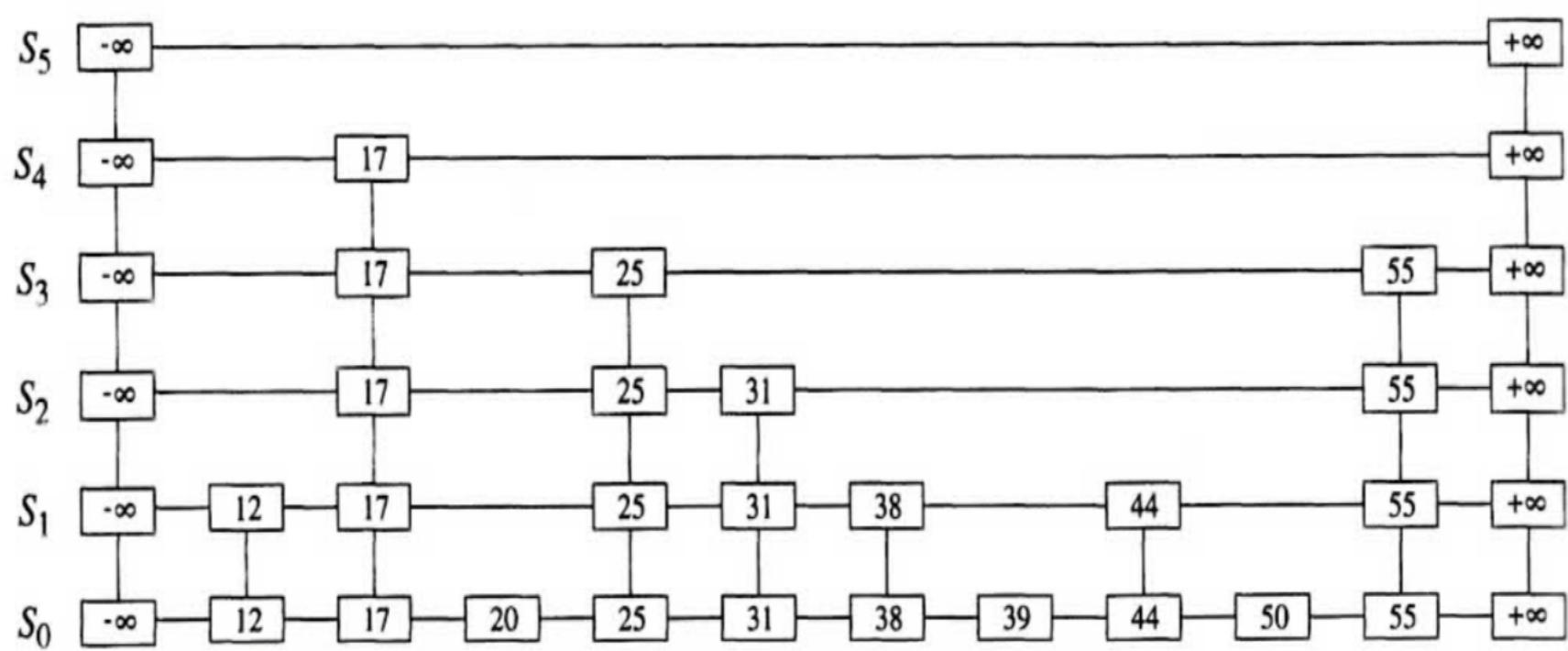
$$S(0) = k$$

Probabilistic data structures for dictionaries and sets

Ravi Varadarajan

Skip-lists

- A linked list of keys in non-decreasing order with short-cut pointers to speed-up search.
- Every key node in list has level_0 link to next key node (with keys in non-decreasing order)
- In addition a node may have level_1, level_2,...level_m links with the link always pointing to node with key no larger than this key.
- In effect there are m+1 lists
- Two categories of operations :
 - (a) before(p), after(p) – to navigate within level_i list
 - (b) below(p), above(p) – to navigate between levels.



Randomization in skip lists

- Each key has $\frac{1}{2}$ probability of being in next level list. If it is not in level__i list it will not be in any higher level list.
- There are n items in level__0 list, $n/2$ items on the average on level__1 list, $n/4$ items on average in level__2 list,..., $n/2^i$ on the average on level__i etc..
- Bounding # of levels or height:

Prob(an item is in level i) = $1/2^i$ (consecutive “ i ” heads in coin tosses)

$$\rightarrow \text{Prob}(\text{level } i \text{ is not empty}) = P_i \leq n / 2^i$$

$$\rightarrow \text{Prob}(\text{height} > c \log n \text{ for } c > 1) \leq n / 2^{c \log n} = 1 / n^{c-1}$$

$$\rightarrow \text{Prob}(\text{height} \leq c \log n) \geq 1 - 1/n^{c-1} \text{ (approaches 1 as } n \rightarrow \infty\text{)}$$

\rightarrow With high probability, height is $O(\log n)$.

- Space complexity : $E(\text{size}) = \sum_{i=0}^h n / 2^i < 2n$, hence $O(n)$

FindElement in skip list

findElement(k):

Input : key k

Output : item associated with k if it exists, else return “key error”

$p \leftarrow$ first position of highest level

while $\text{below}(p) \neq \text{null}$ and $\text{key}(p) \neq k$

$p \leftarrow \text{below}(p)$

 while $\text{key}(\text{after}(p)) \leq k$

$p \leftarrow \text{after}(p)$

 if $\text{key}(p) = k$ return $\text{item}(p)$ else return “key error”

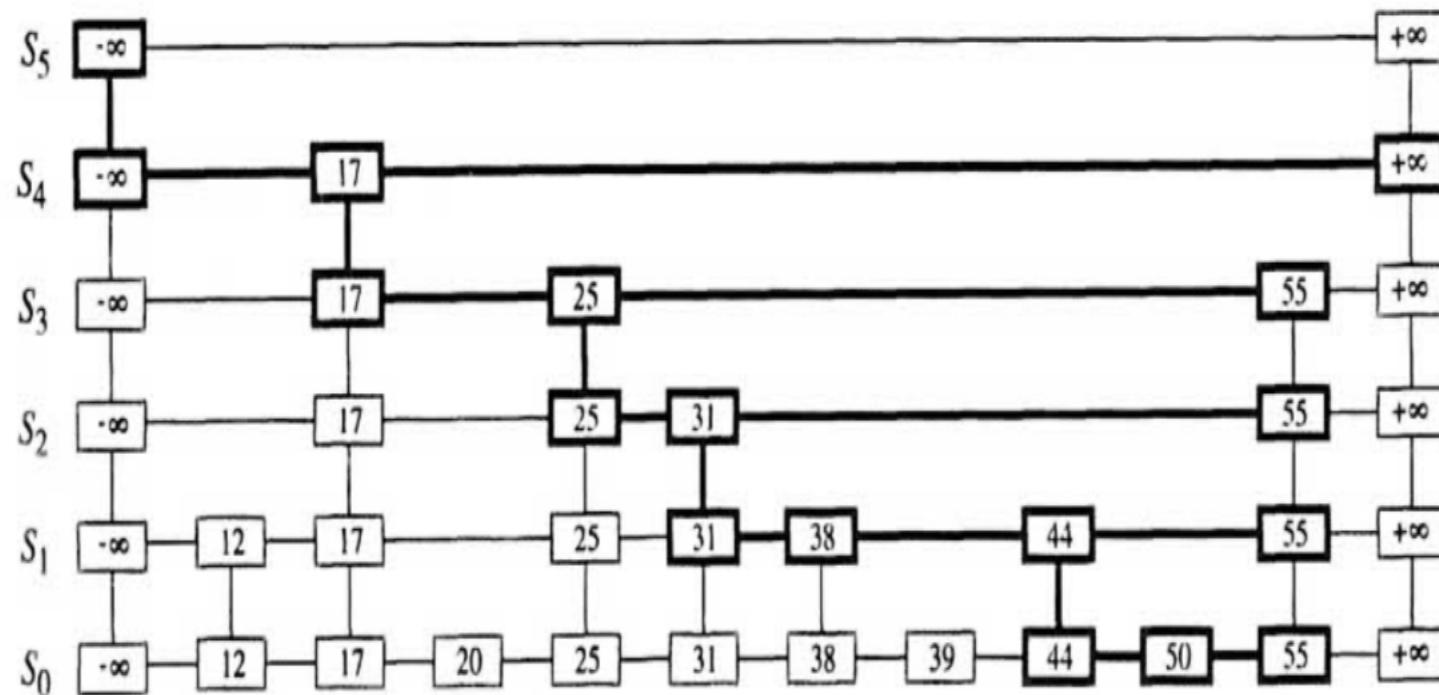
- **Time complexity :**

(a) # of outer loop iterations is $O(\log n)$ with high probability

(b) Expected # of inner loop iterations at level i = Expected number of items encountered in level i but not in level (i+1) = Expected number of coin tosses until a tail

$$\sum_{i=0}^{\infty} i (1/2^i) = 2$$

With high probability `findElement()` takes $O(\log n)$ time.



Insertion/Deletion in skip list

- InsertElement() first does a search all the way to level_0 list
- Flip a coin repeatedly to decide if it has to be in the next higher level list. If it is not in the next level we stop. Else insert it in the list
- Complexity is $O(\log n)$ expected
- DeleteElement() requires removing item from multiple levels and since height is $O(\log n)$ with high prob., complexity is $O(\log n)$ expected.

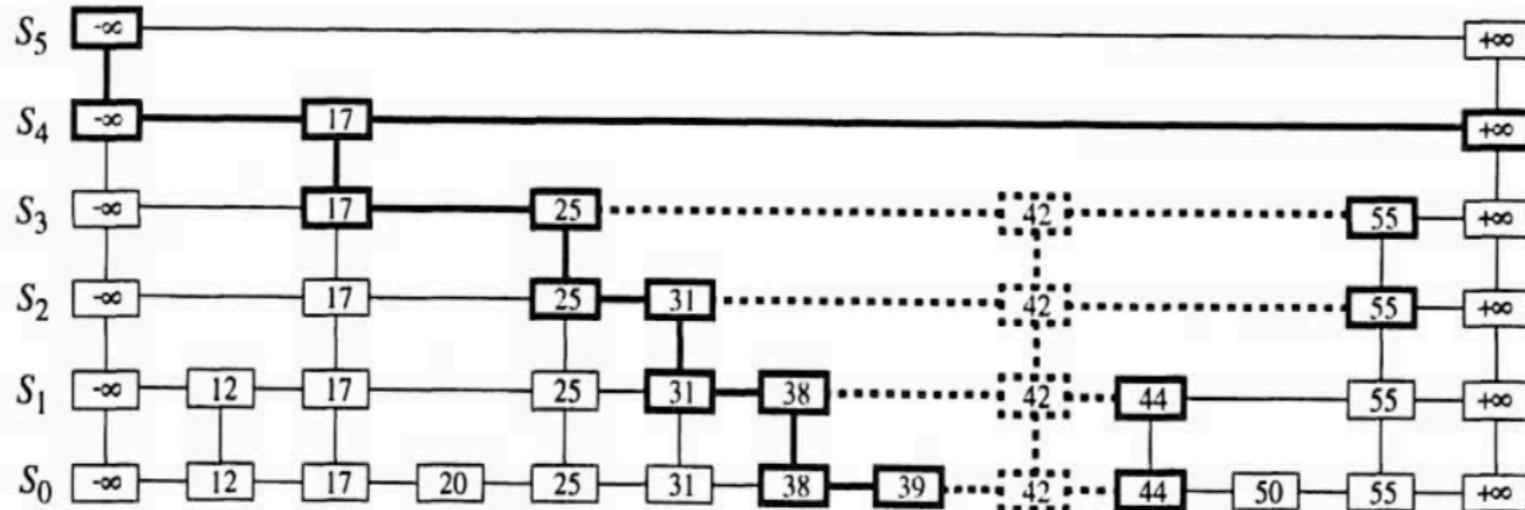


Figure 3.50: Insertion of an element with key 42 into the skip list of Figure 3.46. The positions visited and the links traversed are drawn with thick lines. The positions inserted to hold the new item are drawn with dashed lines.

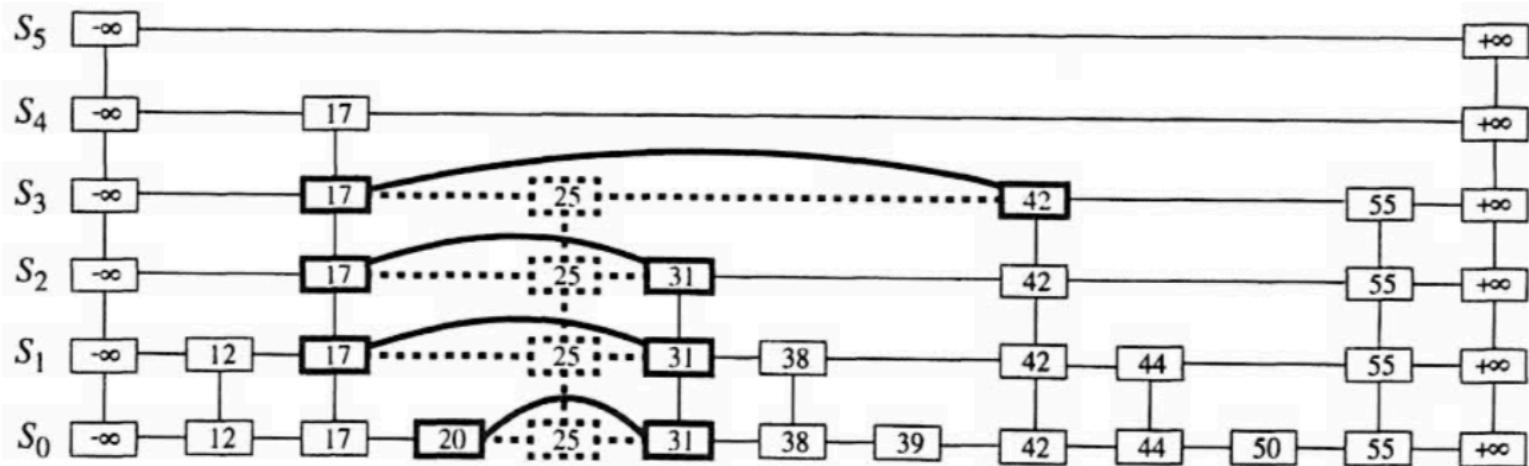


Figure 3.51: Removal of the item with key 25 from the skip list of Figure 3.50. The positions visited and the links traversed after the initial search are drawn with thick lines. The positions removed are drawn with dashed lines.

Sorting and selection

Introduction

4

- Sorting entails arranging data in order
- Familiarity with sorting algorithms is an important programming skill
- The study of sorting algorithms provides insight
 - ▣ into problem solving techniques such as *divide and conquer*
 - ▣ into the analysis and comparison of algorithms which perform the same task

Distribution based sorting

- **Stable sorting** – if for two elements a_i, a_j in input sequence, $i < j$ and $a_i = a_j$, in output sequence a_i should appear before a_j
- Allows sorting on multiple keys of elements easily.

- **Bucket-Sort** : Keys in range $[0, N-1]$ (e.g. state abbreviations)

N buckets, each key id added to list of corresponding bucket, bucket lists are concatenated. Can be made stable.

Time complexity is $O(n+N)$ where n is number of items to be sorted.

Radix sort algorithm

RadixSort(A):

Input : Sequence of m-component items k_1, k_2, \dots, k_n where k_i is a m-tuple $(k_{i1}, k_{i2}, \dots, k_{im})$, where $k_{ij} \in [0, N-1]$

Output : Lexicographically sorted sequence of the items

$I \rightarrow$ copy of input sequence k_1, k_2, \dots, k_n // output list

for $t \leftarrow 0$ to $N-1$

$Q[t] \leftarrow \{\}$ // bucket lists

for $j \leftarrow m$ down to 1 // each component

for $i \leftarrow 1$ to n

remove k_i from list I and add it to the end of the list $Q[k_{ij}]$

for $t \leftarrow 0$ to $N-1$

remove items from $Q[t]$ add them to end of list “ I ”

Return I

Time complexity : $O(m(n+N))$ - For fixed value if m and N , complexity is $O(n)$ best we can do.

Radix sort example

158, 241, 136, 222, 357, 782, 438

Bucket sort on least significant digit :

Keep 10 buckets one for each digit:

0 [] 1 [241] 2 [222, 782] 3[] 4 [] 5 [] 6 [136] 7 [357] 8 [158,438] 9 []

241, 222, 782, 136, 357, 158, 438

Sort on 2nd digit

0[] 1 [] 2 [222] 3 [136,438] 4 [241] 5 [357,158] 6 [] 7 [] 8 [782] 9 []

222, 136, 438, 241, 357, 158, 782

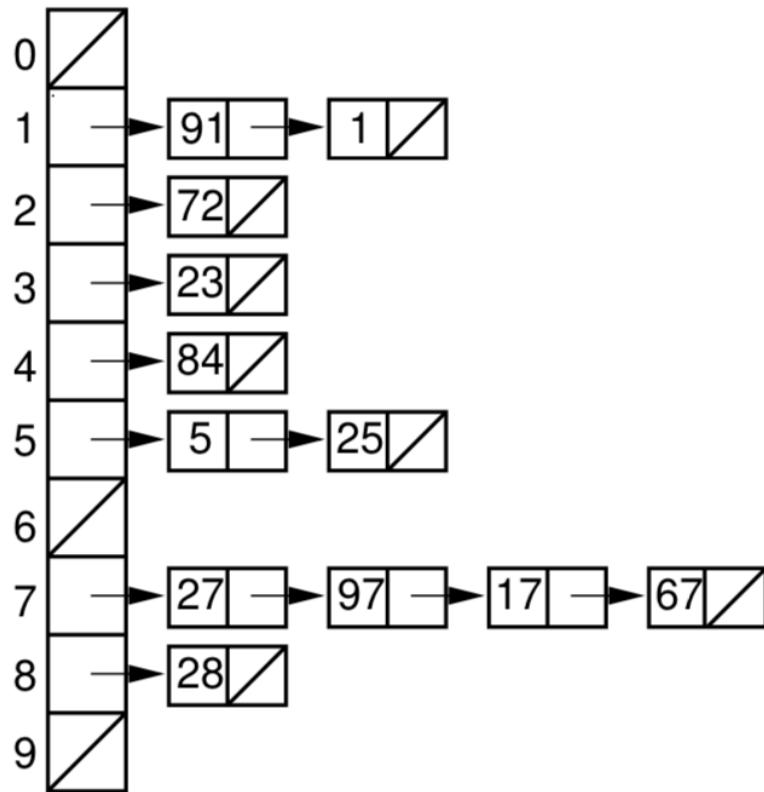
Sort most significant digit

0[] 1 [136,158] 2 [222,241] 3 [357] 4 [438] 5 [] 6 [] 7 [782] 8 [] 9 []

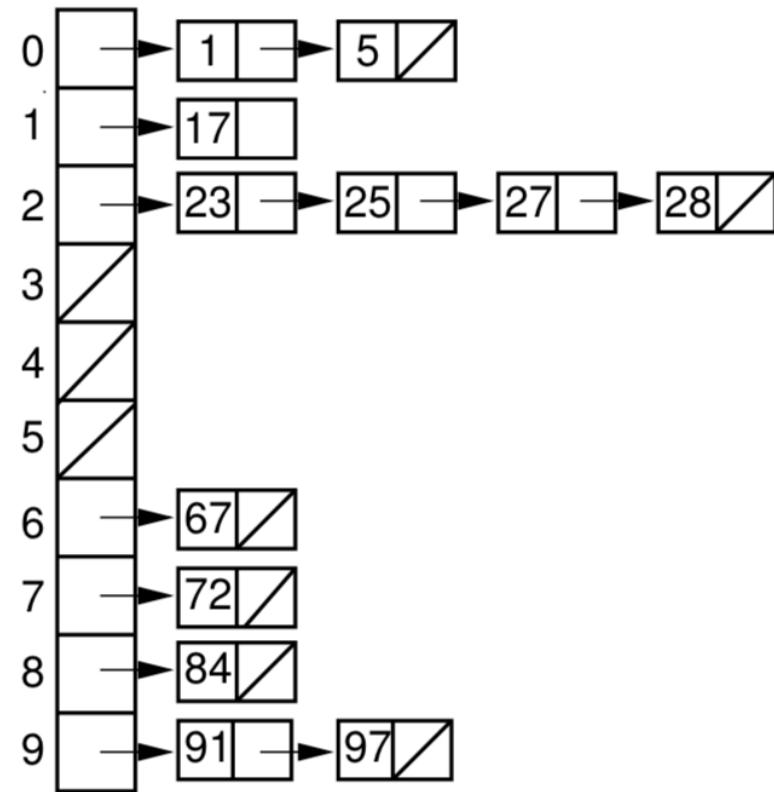
136, 158, 222, 241, 357, 438, 782

Radix Sort

First pass
(on right digit)



Second pass
(on left digit)



Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28

Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

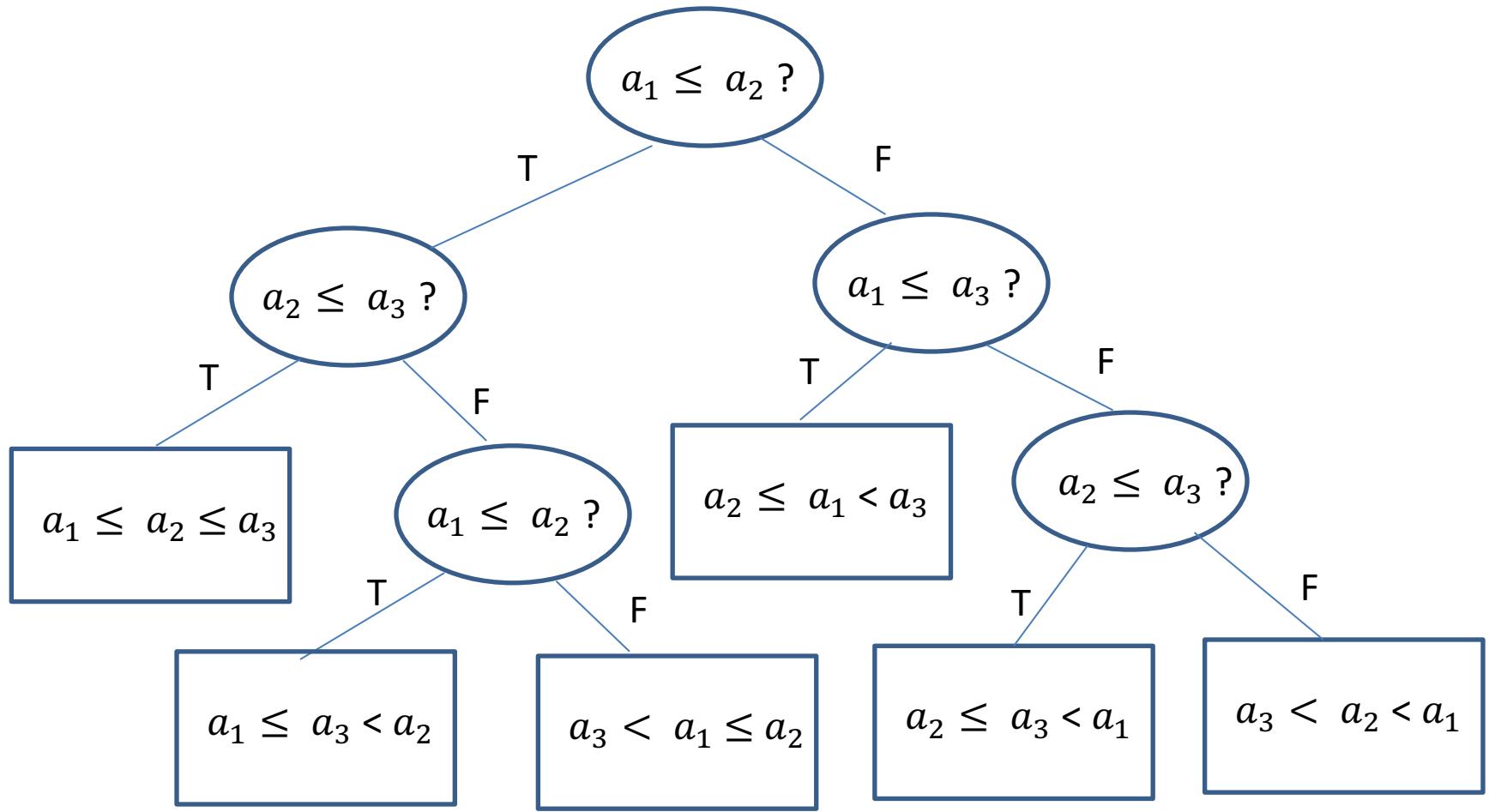
Figure 7.17 An example of Radix Sort for twelve two-digit numbers in base ten.

Two passes are required to sort the list.

III THE SAME BUCKET III THIS STEP.

Sorting lower bound

- How fast can we sort elements ?
- If we don't know the distribution or range of input elements a priori, we can derive lower bound based on comparisons only.
- Suppose we have input elements $a_1, a_2, a_3 \dots, a_n$
- Any comparison based algorithm can be reduced to a decision tree where in each node we check if $a_i \leq a_j$ for two elements a_i and a_j . Only 2 outcomes.
- Leaves of decision tree correspond to sorting order.
- Tree must have at least $n!$ leaves (permutations of n input elements) and must have height $\geq \lceil \log n! \rceil$
- Worst-case time complexity is $O(h)$ where h is height of decision tree
- By Stirling's formula: $n! \approx \left(\frac{n}{e}\right)^n \rightarrow$ any sorting algorithm must have # of comparisons $\geq n \log n - 1.44n$



Decision tree for a 3-element
sorting algorithm

Comparison based Sorting

- Criteria:
 - (a) # of comparisons
 - (b) # of data movements
 - (c) Additional space required
 - (d) in-memory vs external sorting
 - (e) stable sorting
- $O(n^2)$ sorting algorithms
 1. Bubble Sort (lot of comparisons + data movements),
 2. Selection Sort (lot of comparisons + less data movements),
 3. Insertion Sort (on average less comparisons + data movements); takes advantage of input ordering
- $O(n \log n)$ sorting algorithms
Heap sort, Merge Sort, Quick Sort

Comparison of $O(n^2)$ sorting methods

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Using Java Sorting Methods

5

- The Java API provides a class `Arrays` with several overloaded sort methods for different array types
- The Collections class provides similar sorting methods for Lists
- Sorting methods for arrays of primitive types are based on the quicksort algorithm
- Sorting methods for arrays of objects and Lists are based on the merge sort algorithm
- Both algorithms are $O(n \log n)$

Heap sort

HeapSort(A):

Input : Array A[1..n] of elements to be sorted

Output : Same array sorted

h \leftarrow Heapify(A) // use MAX_HEAP A

for i \leftarrow 0 to n-2

 e \leftarrow removeMax(A[n-i]) // consider heap A[1..n-i]

 A[n-i] \leftarrow e

Time complexity : O(n) for heap construction and O(log n) in each iteration of for loop

- Total time complexity : O(n log n) worst-case
- In-place sorting

Heap sort example

35	25	15	20	22	40	14	38
----	----	----	----	----	----	----	----

After heapify to
create max heap :

40	38	35	25	22	15	14	20
----	----	----	----	----	----	----	----

i=0; A[8]← removeMax(A[1..8])

38	25	35	20	22	15	14	40
----	----	----	----	----	----	----	----

i=1; A[7]← removeMax(A[1..7])

35	25	15	20	22	14	38	40
----	----	----	----	----	----	----	----

i=2; A[6]← removeMax(A[1..6])

25	22	15	20	14	35	38	40
----	----	----	----	----	----	----	----

i=3; A[5]← removeMax(A[1..5])

22	20	15	14	25	35	38	40
----	----	----	----	----	----	----	----

i=4; A[4]← removeMax(A[1..4])

20	14	15	22	25	35	38	40
----	----	----	----	----	----	----	----

i=5; A[3]← removeMax(A[1..3])

15	14	20	22	25	35	38	40
----	----	----	----	----	----	----	----

i=6; A[2]← removeMax(A[1..2])

14	15	20	22	25	35	38	40
----	----	----	----	----	----	----	----

Recursive Sort

RecursiveSort(S):

Input : Sequence S of n elements to be sorted

Output : Sorted sequence of S

if $|S| = 1$

 return S

else

$(S_1, S_2) \leftarrow \text{Split}(S)$ // split S into S_1 and S_2 subsequences of roughly equal size

RecursiveSort(S_1)

RecursiveSort(S_2)

 return **Join(S_1, S_2)** // join S_1 and S_2

- In MergeSort, join step takes $O(n)$ time
- In QuickSort, split step takes $O(n)$ time
- Recurrence : $T(n) \leq 2T(n/2) + k n \Rightarrow T(n)$ is $O(n \log n)$

MergeSort

Merge(S1,S2):

Input : Two sorted sequences S1 and S2

Output : A single sorted sequence of elements from S1 and S2

$i \leftarrow 0; j \leftarrow 0;$

$S \leftarrow \{\}$

while $i < \text{size}(S1)$ and $j < \text{size}(S2)$

 if $S1.\text{elementAtRank}(i) \leq S2.\text{elementAtRank}(j)$

$S.\text{append}(S1.\text{elementAtRank}(i)); i \leftarrow i+1$

 else

$S.\text{append}(S1.\text{elementAtRank}(j)); j \leftarrow j+1$

// append remaining elements

while ($i++ < \text{size}(S1)$)

$S.\text{append}(S1.\text{elementAtRank}(i))$

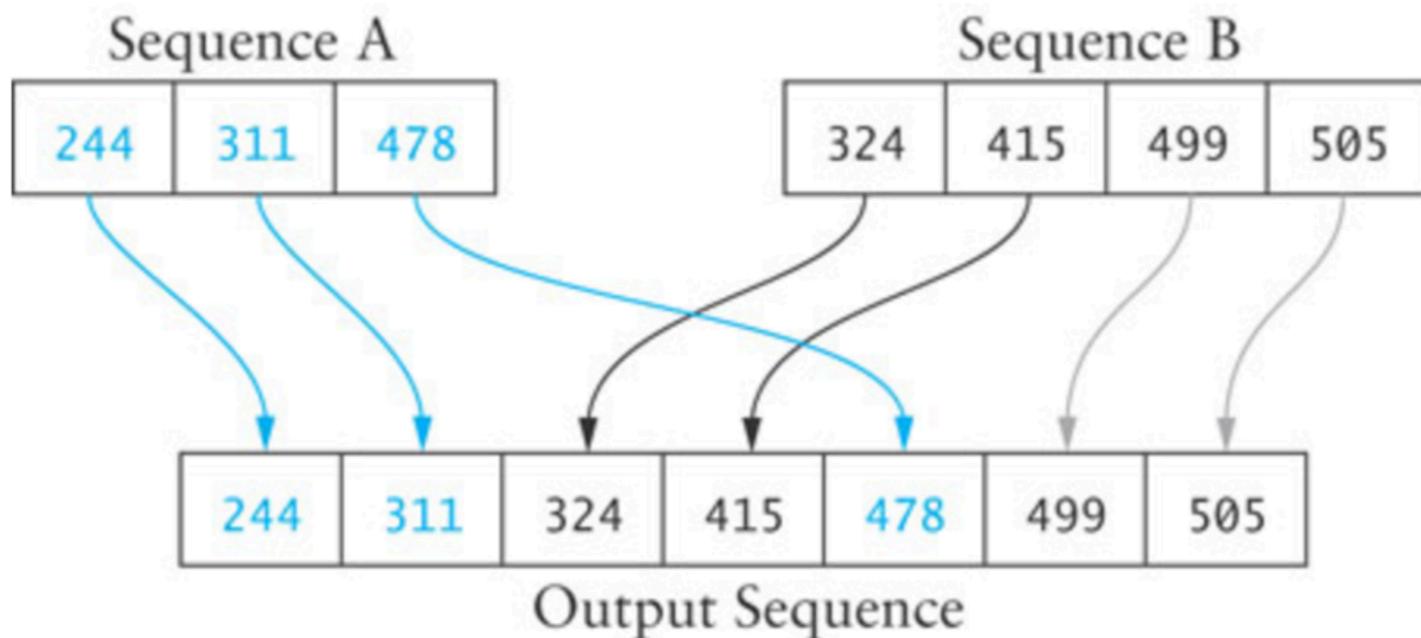
while ($j++ < \text{size}(S1)$)

$S.\text{append}(S2.\text{elementAtRank}(j))$

return S

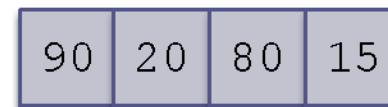
Time and additional space complexity is $O(m+n)$, $m = |S1|$ and $n = |S2|$

Merging example



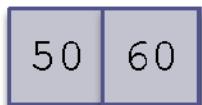
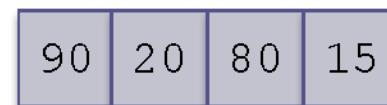
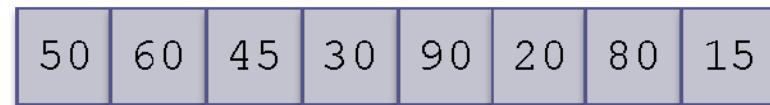
Trace of Merge Sort

137



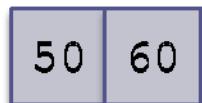
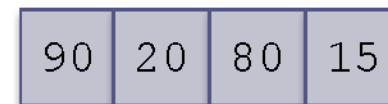
Trace of Merge Sort (cont.)

138



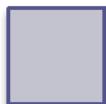
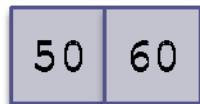
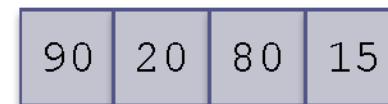
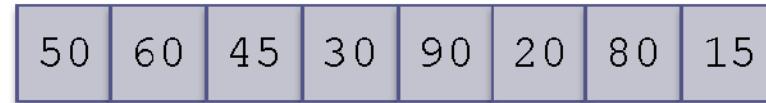
Trace of Merge Sort (cont.)

139



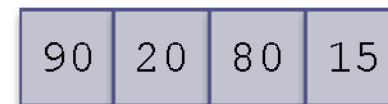
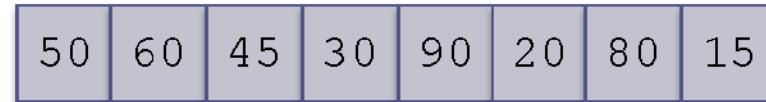
Trace of Merge Sort (cont.)

140



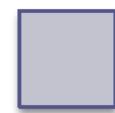
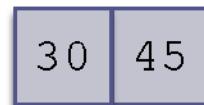
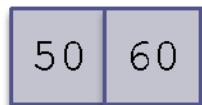
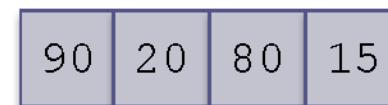
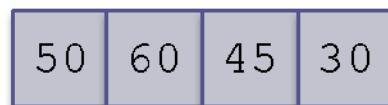
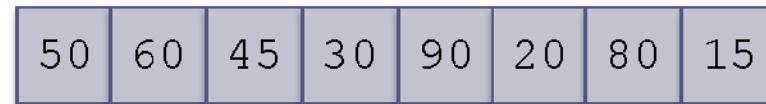
Trace of Merge Sort (cont.)

141



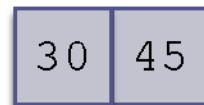
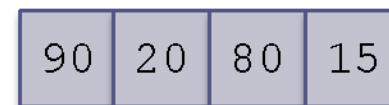
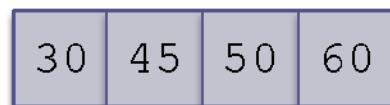
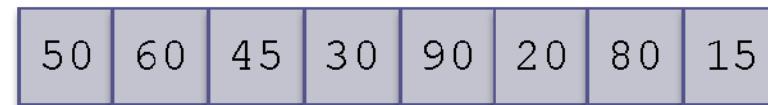
Trace of Merge Sort (cont.)

142



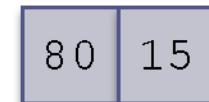
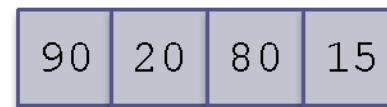
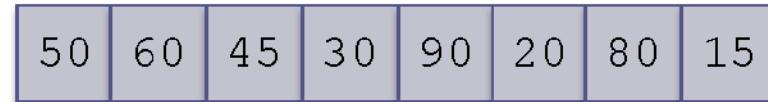
Trace of Merge Sort (cont.)

143



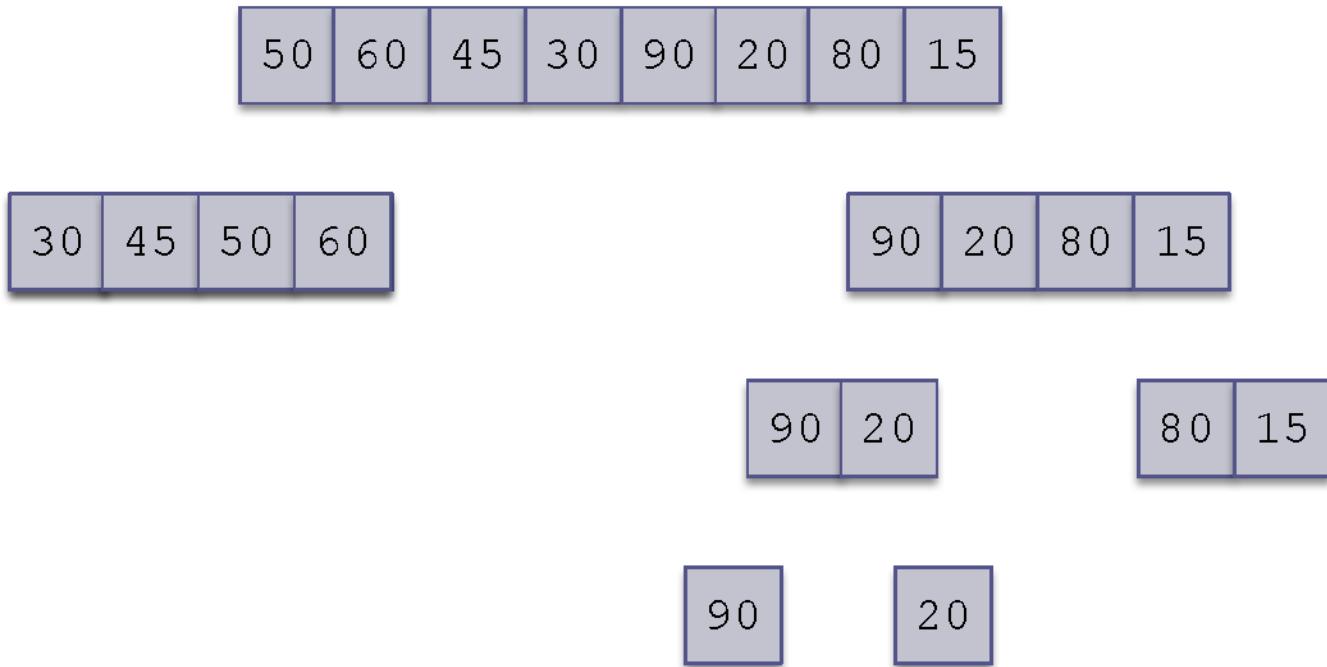
Trace of Merge Sort (cont.)

144



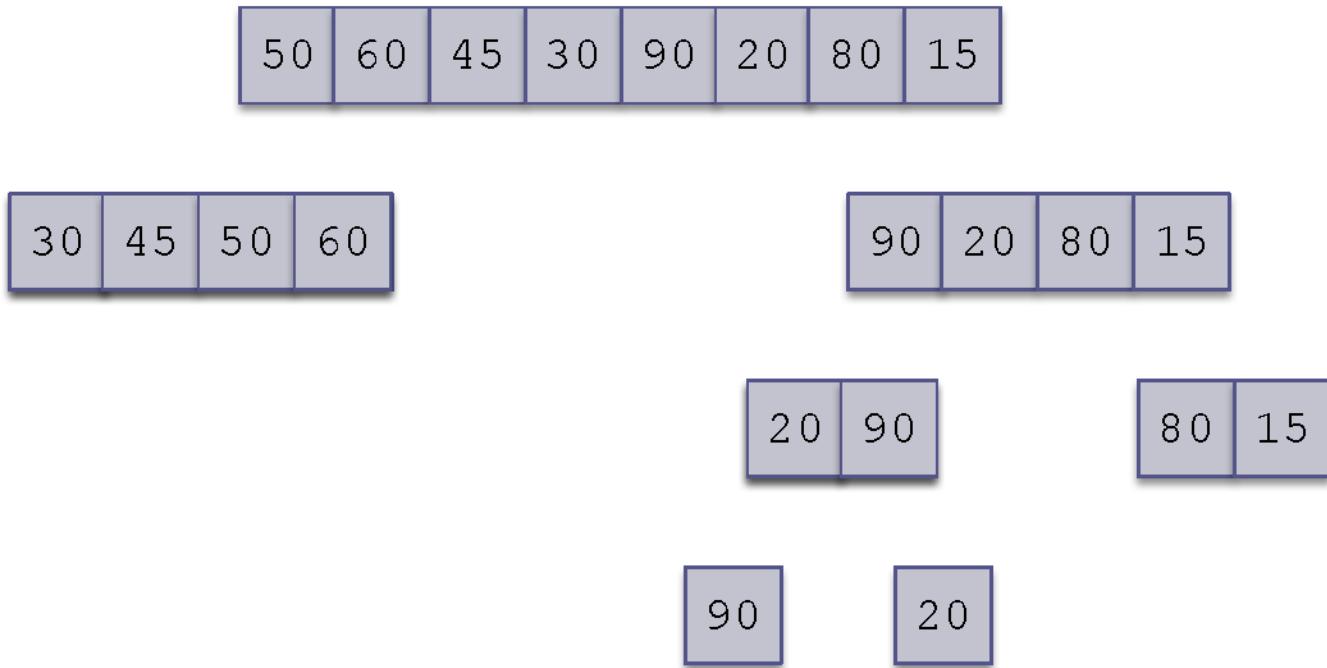
Trace of Merge Sort (cont.)

145



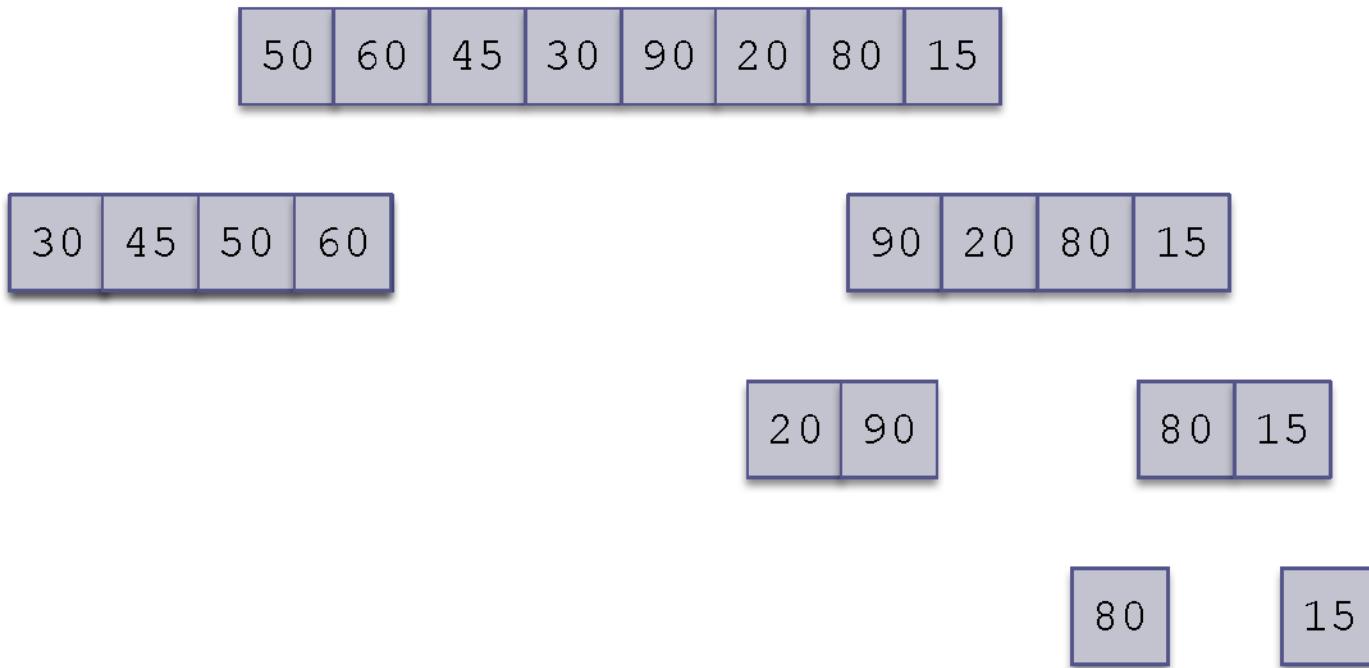
Trace of Merge Sort (cont.)

146



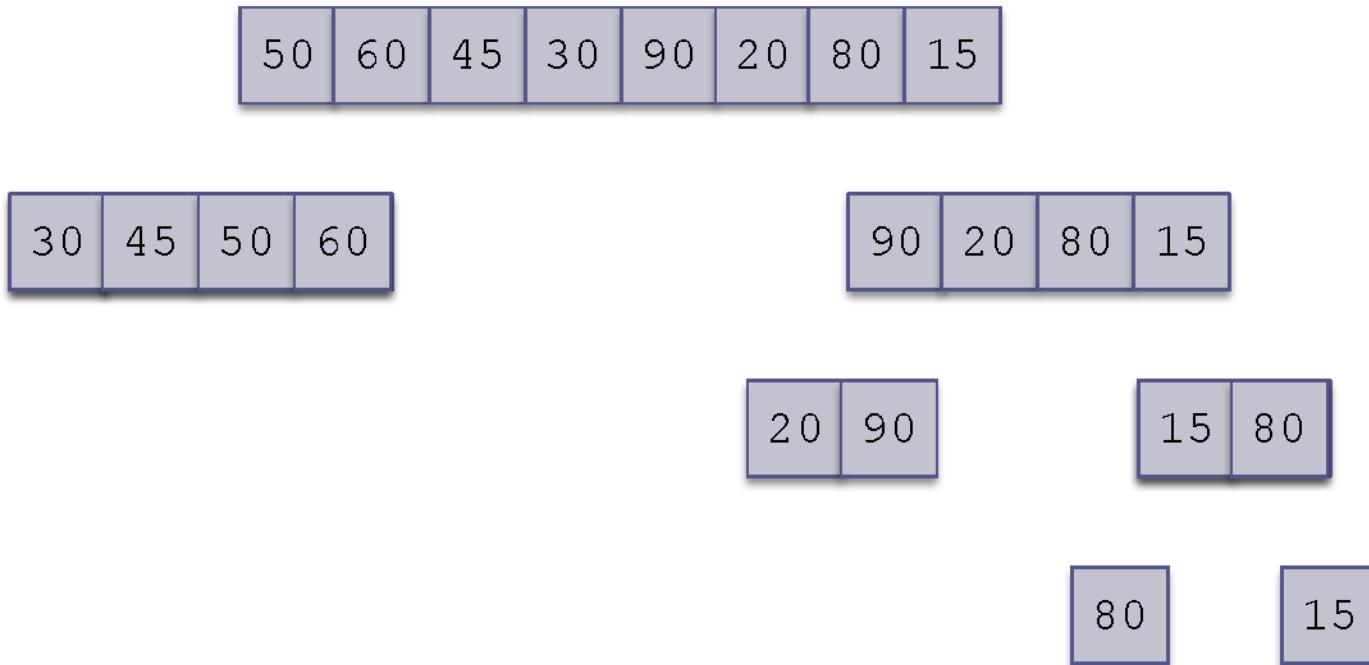
Trace of Merge Sort (cont.)

147



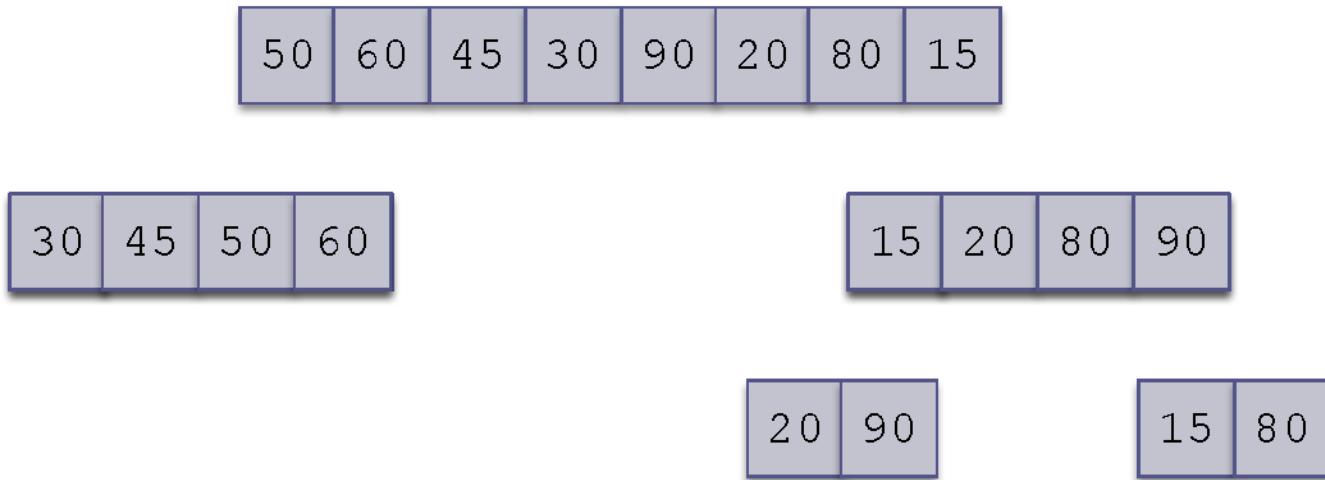
Trace of Merge Sort (cont.)

148



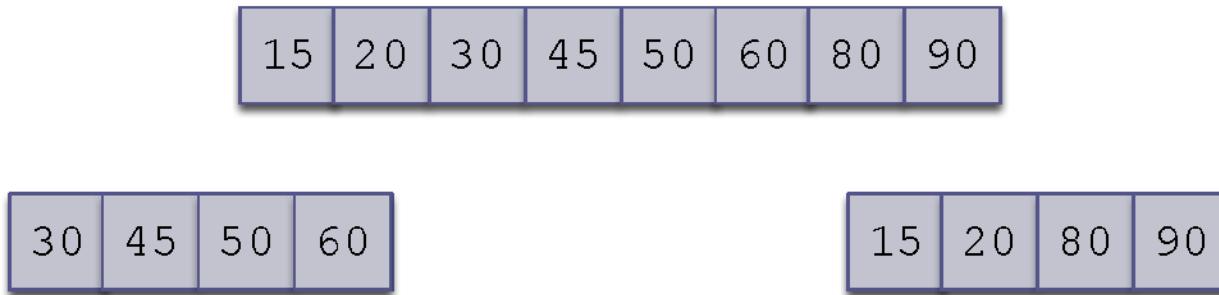
Trace of Merge Sort (cont.)

149



Trace of Merge Sort (cont.)

150



Quick Sort

- Due to Hoare
- Fast in-memory sorting algorithm on the average
- Idea is to choose a “pivot” element “e” in S and divide the original sequence S into

$$S_1 = \{a \in S \mid a < e\}$$

$$S_2 = \{a \in S \mid a \geq e\}$$

Recursively sort S_1 and S_2 and the concatenated sorted subsequences S_1 and S_2

- If we have an array for S, we can do all the above steps in-place unlike merge sort.
- Recursive function : **quicksort**($A, first, last$) where
 $0 \leq first \leq last \leq n - 1$

Quick sort algorithm

- We describe how to do the partitioning later
- The indexes first and last are the end points of the array being sorted
- The index of the pivot after partitioning is pivIndex

Algorithm for Quicksort

1. if first < last then
2. Partition the elements in the subarray first . . . last so that the pivot value is in its correct place (subscript pivIndex)
3. Recursively apply quicksort to the subarray first . . . pivIndex - 1
4. Recursively apply quicksort to the subarray pivIndex + 1 . . . last

Quick Sort Partitioning

Partition(A,first,last):

Input: Array A[first..last] and A[first] contains pivot element e

Output : Return $\text{first} \leq \text{pivotIndex} \leq \text{last}$ such that A is partitioned into subarrays $A_1 = A[\text{first}..\text{pivIndex}-1]$ and $A_2 = A[\text{pivIndex}+1..\text{last}]$ such that $A_1 = \{a \in A \mid a < e\}$ and $A_2 = \{a \in A \mid a \geq e\}$ and $A[\text{pivotIndex}] = e$

$e \leftarrow A[\text{first}]$

$\text{up} \leftarrow \text{first}+1; \text{down} \leftarrow \text{last}$

while $\text{up} < \text{down}$

 while $\text{up} < \text{down}$ and $A[\text{up}] < e$

$\text{up} \leftarrow \text{up} + 1$

 while $\text{up} < \text{down}$ and $A[\text{down}] \geq e$

$\text{down} \leftarrow \text{down} - 1$

 if $\text{up} < \text{down}$

 swap $A[\text{up}]$ and $A[\text{down}]$

$\text{up} \leftarrow \text{up} + 1; \text{down} \leftarrow \text{down} - 1$

swap $A[\text{down}]$ with $A[\text{first}]$

return down

Algorithm for Partitioning

182

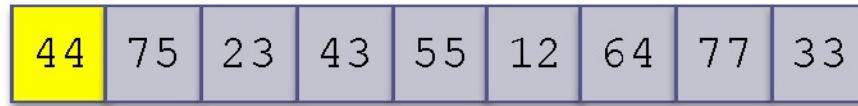
44	75	23	43	55	12	64	77	33
----	----	----	----	----	----	----	----	----

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript first

Trace of Partitioning (cont.)

183

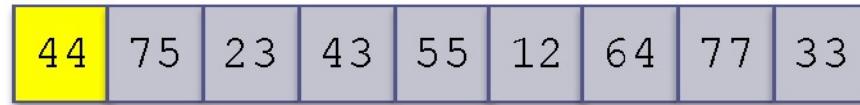


If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript **first**

Trace of Partitioning (cont.)

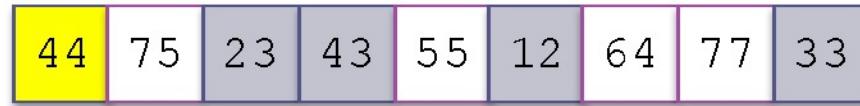
184



For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

Trace of Partitioning (cont.)

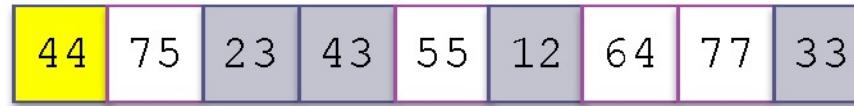
185



For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

Trace of Partitioning (cont.)

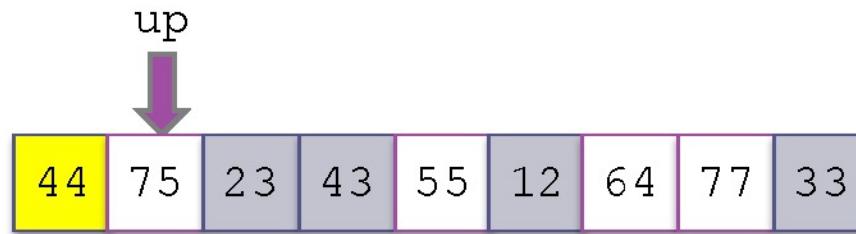
186



Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)

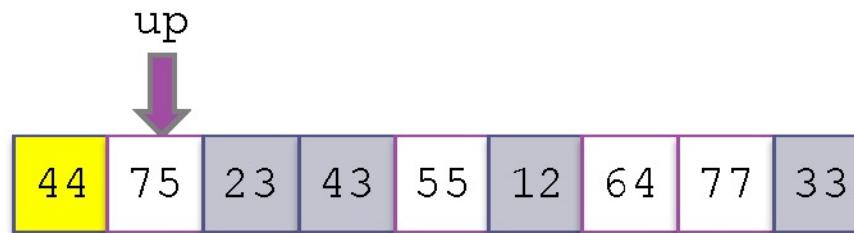
187



Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)

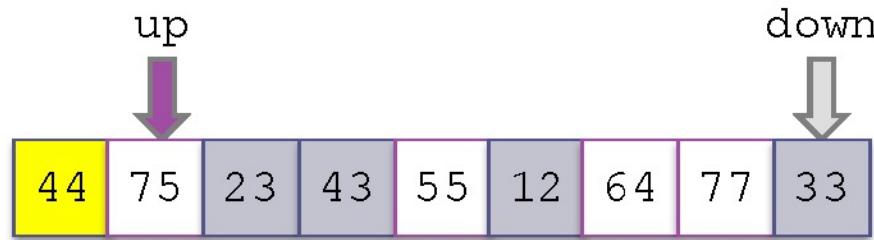
188



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)

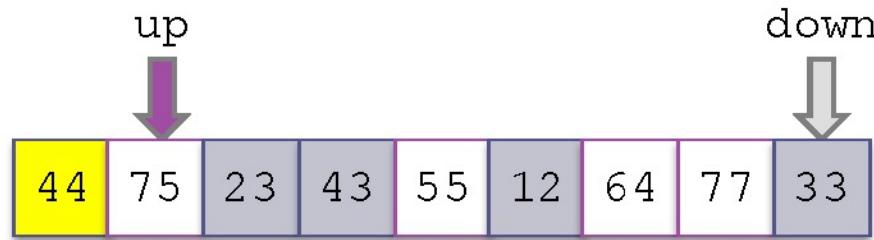
189



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)

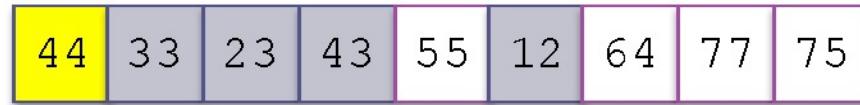
190



Exchange these values

Trace of Partitioning (cont.)

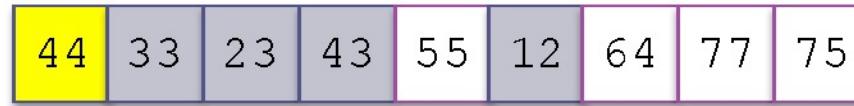
191



Exchange these values

Trace of Partitioning (cont.)

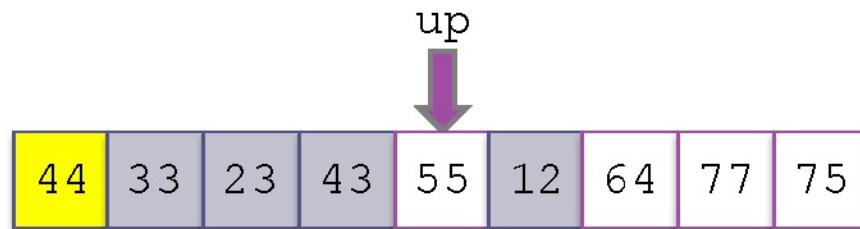
192



Repeat

Trace of Partitioning (cont.)

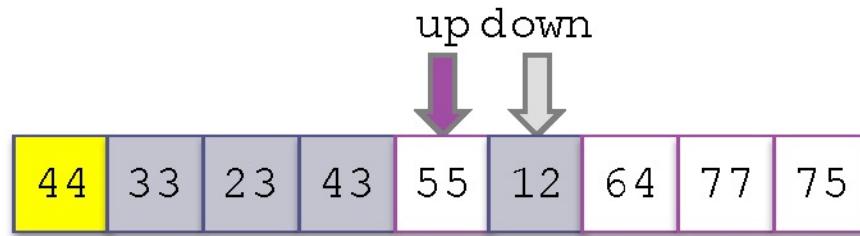
193



Find first value at left end greater
than pivot

Trace of Partitioning (cont.)

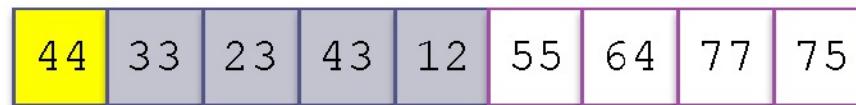
194



Find first value at right end less than
or equal to pivot

Trace of Partitioning (cont.)

195



Exchange

Trace of Partitioning (cont.)

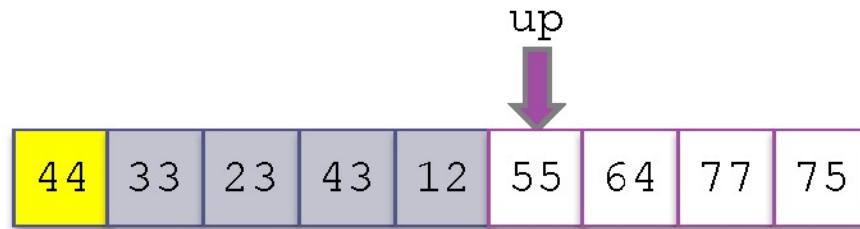
196



Repeat

Trace of Partitioning (cont.)

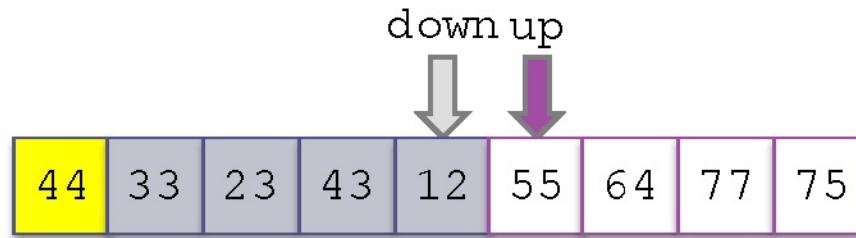
197



Find first element at left end
greater than pivot

Trace of Partitioning (cont.)

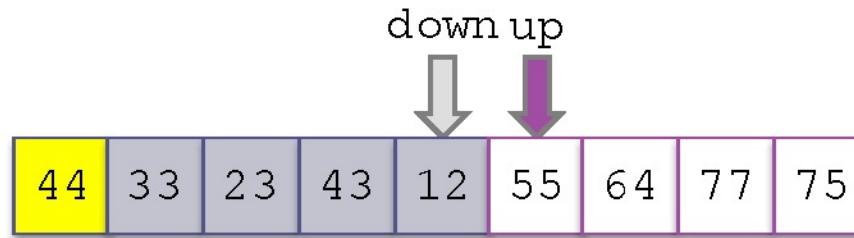
198



Find first element at right end less
than or equal to pivot

Trace of Partitioning (cont.)

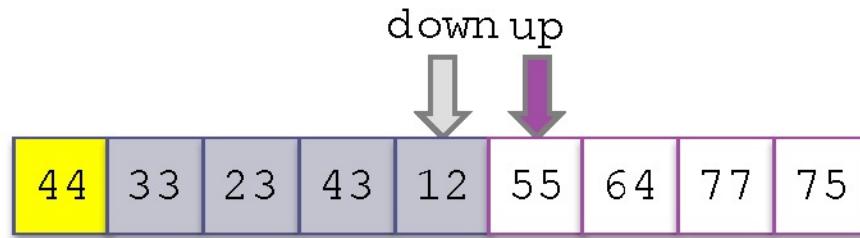
199



Since down has "passed" up, do
not exchange

Trace of Partitioning (cont.)

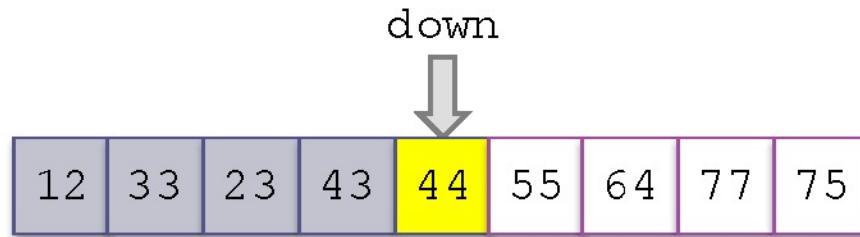
200



Exchange the pivot value with the
value at down

Trace of Partitioning (cont.)

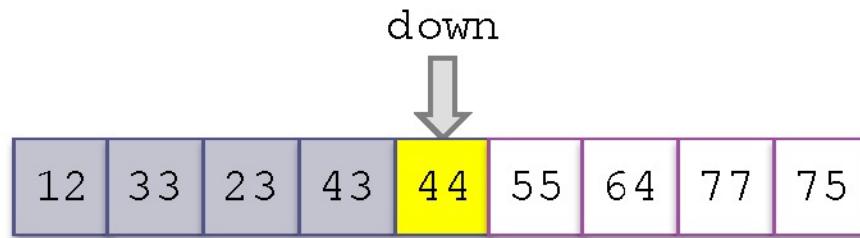
201



Exchange the pivot value with the
value at down

Trace of Partitioning (cont.)

202



The pivot value is in the correct position; return the value of down and assign it to the pivot index pivIndex

Quick sort example



Pivot
↑

After partitioning on pivot



Pivot
↑



After partitioning on pivot



Pivot
↑

After partitioning on pivot

Sorted sub array



Pivot
↑

After partitioning on pivot



Pivot
↑

After partitioning on pivot

After partitioning on pivot



Sorted sub array

Sorted array

Quick Sort

- Due to Hoare
- Fast in-memory sorting algorithm on the average
- Idea is to choose a “pivot” element “e” in S and divide the original sequence S into

$$S_1 = \{a \in S \mid a < e\}$$

$$S_2 = \{a \in S \mid a \geq e\}$$

Recursively sort S_1 and S_2 and the concatenated sorted subsequences S_1 and S_2

- If we have an array for S, we can do all the above steps in-place unlike merge sort.
- Recursive function : **quicksort**($A, first, last$) where
 $0 \leq first \leq last \leq n - 1$

Quick sort algorithm

- We describe how to do the partitioning later
- The indexes first and last are the end points of the array being sorted
- The index of the pivot after partitioning is pivIndex

Algorithm for Quicksort

1. if first < last then
2. Partition the elements in the subarray first . . . last so that the pivot value is in its correct place (subscript pivIndex)
3. Recursively apply quicksort to the subarray first . . . pivIndex - 1
4. Recursively apply quicksort to the subarray pivIndex + 1 . . . last

Quick Sort Partitioning

Partition(A,first,last):

Input: Array A[first..last] and A[first] contains pivot element e

Output : Return $\text{first} \leq \text{pivotIndex} \leq \text{last}$ such that A is partitioned into subarrays $A_1 = A[\text{first}..\text{pivIndex}-1]$ and $A_2 = A[\text{pivIndex}+1..\text{last}]$ such that $A_1 = \{a \in A \mid a < e\}$ and $A_2 = \{a \in A \mid a \geq e\}$ and $A[\text{pivotIndex}] = e$

$e \leftarrow A[\text{first}]$

$\text{up} \leftarrow \text{first}+1; \text{down} \leftarrow \text{last}$

while $\text{up} < \text{down}$

 while $\text{up} < \text{down}$ and $A[\text{up}] < e$

$\text{up} \leftarrow \text{up} + 1$

 while $\text{up} < \text{down}$ and $A[\text{down}] \geq e$

$\text{down} \leftarrow \text{down} - 1$

 if $\text{up} < \text{down}$

 swap $A[\text{up}]$ and $A[\text{down}]$

$\text{up} \leftarrow \text{up} + 1; \text{down} \leftarrow \text{down} - 1$

swap $A[\text{down}]$ with $A[\text{first}]$

return down

Algorithm for Partitioning

182

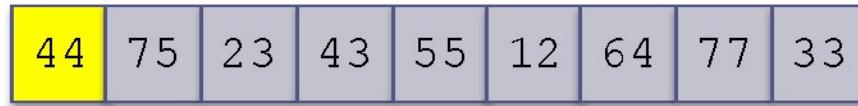
44	75	23	43	55	12	64	77	33
----	----	----	----	----	----	----	----	----

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript first

Trace of Partitioning (cont.)

183

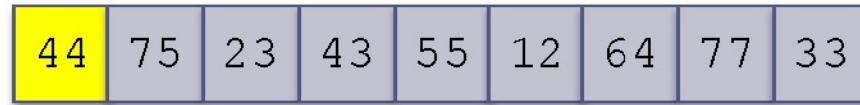


If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript **first**

Trace of Partitioning (cont.)

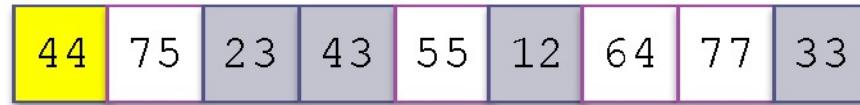
184



For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

Trace of Partitioning (cont.)

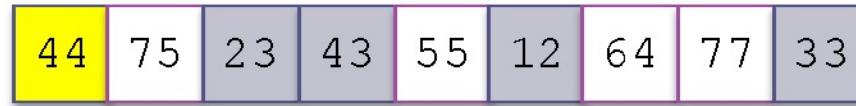
185



For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

Trace of Partitioning (cont.)

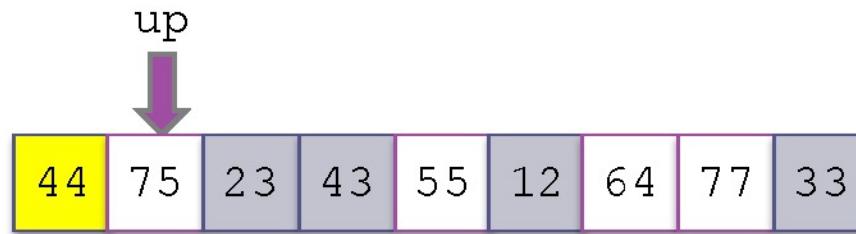
186



Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)

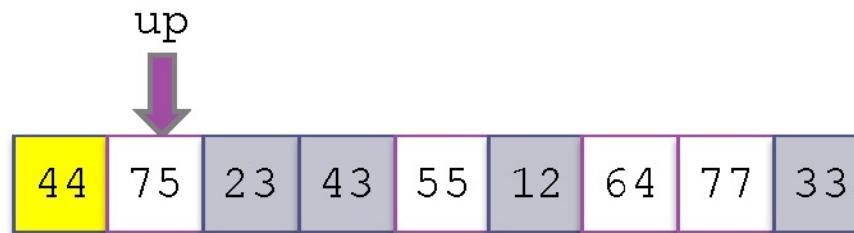
187



Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)

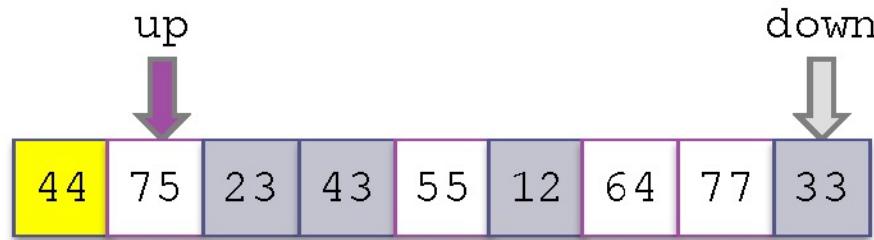
188



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)

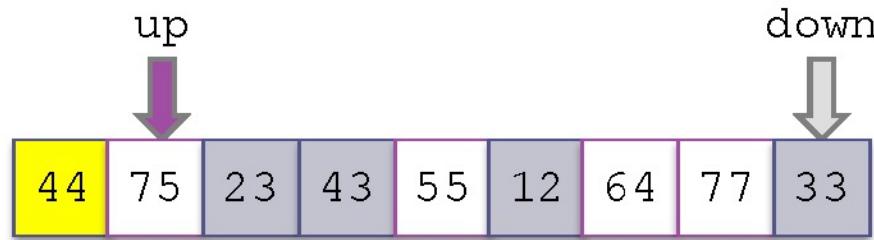
189



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)

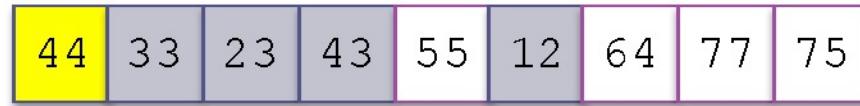
190



Exchange these values

Trace of Partitioning (cont.)

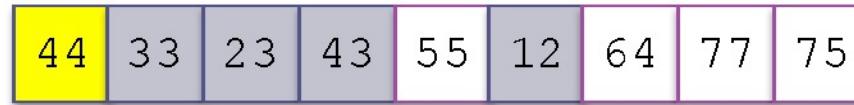
191



Exchange these values

Trace of Partitioning (cont.)

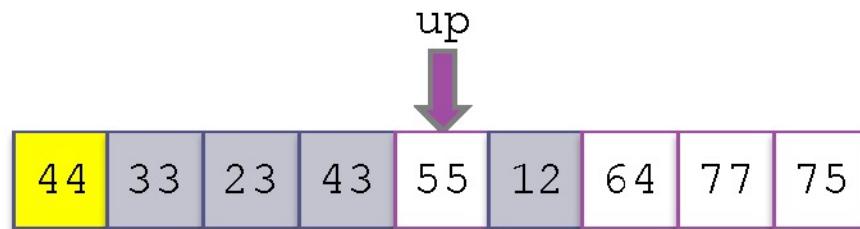
192



Repeat

Trace of Partitioning (cont.)

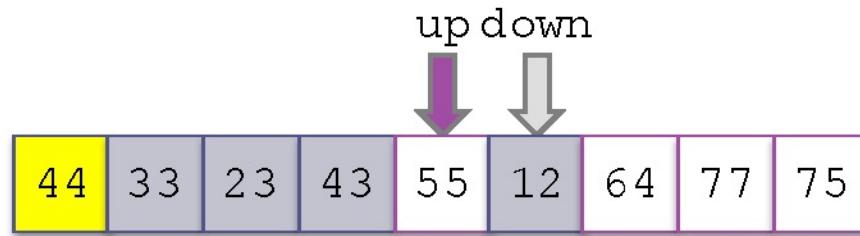
193



Find first value at left end greater
than pivot

Trace of Partitioning (cont.)

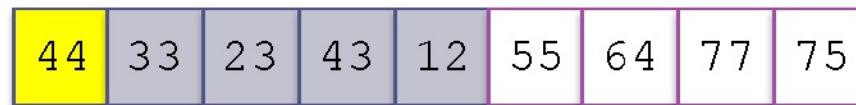
194



Find first value at right end less than
or equal to pivot

Trace of Partitioning (cont.)

195



Exchange

Trace of Partitioning (cont.)

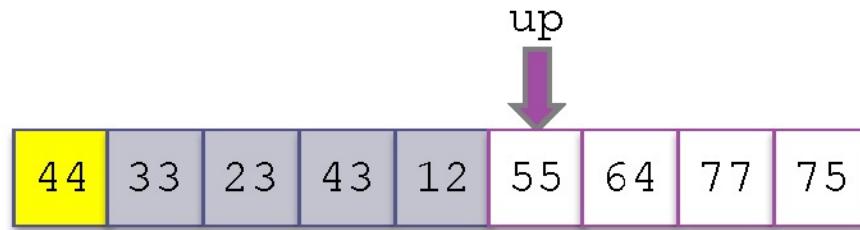
196



Repeat

Trace of Partitioning (cont.)

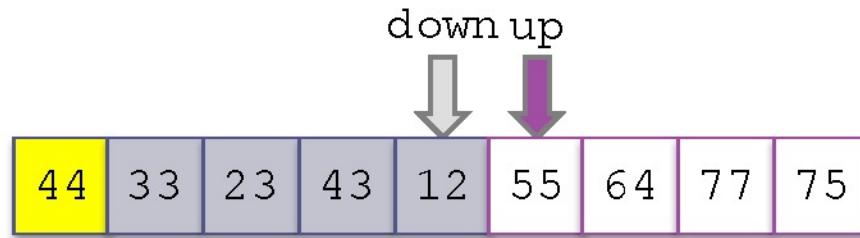
197



Find first element at left end
greater than pivot

Trace of Partitioning (cont.)

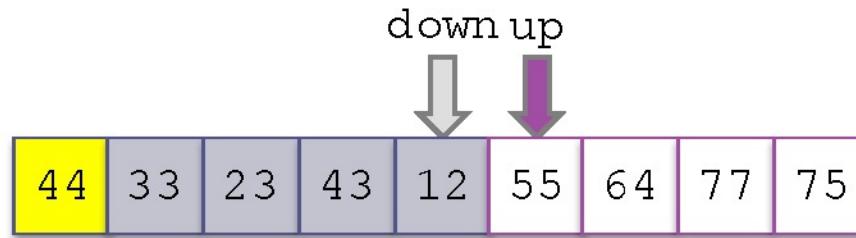
198



Find first element at right end less
than or equal to pivot

Trace of Partitioning (cont.)

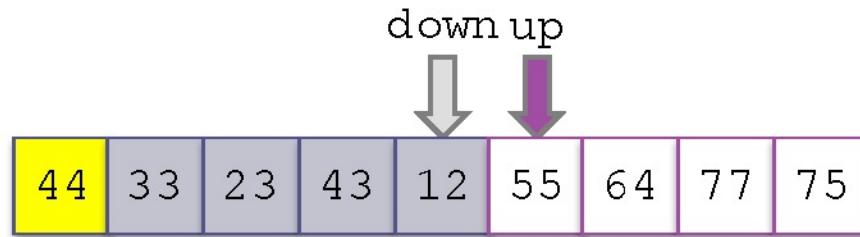
199



Since down has "passed" up, do
not exchange

Trace of Partitioning (cont.)

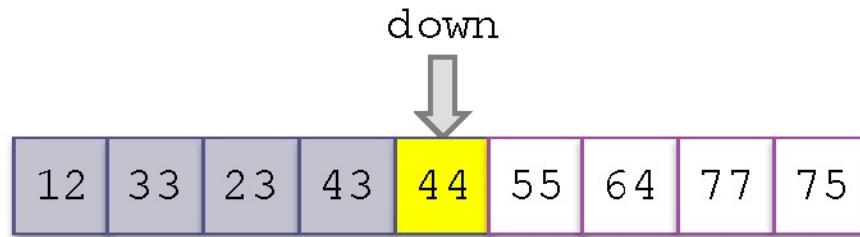
200



Exchange the pivot value with the
value at down

Trace of Partitioning (cont.)

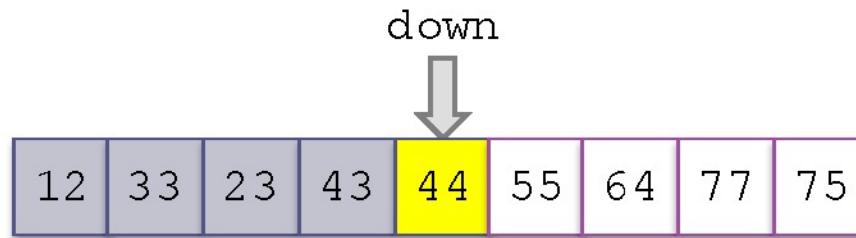
201



Exchange the pivot value with the
value at down

Trace of Partitioning (cont.)

202



The pivot value is in the correct position; return the value of down and assign it to the pivot index pivIndex

Quick sort example



Pivot
↑

After partitioning on pivot



Pivot
↑



Pivot
↑

After partitioning on pivot

Sorted sub array



Pivot
↑



Pivot
↑



Pivot
↑

After partitioning on pivot

After partitioning on pivot



Sorted sub array

Sorted array

Sorting and selection (contd.)

Quick Sort

- Due to Hoare
- Fast in-memory sorting algorithm on the average
- Idea is to choose a “pivot” element “e” in S and divide the original sequence S into

$$S_1 = \{a \in S \mid a < e\}$$

$$S_2 = \{a \in S \mid a \geq e\}$$

Recursively sort S_1 and S_2 and the concatenated sorted subsequences S_1 and S_2

- If we have an array for S, we can do all the above steps in-place unlike merge sort.
- Recursive function : **quicksort**($A, first, last$) where
 $0 \leq first \leq last \leq n - 1$

Choice of pivot element

- Quick sort can have worst-case $O(n^2)$ time complexity. e.g. choice of pivot at each recursive step divides into $n-1$ and 1 size partitions (recursion tree skewed)
- It has $O(n \log n)$ best case time complexity when recursion tree is balanced with equal size partitions
- Typically randomized quick sort used where pivot is chosen at random. Customary to pick median of a random sample of few elements.
- For input size m , let $a_1 \leq a_2 \dots \leq a_m$. Consider app. $m/2$ elements, $a_{\lfloor m/4 \rfloor + 1}, \dots, a_{\lfloor 3m/4 \rfloor}$. Each of these elements bounds sizes $a_{\lfloor m/4 \rfloor}$ of smaller and larger subsequences between $m/4$ and $3m/4$; input size for recursion $\leq 3m/4$
- Probability of choosing one of these as pivot is $\frac{1}{2}$
- Expected height of recursion tree = expected # of recursive invocations until we choose $\log_{4/3} n$ such pivots = $2 \log_{4/3} n$
- Time spent at each level is $O(n)$ \rightarrow expected time complexity is $O(n \log n)$

Example for m=12

Let $a_1 \leq a_2 \leq a_3 \dots \leq a_{12}$. $m/4 = 3$ and $3m/4 = 9$

Consider elements $a_4, a_5, a_6, a_7, a_8, a_9$.

Approximately $m/2 = 6$ elements

$a_4 \geq 4$ elements and $a_4 \leq 9$ elements

$a_5 \geq 5$ elements and $a_5 \leq 8$ elements

$a_6 \geq 6$ elements and $a_6 \leq 7$ elements

$a_7 \geq 7$ elements and $a_7 \leq 6$ elements

$a_8 \geq 8$ elements and $a_8 \leq 5$ elements

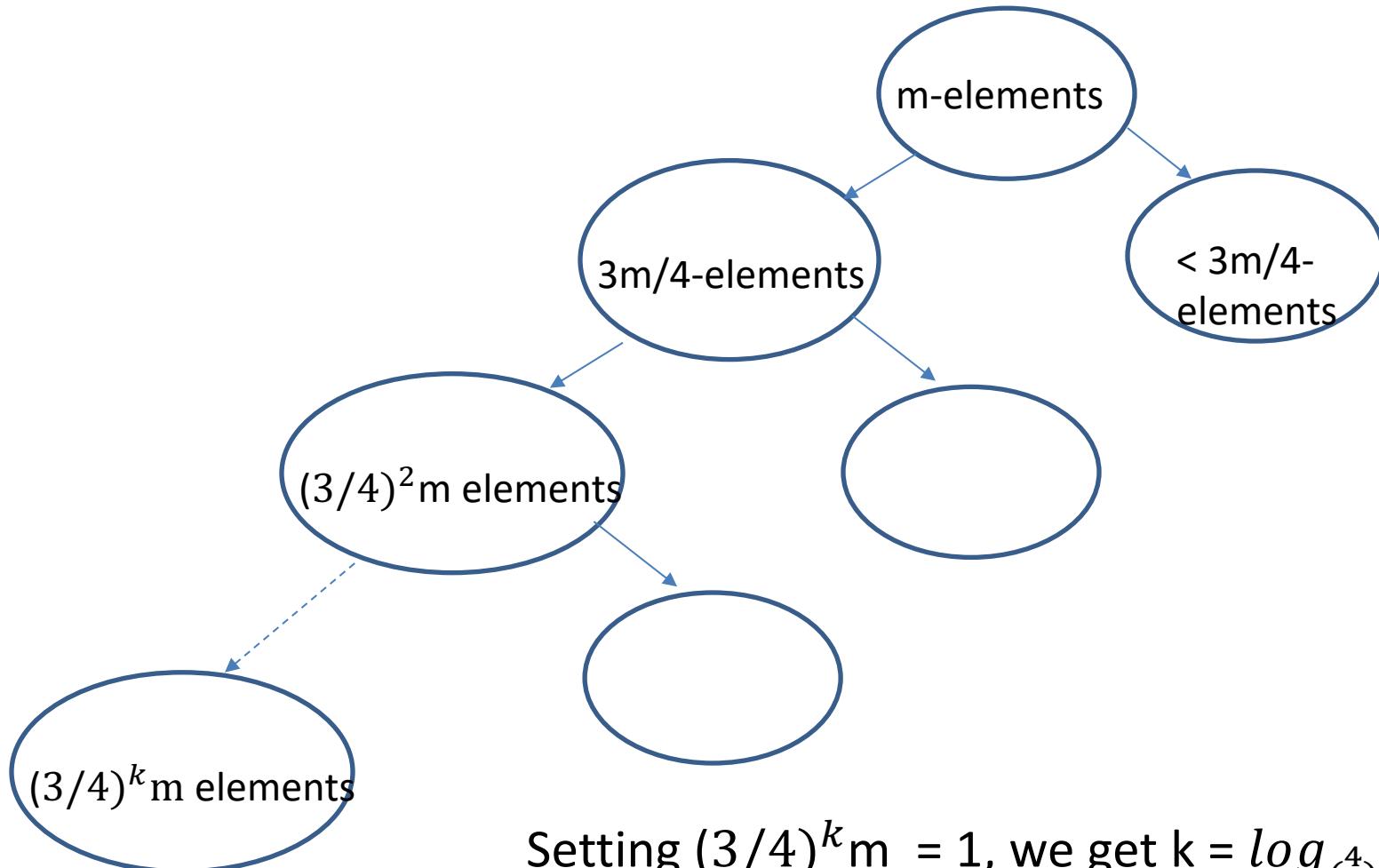
$a_9 \geq 9$ elements and $a_9 \leq 4$ elements

If you take any of these as $m/2$ elements as pivots,

$$\max(|S1|, |S3|) \leq 9 = 3m/4$$

Probability of choosing any of these as pivots = $(m/2)/m = 1/2$

Quick sort (worst-case) recursion tree for $m/2$ pivot choices



Setting $(3/4)^k m = 1$, we get $k = \log_{(\frac{4}{3})} m$

Selection problem

- Finding k-th smallest element of a set S of n elements
- Using heap and extracting minimum k times gives time complexity $O(n + k \log n)$. This finds all p-th smallest elements where $1 \leq p \leq k$
- Can be done faster using divide-and-conquer approach.
- Similar to QuickSort, choose a pivot e and divide S into 3 sets S1, S2 and S3.
 - (a) if $k \leq |S1|$, recursively find k-th smallest element in S1
 - (b) if $|S1| < k \leq |S1| + |S2|$, 'e' is k-th smallest element
 - (c) Otherwise recursively find $k - (|S1| + |S2|)$ -th smallest element in S3.
- If we choose pivot to be median of 5-element medians
- $T(|S|) \leq T(\max(|S1|, |S3|)) + T(|S4|) + k |S|$
S4 is the set of 5-element medians from S.

Selection example

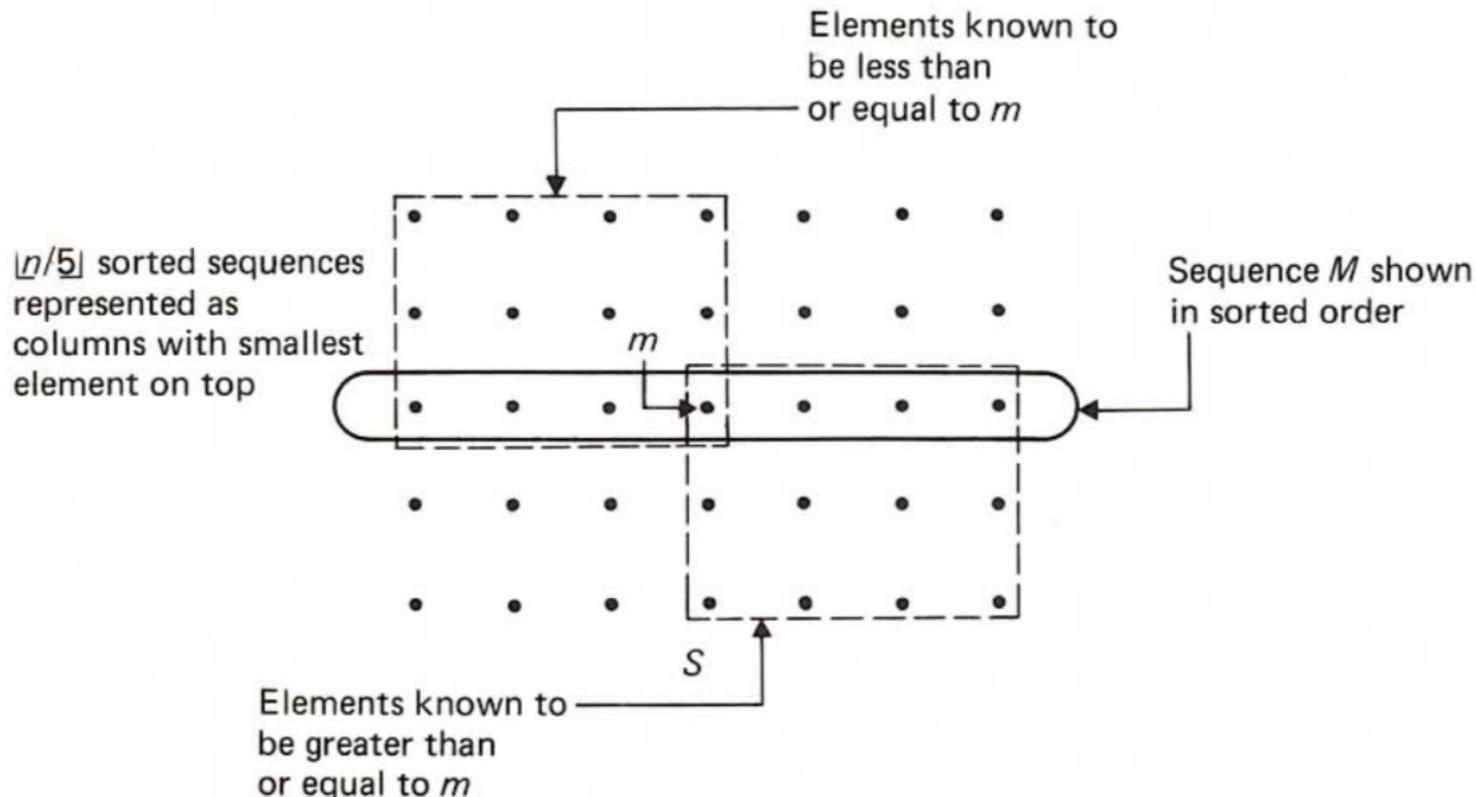
- S has 12 elements using pivot e

$$|S_1| = 5, |S_2| = 3, |S_3| = 4$$

3rd smallest element must be in |S₁|

6th smallest element = pivot element

10th smallest element -- 2nd smallest element in S₃



Choice of pivot

- Pivot element \geq At least $\frac{1}{2}$ of $\lceil n/5 \rceil$ medians Each of these elements ≥ 3 elements
 - pivot element $\geq 3/2 \lceil n/5 \rceil \geq 3n/10$ elements
 - $|S_3| \leq 7n/10$. By symmetric argument, $|S_1| \leq 7n/10$
- $|S_4| \leq n/5$
- $T(n) \leq T(7n/10) + T(n/5) + k n$ where $|S| = n$
- Note that $7n/10 + n/5 = 9n/10 < n$
- $T(n)$ is $O(n)$ – linear time worst-case complexity but constant may be prohibitive
- A better choice is to use random pivot and it gives expected $O(n)$ time complexity.

Text Processing Algorithms

Pattern (String) Matching

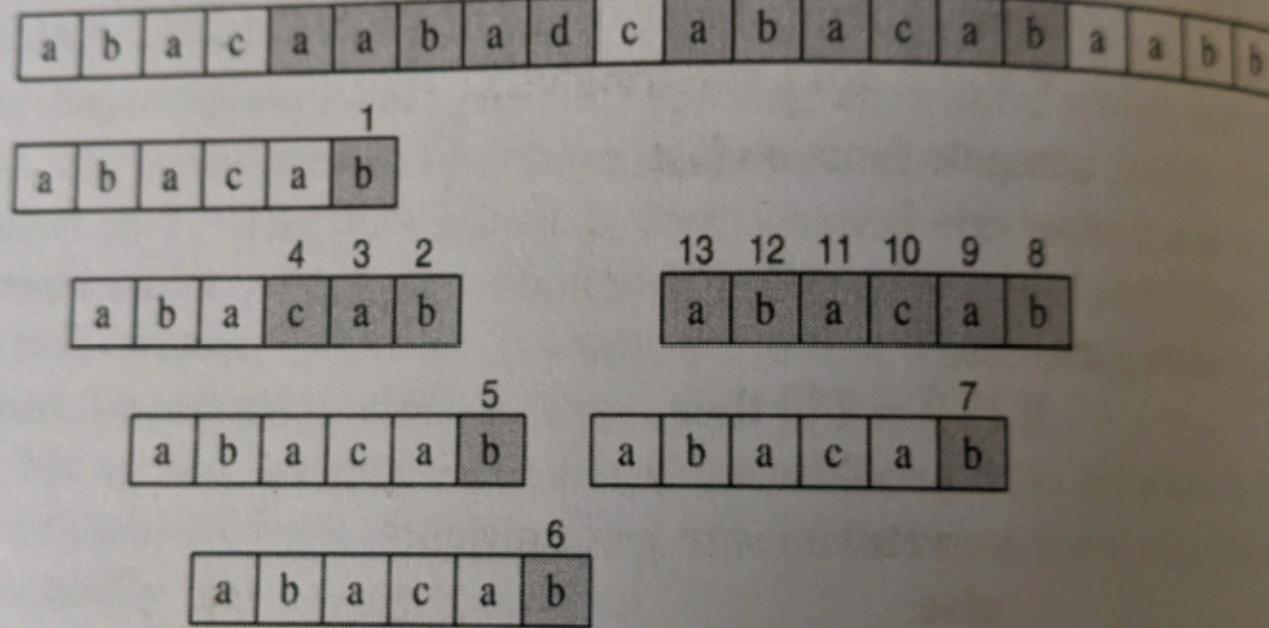
- **Given :** A text of characters T of length n and a pattern string P of length m
Find if P occurs in T and if it does get starting position of first occurrence in T
- **Brute force approach :**
Align pattern at each position of text and check if it matches
At each position it takes at most m comparisons and time complexity is $O(mn)$ assuming character comparison takes $O(1)$ time.
If m is very small compared to n this is ok
- Better approaches take advantage of
 - (i) failed comparison at a position to skip text characters
 - (ii) repetition of sub pattern within a pattern to avoid back tracking in text

Boyer-Moore Algorithm

- Simple version of BM algorithm:
 - (a) At text position, compare pattern with text T starting from end of pattern (say of length m)
 - (b) If a mismatch occurs for a text character “c” at a position “k”, if it does not occur in the pattern at all, then we can slide the pattern past position k so that we can start comparison from end of pattern from position $k+m$
This allows some text characters to be skipped
- we define for each character “c” of text :

$\text{last}(c) = \text{highest index } j \text{ in pattern } P \text{ such that } P(j) = c,$
 $= -1$ if c does not occur in P
- For a mismatch at position k, restart pattern comparison at text position $k + m - \min(j, 1 + \text{last}(T[i]))$ where j is the pattern position where mismatch occurred. In the worst case, it can slide by only one position

Example of BM matching alg.



The $\text{last}(c)$ function:

c	a	b	c	d
$\text{last}(c)$	4	5	3	-1

BM algorithm complexity

- Simple version has worst-case complexity $O(nm + d)$ where d is size of alphabet
- Simple version wastes comparisons already made when pattern slides less than m
- Original BM algorithm avoids this using ideas of KMP
- More complex but takes $O(n+m+d)$ time in the worst-case

Knuth-Morris-Pratt Algorithm

- Uses a DFA (deterministic finite state automaton) in recognizing substring patterns of a text
- A DFA has $(S, \Sigma, \delta, s_0, F)$
 - S is a finite set of states
 - Σ finite alphabet
 - $s_0 \in S$ is start state
 - $F \subseteq S$ set of final states
 - $\delta : S \times \Sigma \rightarrow S$ (partial) transition function

$\delta(s,a)$ specifies for a current state s and reading an input character a what the next state should be

- A string $a_1 a_2 \dots a_n$ is accepted if we reach a final state after processing string using δ
- We are interested in constructing a DFA such that all strings with a pattern (regular expression) are accepted by that DFA.

Failure function in string matching

- KMP algorithm uses a DFA where

$S = \{0, 1, 2, \dots, m\}$, 0 is start state and m is final state

For a pattern string $P = p_1 \ p_2 \dots \ p_m$ transition function δ is defined as follows for $1 \leq j \leq m-1$:

$$\delta(j-1, p_j) = j + 1$$

$$\delta(j-1, a) = f(j), \text{ for } a \neq p_j$$

- f is called failure function which specifies the latest previous position in pattern from which we can resume comparison for the current or next text position.
- If $f(j) = k$ and $k > 0$, then k is the largest value $< j$ such that $p_1 \ p_2 \dots \ p_k$ is a suffix of $p_1 \ p_2 \dots \ p_j$

Text Processing Algorithms

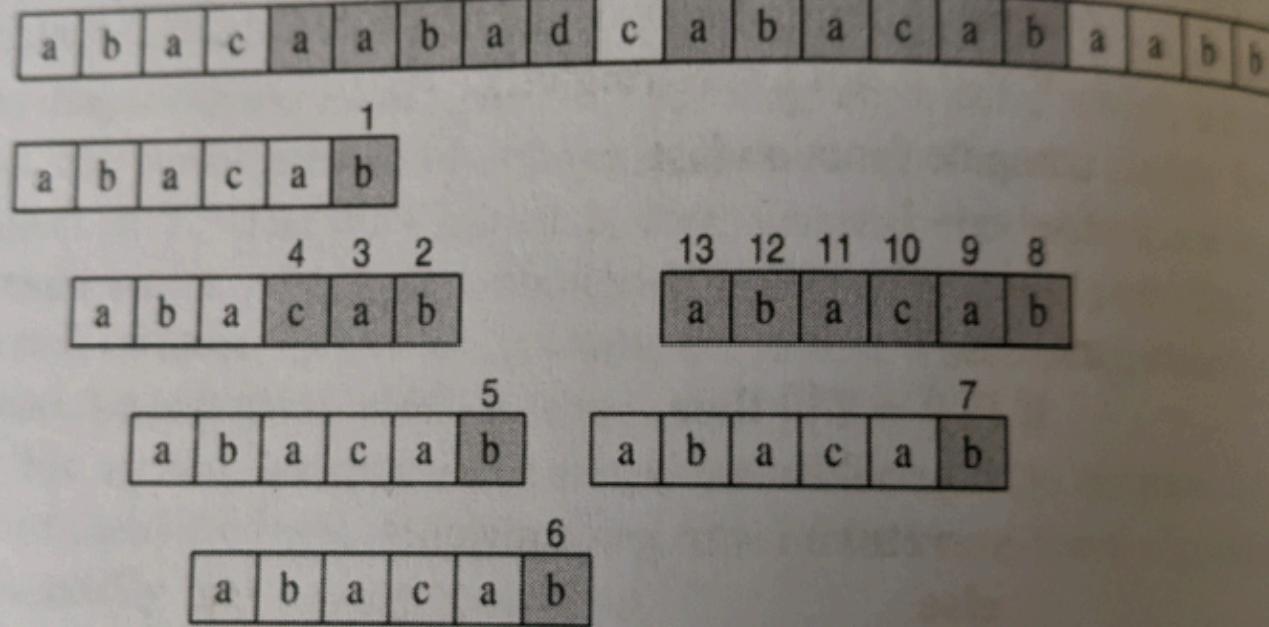
Pattern (String) Matching

- **Given :** A text of characters T of length n and a pattern string P of length m
Find if P occurs in T and if it does get starting position of first occurrence in T
- **Brute force approach :**
Align pattern at each position of text and check if it matches
At each position it takes at most m comparisons and time complexity is $O(mn)$ assuming character comparison takes $O(1)$ time.
If m is very small compared to n this is ok
- Better approaches take advantage of
 - (i) failed comparison at a position to skip text characters
 - (ii) repetition of sub pattern within a pattern to avoid back tracking in text

Boyer-Moore Algorithm

- Simple version of BM algorithm:
 - (a) At text position, compare pattern with text T starting from end of pattern (say of length m)
 - (b) If a mismatch occurs for a text character “c” at a position “k”, if it does not occur in the pattern at all, then we can slide the pattern past position k so that we can start comparison from end of pattern from position $k+m$
This allows some text characters to be skipped
- we define for each character “c” of text :
 $\text{last}(c) = \text{highest index } j \text{ in pattern } P \text{ such that } P(j) = c,$
 $= -1$ if c does not occur in P
- For a mismatch at text position k, restart pattern comparison at text position $k + m - \min(j, 1 + \text{last}(T[k]))$ where j is the pattern position where mismatch occurred. In the worst case, it can slide by only one position

Example of BM matching alg.



The $\text{last}(c)$ function:

c	a	b	c	d
$\text{last}(c)$	4	5	3	-1

BM algorithm complexity

- Simple version has worst-case complexity $O(nm + d)$ where d is size of alphabet
- Simple version wastes comparisons already made when pattern slides less than m
- Original BM algorithm avoids this using ideas of KMP
- More complex but takes $O(n+m+d)$ time in the worst-case

Knuth-Morris-Pratt Algorithm

- Uses a DFA (deterministic finite state automaton) in recognizing substring patterns of a text
- A DFA has $(S, \Sigma, \delta, s_0, F)$
 - S is a finite set of states
 - Σ finite alphabet
 - $s_0 \in S$ is start state
 - $F \subseteq S$ set of final states
 - $\delta : S \times \Sigma \rightarrow S$ (partial) transition function

$\delta(s,a)$ specifies for a current state s and reading an input character a what the next state should be

- A string $a_1 a_2 \dots a_n$ is accepted if we reach a final state after processing string using δ
- We are interested in constructing a DFA such that all strings with a pattern (regular expression) are accepted by that DFA.

Failure function in string matching

- KMP algorithm uses a DFA where

$S = \{0, 1, 2, \dots, m\}$, 0 is start state and m is final state

For a pattern string $P = p_1 \ p_2 \dots \ p_m$ transition function δ is defined as follows for $1 \leq j \leq m-1$:

$$\delta(j-1, p_j) = j$$

$$\delta(j-1, a) = f(j), \text{ for } a \neq p_j$$

- f is called failure function which specifies the latest previous position in pattern from which we can resume comparison for the current or next text position.
- If $f(j) = k$ and $k > 0$, then k is the largest value $< j$ such that $p_1 \ p_2 \dots \ p_k$ is a suffix of $p_1 \ p_2 \dots \ p_j$ if it exists else 0.

KMP Matching Algorithm

KMPMatch(T,P):

Input : text T[0..n-1] and pattern P[0..m-1] of n and m characters

Output : Start index of first match of P in T, -1 otherwise

$f \leftarrow \text{KMPFailureFunction}(P)$

$i \leftarrow 0; j \leftarrow 0$

while $i < n$

 if $T[i] = P[j]$

 if $j = m-1$

 return $i-m+1$

$i \leftarrow i+1; j \leftarrow j+1$

 else if $j > 0$

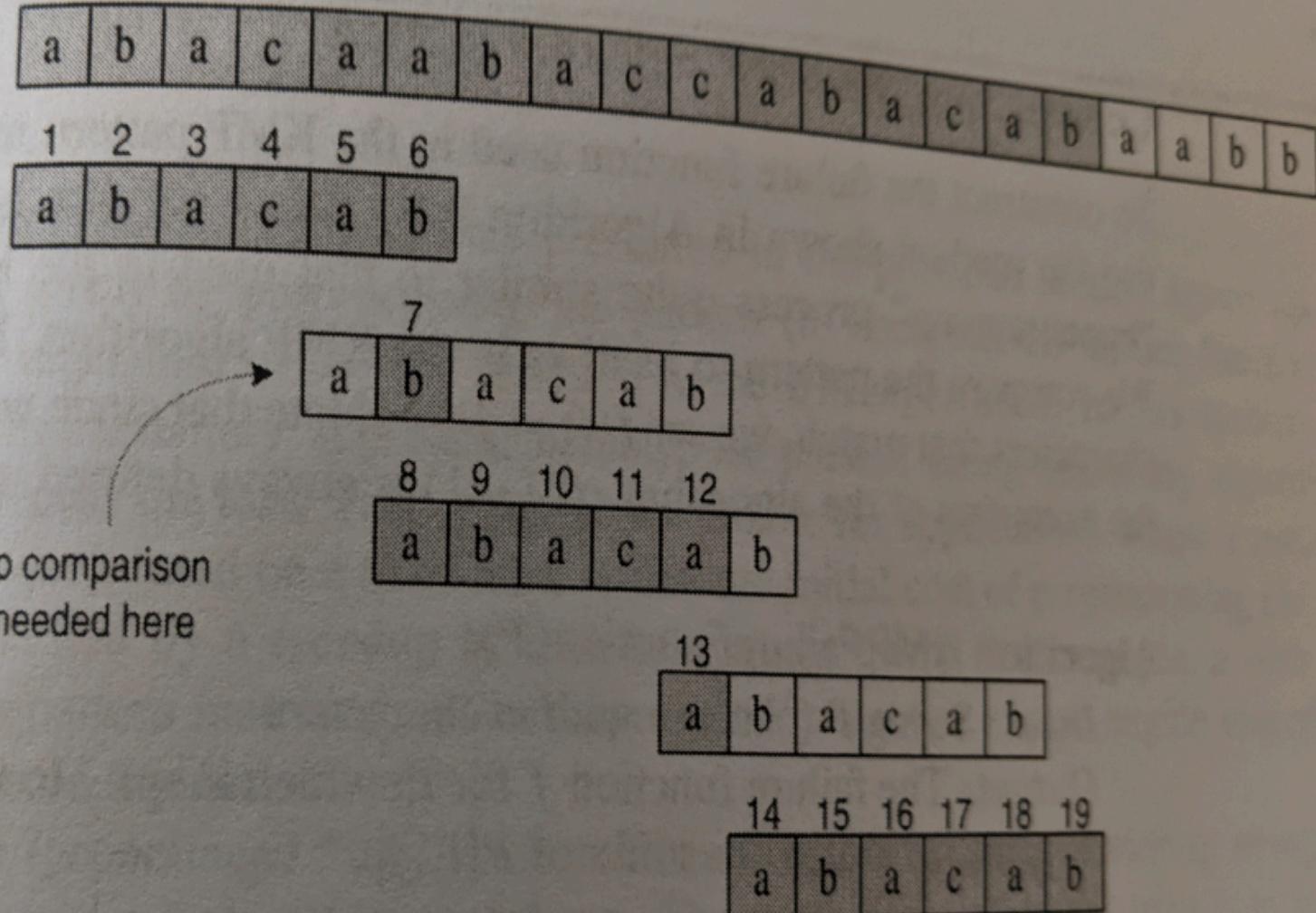
$j \leftarrow f(j-1)$

 else

$i \leftarrow i+1$

return -1

Example of KMP matching alg.



KMP failure function

- **Idea:** Feed pattern itself as an input . If we know $f(1), \dots, f(j-1)$ and $f(j-1) = k_j$. we can compute $f(j)$ as:
 - (i) if $p_j = p_{k_j+1}$ then $f(j) = k_j + 1$
 - (ii) Else apply f repeatedly, if $f(k_i) = k_{i+1}$ then we check (i) again as long as $k_{i+1} > 0$

KMPFailureFunction($P[1..m]$):

```
f(1) ← 0
for j ← 2 to m
    k ← f(j-1)
    while P[j] ≠ P[k+1] and k > 0
        k ← f(k)
    if P[j] ≠ P[k+1] and k = 0
        f(j) ← 0
    else
        f(j) ← k+1
return f
```

KMP Time complexity

- Excluding construction of failure function, in KMPMatch, $O(1)$ time is spent in each while loop iteration assuming character comparison takes $O(1)$ time.
- To determine number of iterations, consider $i-j$, ($i-j \leq n$)
 - (i) when there is a match $i-j$ remains same (i, j both increase by 1)
 - (ii) when there is no match and $j = 0$, $i-j$ increases by 1 (only i increases by 1)
 - (iii) when there is no match and $j > 0$, j is set to $f(j-1)$ which is less than j and hence $i-j$ increases by at least 1
- Hence at end of each iteration either i increases (text position advances) or $i-j$ increases (pattern shift) by at least 1
 \rightarrow # of iterations $\leq 2n \rightarrow$ complexity is $O(n)$ excl. failure function construction
- Failure function takes $O(m)$ time to compute \rightarrow total complexity is $O(n+m)$

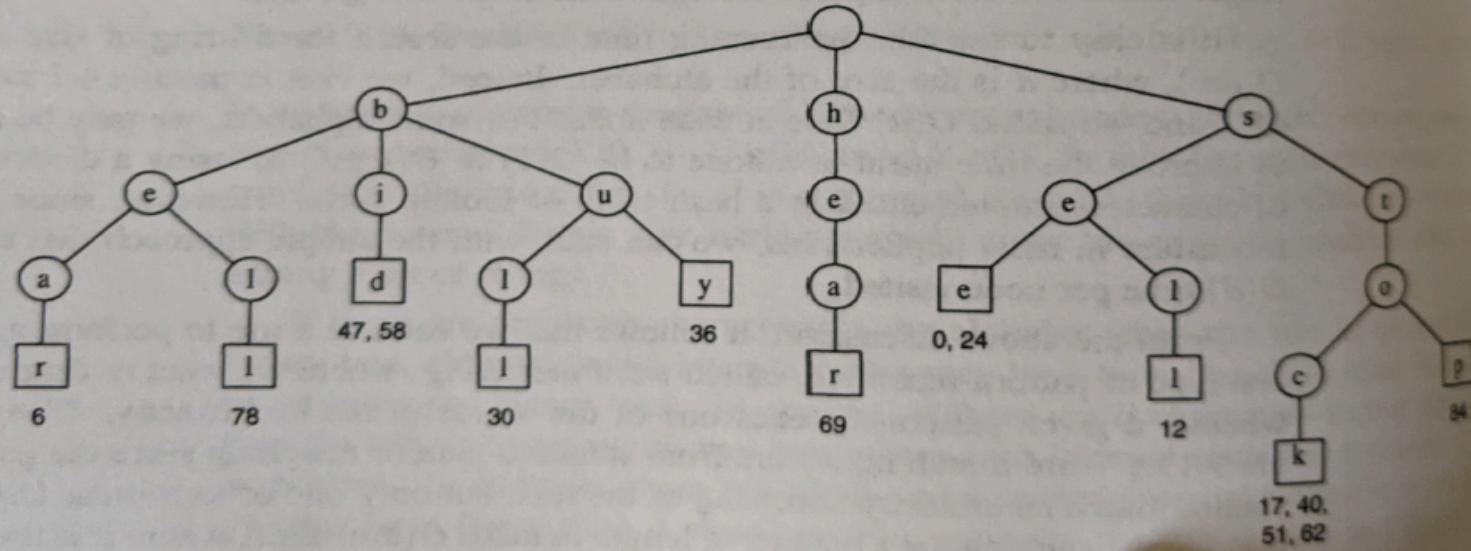
Tries

- Efficient data structure for processing a series of search queries on the same set of text strings
- Given (S, Σ) where $S \subseteq \Sigma^*$, a (compressed) trie T is an ordered tree where
 - (a) each node is labeled with a string from alphabet Σ
 - (b) ordering of children nodes according to some canonical (usually alphabetical) ordering of labels
 - (c) an external node is associated with a string of S formed by concatenation of all labels from root to that node; every string of S is associated with an external node.
 - (d) every internal node must have at least 2 children
- In a standard trie, the label is just a character in Σ and an internal node can have just one child

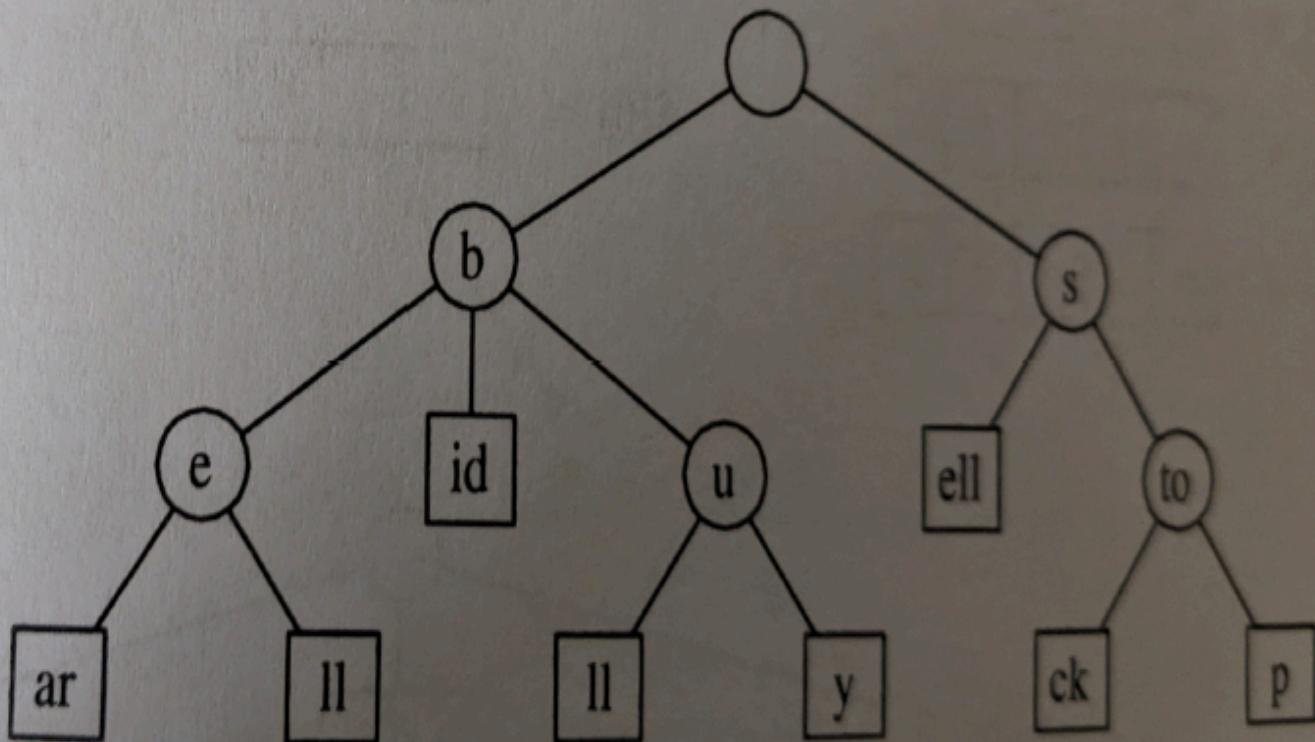
Trie example

s	e	e	a	b	e	a	r	?		s	e	l	l		s	t	o	c	k	!			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e	a	b	u	l	l	?		b	u	y			s	t	o	c	k	!			
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!			
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	r	t	h	e	b	e	l	l	?				s	t	o	p	!				
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				

(a)



Compressed Trie example



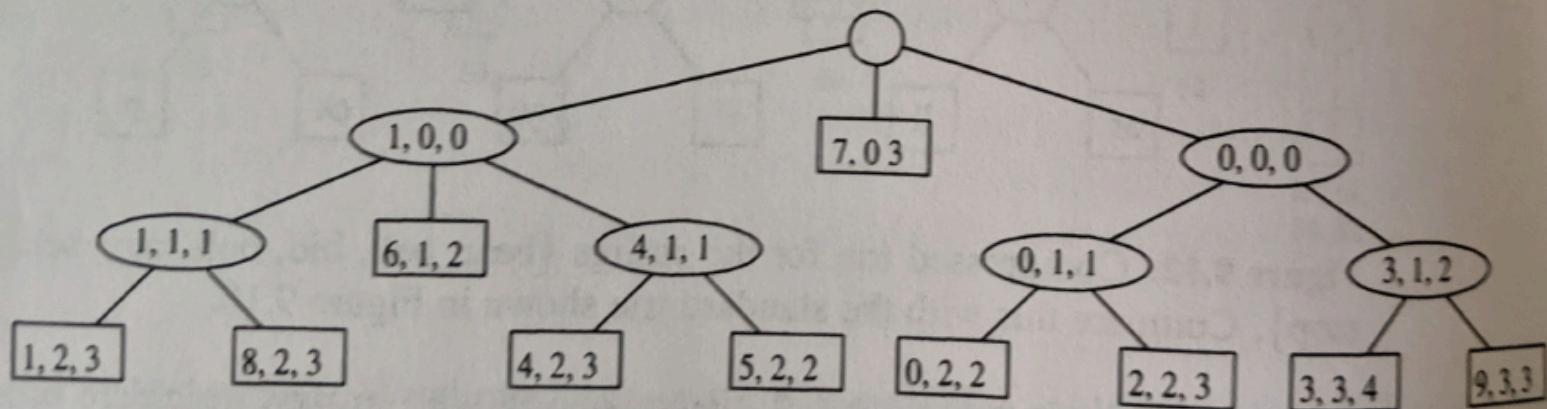
Compressed Trie (with positions) example

	0	1	2	3	4
$S[0] =$	s	e	e		
$S[1] =$	b	e	a	r	
$S[2] =$	s	e	l	l	
$S[3] =$	s	t	o	c	k

	0	1	2	3
$S[4] =$	b	u	l	l
$S[5] =$	b	u	y	
$S[6] =$	b	i	d	

	0	1	2	3
$S[7] =$	h	e	a	r
$S[8] =$	b	e	l	l
$S[9] =$	s	t	o	p

(a)



(b)

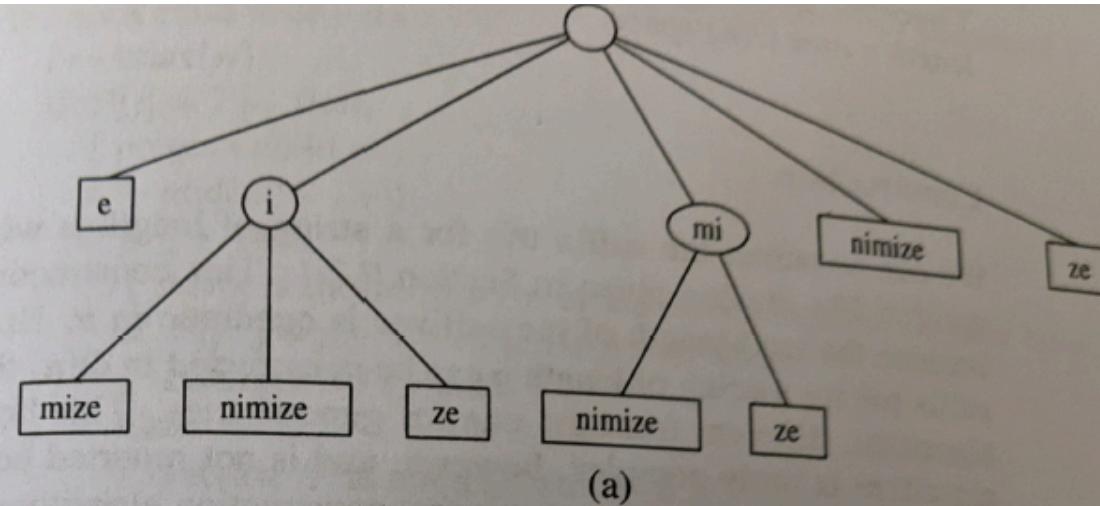
Complexity of tries

- Given (S, Σ) where $|S| = n$ and $|\Sigma| = d$
- Number of nodes in trie is $O(n)$,
- Height is length of longest string in S denoted by m
- Every internal node has at least 2 but at most d children
- Space complexity is $O(n m)$
- Time to search for a string of size k is $O(dk)$
 - find path in T by matching substrings of search string
- Time complexity to construct T
 - (a) inserting a string at a time --- find path in T by tracing prefix of string and when we stop at an internal node (i.e. cannot match any children), insert a new node there for suffix
 - Time to insert a string x is $O(d |x|) \rightarrow$ complexity is $O(dn)$ where $n = \sum_{x \in S} |x|$ where $|x|$ is length of x .

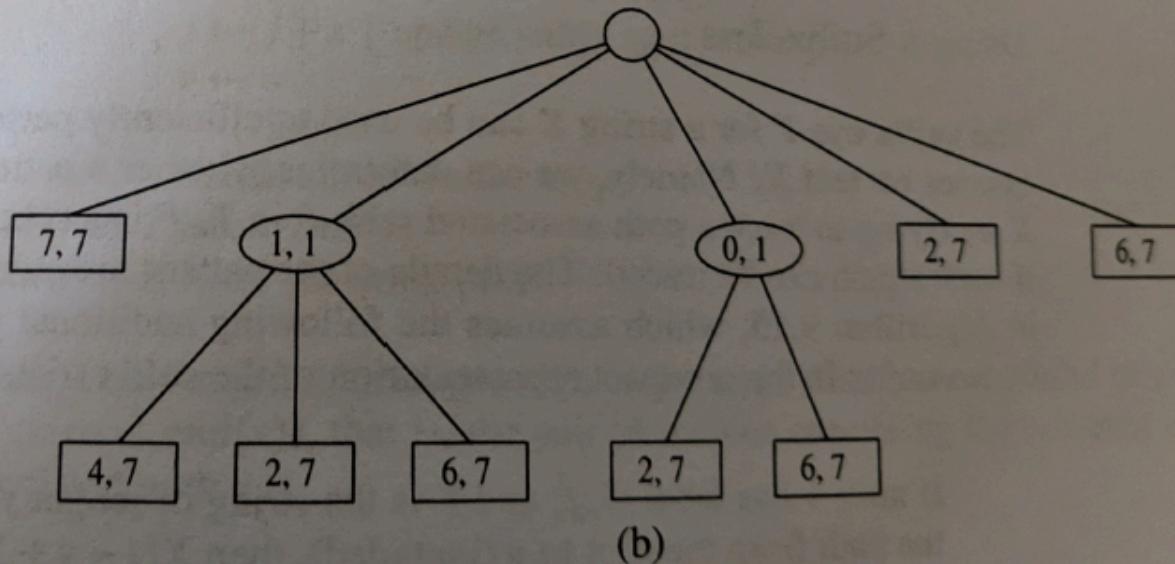
Suffix Tries

- Also called “position tree” that has external nodes representing all possible suffixes of a given string $s = s_0 s_1 \dots s_{m-1}$
- Each node is labeled with an interval (i,j) which represents the substring $s_i \dots s_j$ for $0 \leq i < j < m$
- Useful for many efficient string operations
 - (a) substring search $O(m)$
 - (b) longest common substring
 - (c) longest repeated substring
 - (d) useful in Bioinformatics to search for patterns in DNA or protein sequences
- Space complexity is $O(n)$
- Time to construct a suffix trie – $O(dn)$
- Time to search for a substring of size m in the Suffix Trie – $O(dm)$
- Generalized version for a set of words

Suffix Trie example



m	i	n	i	m	i	z	e
0	1	2	3	4	5	6	7



ADT Set

- Stores distinct elements
- Operations supported :
 - makeSet(e)** – make a set with a single element e
 - union(S₁,S₂)** – returns S₁ U S₂
 - intersect(S₁,S₂)** – returns S₁ ∩ S₂
 - subtract(S₁,S₂)** – returns set of elements in S₁ but not in S₂
- Implementation with ordered sequence takes O(n) time for union, intersect and subtract operations
(use modified merging algorithm)
- Implementation with hash table (no ordering needed) takes O(n) time on the average

Disjoint set Union-Find

- Many algorithms require maintenance of partition sets, i.e. element belonging to a set and only one set.
- They typically require a sequence of union and find operations:
 - union(S_1, S_2) – modify S_1 by $S_1 \cup S_2$ (merge)
 - find(e) – find the set containing element e .
- Two types of implementations:
 - (a) linear – let each element node have a direct link to the set identifier node (changes during union)
 - makeSet(e) – takes $O(1)$ time
 - find(e) – takes $O(1)$ time
 - (b) tree – let each set node have a link to its parent (new set node) during union operation

Example of disjoint set union-finds

$S = \{2, 5, 10, 11, 15, 35\}$

- $S1 \leftarrow \text{makeSet}(2); S2 \leftarrow \text{makeSet}(5); S3 \leftarrow \text{makeSet}(10); \dots$

$S1 = \{2\}, S2 = \{5\}, S3 = \{10\}, S4 = \{11\}, S5 = \{15\}, S6 = \{35\}$

- $\text{Find}(15) = S5$

- $S1 \leftarrow \text{Union}(S1, S3)$

$S1 = \{2, 10\}, S2 = \{5\}, S4 = \{11\}, S5 = \{15\}, S6 = \{35\}$

- $S2 \leftarrow \text{Union}(S2, S5)$

$S1 = \{2, 10\}, S2 = \{5, 15\}, S4 = \{11\}, S6 = \{35\}$

- $\text{Find}(15) = S2$

- $S1 \leftarrow \text{Union}(S1, S2)$

$S1 = \{2, 10, 5, 15\}, S4 = \{11\}, S6 = \{35\}$

- $\text{Find}(15) = S1$

Array implementation (with parent links)

2	5	10	11	15	35
S1	S2	S3	S4	S5	S6

Initially

S1	S2	S3	S4	S5	S6
S1	S2	S3	S4	S5	S6

$S1 \leftarrow \text{Union}(S1, S3)$

S1	S2	S3	S4	S5	S6
S1	S2	S1	S4	S5	S6

$S2 \leftarrow \text{Union}(S2, S5)$

S1	S2	S3	S4	S5	S6
S1	S2	S1	S4	S2	S6

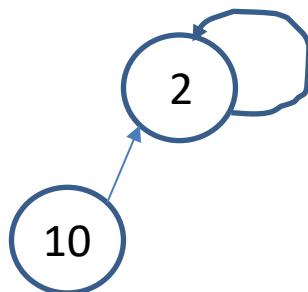
$S1 \leftarrow \text{Union}(S1, S2)$

S1	S2	S3	S4	S5	S6
S1	S1	S1	S4	S2	S6

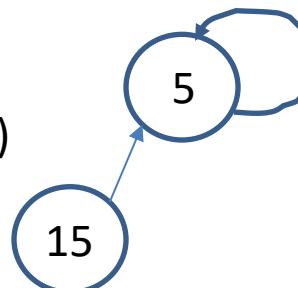
Find(15) : S5 \rightarrow S2 \rightarrow S1

Weighted union

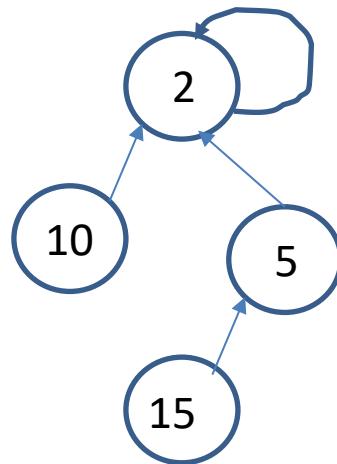
$S1 \leftarrow \text{Union}(S1, S3)$



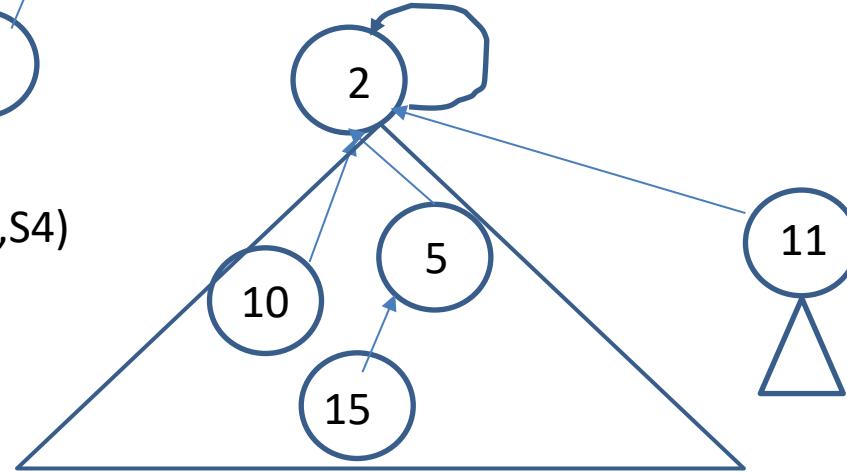
$S2 \leftarrow \text{Union}(S2, S5)$



$S1 \leftarrow \text{Union}(S1, S2)$



$S4 \leftarrow \text{Union}(S1, S4)$



Make root of smaller set
child of root of larger set.

Amortized cost analysis for union-find

- $\text{union}(S1, S2)$ – make smaller sequence elements point to the set node of larger sequence elements
- Complexity of sequence of n operations consisting of union and find starting with singleton sets of n elements.
- Amortization : Accounting method
 - (i) For find operation charge unit cost to operation itself
 - (ii) For union operation, charge unit cost to each of the elements whose links have changed (no cost to operation itself).
- Total amortization cost = # of cost units assigned to elements + # of cost units assigned to Find operation \geq total actual cost
- # of cost units assigned to elements \leq # of times an element can change set
- A set doubles when an element moves, at most $\log n$ times an element can change sets \rightarrow time complexity is $O(n \log n)$

Amortized analysis for a sequence of n (weighted) union-find operations

- Accounts : Find_1, Find_2, Find_3....
Element_1, Element_2,.....Element_n

Amortized cost using accounting method is given as follows:

- (a) Union → if Element_k (root of smaller set) points to Element_p, assign 1 unit cost to Element_k account
- (b) Find_k → assign unit cost to Find_k account
- Note that actual cost of a Find(e) is the path length from e to the root = number of unions performed before this find that causes the element to change its set membership.
- Actual cost of a union operation is unit cost

Total amortized cost of union and finds \geq actual cost of union and finds

- Element e in S1 – find operation finds path from e to root of S1 – path length = p
After $S1 \cup S2$ – find(e) – finds path from e to root of $S1 \cup S2$. Call this path length p1.

When can $p1 = p+1$? $|S1| \leq |S2|$ before merge. After merge , combined set $|S1| + |S2| \geq 2|S1|$ e has moved to a set which is double the size of the set it belonged to before

Max. # of times path length increases for e = 1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow n == log n times

- Each element account charged at most log n units

Actual cost of union and finds \leq Total amortized cost of union and finds $\leq n + n \log n$

Efficient Union-Find DS

- Use a tree for a set where root node identifies the set and each child node has a parent link. Root node's parent link points to itself.
- Union – make the root of the tree for one set a child node of root of another. – Takes $O(1)$ time
- To make find op more efficient, make root of smaller height (rank) set child of root of bigger height (rank) set – keep track of # of nodes in set

Let $S(h)$ – minimum size of set with a tree of height h

$S(h) \geq 2 S(h-1)$, $h \geq 1$, $S(0) = 1 \rightarrow S(h) \geq 2^h \rightarrow h \leq \log n$, number of elements in partition sets

Complexity of Find is $O(\log n)$

- Total complexity of a sequence of n union-find operations is $O(n \log n)$
- Can we do better ?

Efficient Union-Find DS

- Due to Robert E. Tarjan
- Use Union by rank – Similar to union by weight by making root of set of smaller rank a child of root of set of larger rank.

MakeSet(x):

```
x.parent ← x; x.rank ← 0
```

Union(x,y):

```
if x.rank > y.rank  
    y.parent ← x  
else  
    x.parent ← y  
    if x.rank = y.rank  
        y.rank ← y.rank+1
```

Efficient Union-Find DS (contd.)

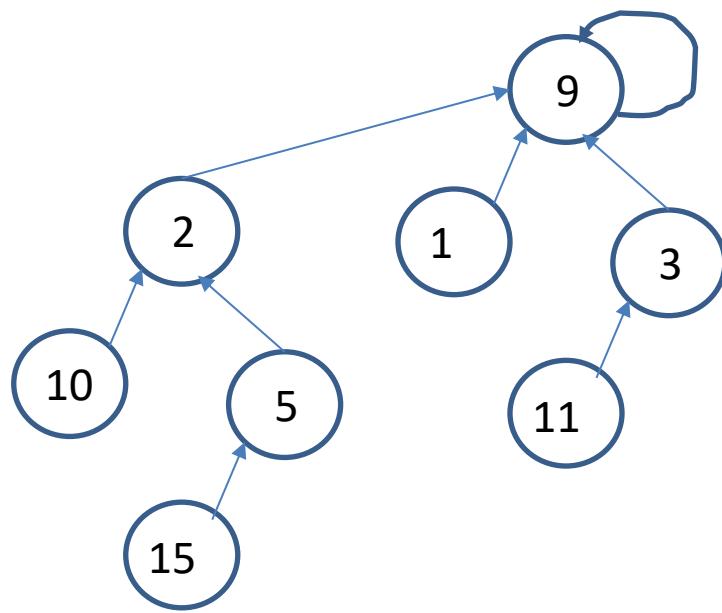
- Use path-compression – After a `find(e)`, make all nodes in path from `e` to root children of the root. Should take same order of time as without compression.

Find(`e`):

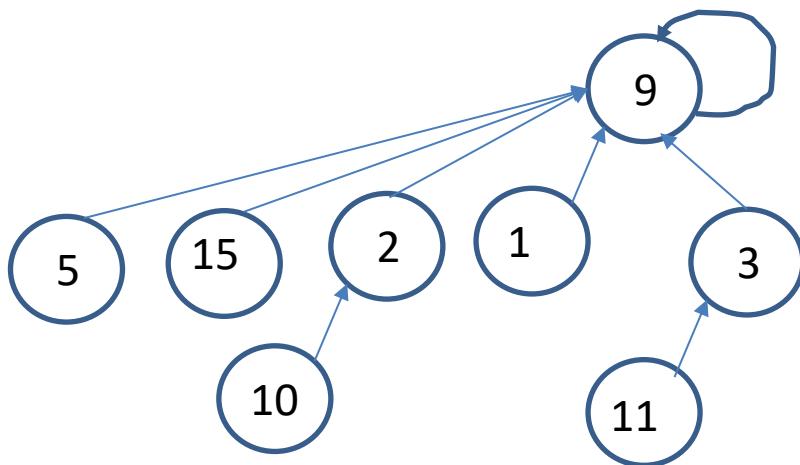
```
if e ≠ e.parent
    e.parent ← Find(e.parent)
return e.parent
```

- Recursion down finds path from element to root and unwinding recursion sets parent of all elements in the path to the root.
- Run-time : shown to be almost linear in the worst-case.

Efficient Union-Find (path-compression)



Before `Find(15)`



After `Find(15)`

Efficient Union-Find DS Complexity

- Ackerman functions (fast-growing) :

$$A_0(n) = 2n, \quad n \geq 0$$

$$A_i(n) = A_{(i-1)}(A_i(n-1)), \quad n > 0 \text{ and } A_i(0) = 1 \text{ for } i > 0$$

A_0 seq : 0, 2, 4, 6,.....

A_1 seq : $2^0, 2^1, 2^2, 2^3, \dots$

A_2 seq (modified, call it F) : 1, 2, $2^2, 2^{2^2}, 2^{2^{2^2}}, \dots$

- Inverse F function I (very-slowly growing):

$$\log^*(n) = \min\{i > 0 : F(i) \geq n\}$$

$$\log^*(k) = 1, k = 1, 2 ; \log^*(k) = 2, k = 3, 4, 5, \dots 16$$

$$\log^*(k) = 3, \quad 17 \leq k \leq 2^{16}, \quad \log^*(65536) = 4$$

- We will use amortized analysis to show that sequence of n union-finds will take $O(n \log^* n)$

Offline MIN problem

Given: A sequence of two types of instructions:

- (a) insert(k) – insert an integer k , $1 \leq k \leq n$, in set T
- (b) extract_min – get and remove min value from T

Required: Sequence of values output by extract_min instructions

- Simple solution – Treat it as online problem.
Keep values in a heap, extract min when required. May take $O(n \log n)$ where n is number of insert instructions.
- Better solution – use union-find data structure. Takes $O(n \log^* n)$ time
Let I_j be the set of insert operations before j -th extract_min in the sequence, $1 \leq j \leq m$ where m is number of extract_min instructions.
There is a total of n insert operations.

Sequence $S : I_1 \ E_1 \ I_2 \ E_2 \ \dots \ I_m \ E_m \ I_{m+1}$ (I_j may be empty)

Approach: Create key sets K_j 's corresponding to I_j 's.

For each i from 1 to n , find K_j i belongs to. If $j \leq m$, then output of E_j must be “ i ”.

We can merge K_j to following adjacent K_l as they are candidates for E_l

Off line min example

Insert 3, Insert 5, ExtractMin_1, Insert 4, Insert 1, ExtractMin_2, Insert 6, Insert 2, Insert 7, ExtractMin_3

Initially form insert sets i.e. Set of inserts before each extract, Final insert set follows last extractMin which can be empty.

$$I_1 = \{\text{Ins. 3, Ins. 5}\}, I_2 = \{\text{Ins. 4, Ins. 1}\}, I_3 = \{\text{Ins. 6, Ins. 2, Ins. 7}\}, I_4 = \emptyset$$

$$K_1 = \{3,5\}, K_2 = \{4,1\}, K_3 = \{6,2,7\}, K_4 = \emptyset$$

For each element from 1 to n we try to find if it is the result of an extract min op.

For element 1 : What key set 1 belongs to ? K_2 . i.e $\text{Find}(1) = 2$ Hence $\text{ExtractMin}_2() = 1$

Then we merge K_2 and its adjacent set $K_3 \leftarrow \text{Union}(K_3, K_2) = \{4,1,6,2,7\}$

What key set 2 belongs to ? $\text{Find}(2) = K_3 \rightarrow \text{ExtractMin}_3() = 2$

$K_4 \leftarrow \text{Union}(K_4, K_3) = \{4,1,6,2,7\}$

What key set 3 belongs to ? $\text{Find}(3) = K_1 \rightarrow \text{ExtractMin}_1() = 3$

$K_4 \leftarrow \text{Union}(K_4, K_1) = \{3,5,4,1,6,2,7\}$

What key set 4 belongs to ? $\text{Find}(4) = K_4 \rightarrow$ will not be result of any ExtractMin

We conclude the same for Elements 5, 6 and 7.

Weighted graph problems

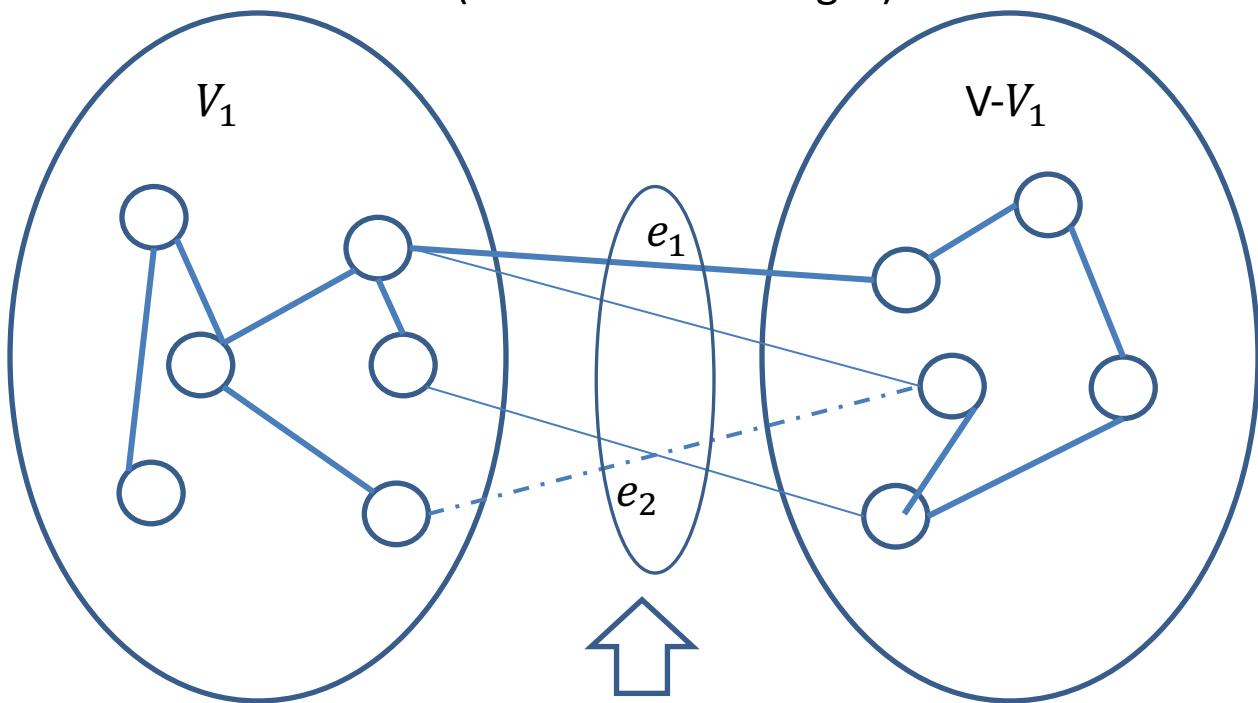
Minimum cost spanning tree

- **Given :** Undirected connected graph $G = (V, E)$ with weights (costs) on edges ($w : E \rightarrow \mathbb{R}$)
- **Required:** Minimum cost spanning tree (tree with min. sum of costs of edges)
- Useful in network designs
- **Greedy approach basis:**

If $(V_1, V - V_1)$ is a cut and E_1 is the cut-set. If $w(e_1) = \min_{e \in E_1} w[e]$ then there is a min cost spanning tree that includes e_1 .

- Two popular greedy algorithms:
 - (a) Kruskal's algorithm (edge addition to tree)
 - (b) Prim-Jarnik algorithm (vertex addition to tree)

T is a spanning tree
(shown as thick edges)



Cut set E_1

$$w(e_2) = \min_{e \in E_1} w(e)$$

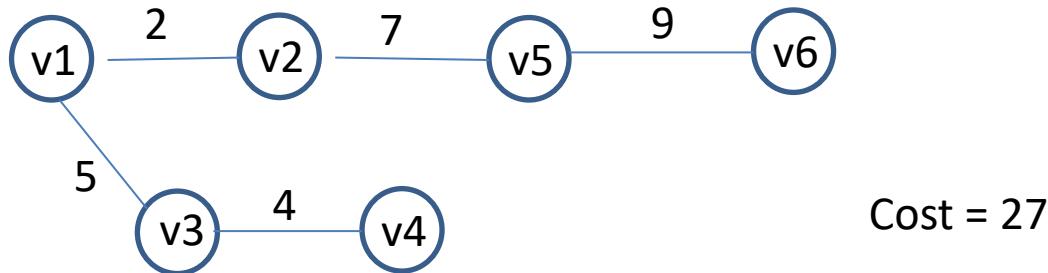
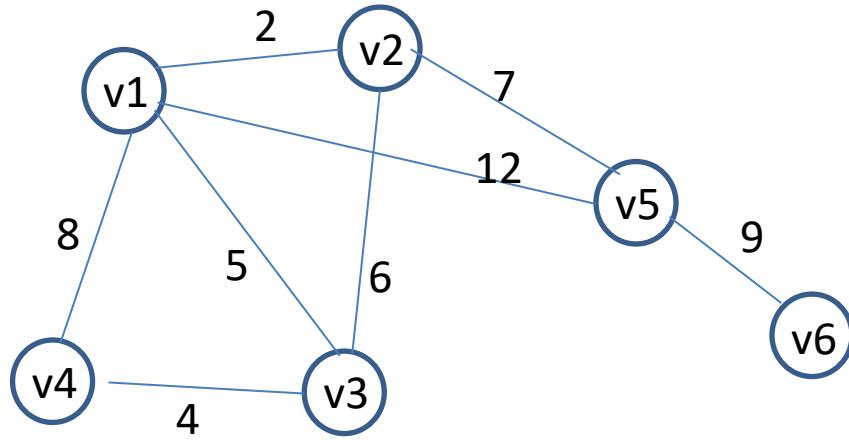
$$T_1 = (T - \{e_1\}) \cup \{e_2\}$$

$$C(T_1) = C(T) - w(e_1) + w(e_2) \leq C(T)$$

Kruskal's MST algorithm

- Idea:
 - At each step we find a minimum cost edge that connects a vertex in one spanning tree to another in the forest.
 - This greedy choice property is possible
 - Initially each vertex by itself is a spanning tree in the forest
 - We choose in non-decreasing order of edge weights
 - (a) if the edge connects vertices in same tree, ignore it
 - (b) else choose the edge and reduce number of spanning trees by 1

Kruskal's MST algorithm example



Kruskal's algorithm time complexity

- Use a min-heap priority queue for edges using $w(e)$
- At each step $O(\log m)$ time to find min cost edge, m number of edges

Total number of steps $\leq m$

- Use a disjoint set union-find DS for spanning forest
- n makeSet() operations, n is number of vertices
- set find to detect if vertices of an edge are in same spanning tree
- set union merge spanning trees

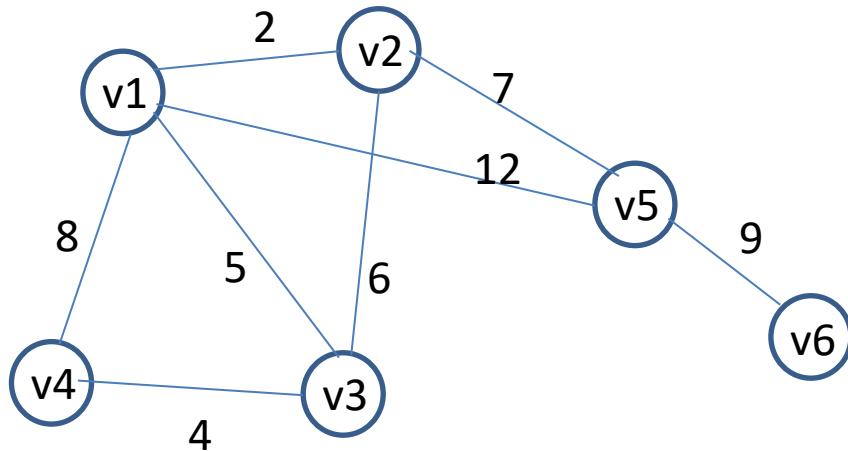
Time complexity of union-finds – $O(m \log *m)$

- Total time complexity is $O(n + m \log m)$

Prim-Jarnik's MST algorithm

- Idea:
 - Start from a vertex and grow a spanning tree
 - Maintain $D[v]$, min cost of an edge from v to a vertex in existing spanning tree; Initially $D[v]$ set to $+\infty$
 - At each step choose vertex u with minimum $D[v]$ to be added to the spanning tree
 - This greedy choice property possible
 - Then update $D[w]$ for each vertex not in spanning tree that is adjacent to u

Prim-Jarnik's Algorithm example



D[v]

Add v5;
update
v1,v2,v6

	v1	v2	v3	v4	v5	v6
	12	7	∞	∞	0 (*)	9

Add v2
update v1,v3

	v1	v2	v3	v4	v5	v6
	2	7(*)	6	∞	0(*)	9

Add v1
update v4,v3

	v1	v2	v3	v4	v5	v6
	2(*)	7(*)	5	8	0(*)	9

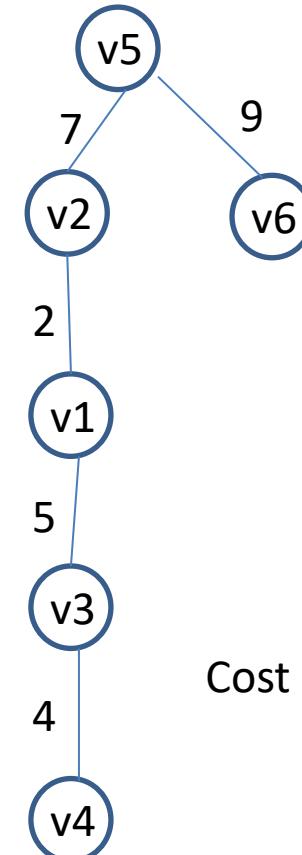
Add v3
update v4

	v1	v2	v3	v4	v5	v6
	2(*)	7(*)	5 (*)	4	0(*)	9

Add v4; no
updates

	v1	v2	v3	v4	v5	v6
	2(*)	7(*)	5 (*)	4 (*)	0(*)	9(*)

Add v6



Prim's MST algorithm time complexity

- Use min-heap priority queue for vertices : $D[v]$ is heap key
- At each step extract-min to find one with min $D[v]$
This takes $O(\log n)$ time
- Total time for extract-min – $O(n \log n)$
- Updating $D[v]$ takes $O(\log n)$ time, n number of vertices
Total number of updates $\leq m$, number of edges
- Total time for updates – $O(m \log n)$
- Total time complexity – $O((m+n) \log n)$ which is $O(m \log n)$ time.

Single source shortest path problem

- **Given :** A directed graph $G = (V, E)$ with weights on edges ($w : E \rightarrow \mathbb{R}$) and a source vertex $s \in V$
- **Required:** Find the shortest path length $D[v]$ from s to v for all $v \in V$
- Example : shortest route between cities
Length of path = sum of weights of edges in the path
- Negative weights with cycles can cause shortest path between vertices to have infinite number of edges (not converge)
- Focus on non-negative weights on edges

Dijkstra's shortest path alg.

- **Cut** – Partition of V into disjoint subsets V_1 and V_2 ($V_2 = V - V_1$)
Cut-set – $\{(u, v) \in E \mid u \in V_1 \text{ and } v \in V_2\}$
- **Greedy approach idea :**
 - (a) Have a cut where V_1 is set of vertices we know shortest path length from s ; initially $V_1 = \{s\}$
 - (b) For all $v \in V_2$, we know shortest path length from s to v with vertices in path (except v) restricted to V_1 (at most one edge in cut-set). Let it be $D[v]$
 - (c) At each step we can move one vertex from V_2 to V_1 if V_2 is not empty. Why ?

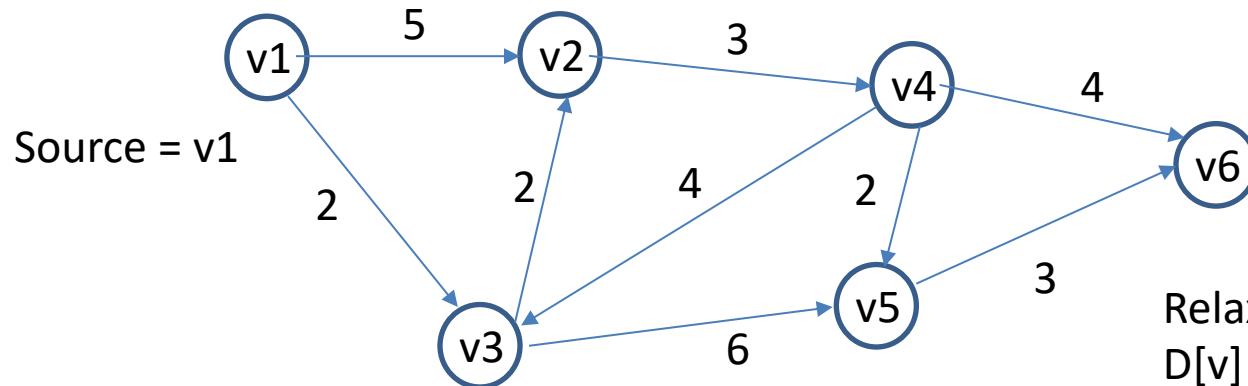
Let $D[u] = \min_{v \in V_2} D[v]$.

Cannot have a better path from s to u that includes a vertex from V_2
→ $D[u]$ is shortest path from s to u

Move u to V_1 and also update $D[v]$ for all other v in V_2 . How ?

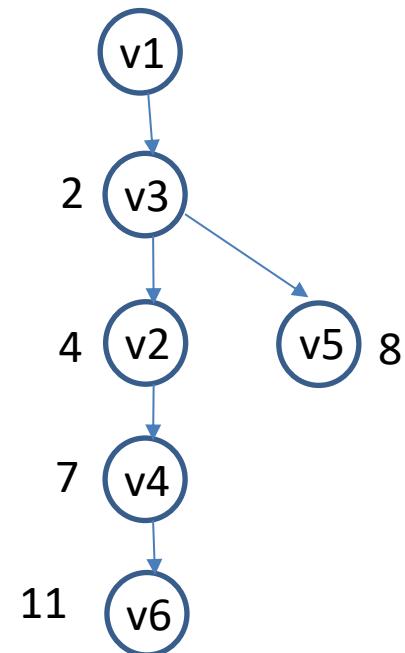
Similar to Prim's MST alg. Only need to consider paths that go through u

Dijkstra's algorithm example



Relaxation :
 $D[v] \leftarrow \min(D[v], D[u] + w((u,v)))$

$D[v]$	v_1	v_2	v_3	v_4	v_5	v_6
Add v_1 ; update v_2, v_5, v_6	0(*)	5	2	∞	∞	∞
Add v_3 update v_2, v_5	0(*)	4	2(*)	∞	8	∞
Add v_2 update v_4	0(*)	4 (*)	2(*)	7	8	∞
Add v_4 update v_6	0(*)	4(*)	2 (*)	7(*)	8	11
Add v_5 ; no updates	0(*)	4(*)	2 (*)	7 (*)	8(*)	11
Add v_6						



Shortest path alg. time complexity

- Min-Heap priority queue for vertices in V2 based on $D[v]$
- $O(\log n)$ to get min $D[v]$
- number of extract-min ops $\leq n$, number of nodes
- For a vertex v added to V1, need to update $D[w]$ only for each edge (v,w) ; each heap update $O(\log n)$
- Total updates $\leq m$, number of edges
- Total time complexity – $O(n \log n + m \log n)$
- Since for a connected graph $m \geq n-1$, time complexity for a connected graph is $O(m \log n)$

Bellman Ford algorithm

- Finds single-source shortest paths even when there are negative edges
- Can identify negative weight cycles
- Works by iterating at most $n-1$ times
- During each iteration look at all edges (u,v) and update $D[v]$ to better value as in Dijkstra's algorithm if possible, i.e. $D[v] = \min(D[v], D[u]+w((u,v)))$ This is called “relaxation”.
- Total time-complexity is $O(nm)$

All pair path problems

Closed semi-rings

- $(S, \circ, 1)$ is a monoid if
 - (a) S is closed under \circ ($a \circ b \in S, \forall a, b \in S$)
 - (b) \circ is associative ($a \circ (b \circ c) = (a \circ b) \circ c$)
 - (c) 1 is identity for \circ ($a \circ 1 = 1 \circ a = a$)
- $(S, +, \circ, 0, 1)$ is a closed semi-ring if
 - (a) $(S, +, 0)$ is a monoid
 - (b) $(S, \circ, 1)$ is a monoid and 0 is annihilator for \circ (i.e $a \circ 0 = 0 \circ a = 0, \forall a$)
 - (c) $+$ is commutative ($a + b = b + a$) and idempotent ($a + a = a$)
 - (d) \circ distributes over $+$ i.e. $a \circ (b+c) = (a \circ b) + (a \circ c)$
 - (e) Associative, distributive, commutative and idempotency properties extend to finite and infinite sums

$$\sum_i a_i \circ \sum_j b_j = \sum_{i,j} a_i \circ b_j = \sum_i (\sum_j a_i \circ b_j)$$

Note infinite sum $\sum_i a_i$ exists and is unique (idempotency)

Examples of closed semi-rings

- Boolean algebra : $(\{0,1\}, \vee, \wedge, 0, 1)$ is a closed semi-ring
 \wedge distributes over \vee , \vee is idempotent ≥ 0 and 0 is annihilator for \wedge
- $(R_{\geq 0} \cup \{+\infty\}, \text{MIN}, +, +\infty, 0)$ is a closed semi-ring (here + is arithmetic addition)
 - $(R_{\geq 0} \cup \{+\infty\}, \text{MIN}, +\infty)$ is a monoid
 - $(R_{\geq 0} \cup \{+\infty\}, +, 0)$ is a monoid and $+\infty$ is annihilator for + ($a + \infty = +\infty$)
 - + distributes over MIN : $a + \text{MIN}(b,c) = \text{MIN}(a+b, a+c)$
 - Infinite sum $\text{MIN}(a_1, \text{Min}(a_2, \dots)) = \text{Min}(a_1, a_2, a_3, \dots)$ exists and is unique
- $(F_\Sigma, \cup, \circ, \emptyset, \{\epsilon\})$ is a closed semi-ring where
 - F_Σ is the family of sets of finite length strings from alphabet Σ including empty string ϵ (countably infinite sets)
 - \cup - set union, associative, identity empty set \emptyset
 - \circ - concatenation of sets $S_1 \circ S_2 = \{ xy \mid x \in S_1 \text{ and } y \in S_2\}$; \emptyset is annihilator for \circ
 - \circ distributes over \cup : $S_1 \circ (S_2 \cup S_3) = (S_1 \circ S_2) \cup (S_1 \circ S_3)$

All pairs path problem

Given : A directed graph $G = (V, E)$ with possible self-cycles and a label function $l : E \rightarrow S$ where $(S, +, \circ, 0, 1)$ is a closed semi-ring

- For a directed path $p = (e_1, e_2, \dots, e_n)$, path product $l(p) = l(e_1) \circ l(e_2) \circ \dots \circ l(e_n)$
- Sum of two path products p_1 and $p_2 = l(p_1) + l(p_2)$
- $S(u, v)$ is sum of product of all paths from u to v in the graph

Required: Find $S(u, v)$ for all pairs of vertices in G .

All pair path problems (contd.)

All pairs path problem

Given : A directed graph $G = (V, E)$ with possible self-cycles and a label function $l : E \rightarrow S$ where $(S, +, \circ, 0, 1)$ is a closed semi-ring

- For a directed path $p = (e_1, e_2, \dots, e_n)$, path product $l(p) = l(e_1) \circ l(e_2) \circ \dots \circ l(e_n)$
- Sum of two path products p_1 and $p_2 = l(p_1) + l(p_2)$
- $S(u, v)$ is sum of product of all paths from u to v in the graph

Required: Find $S(u, v)$ for all pairs of vertices in G .

Closure in closed semi-rings

- Define a closure operation * for a closed semi-ring $(S, +, \circ, 0, 1)$ element a as follows:

$$a^* = 1 + a + a \circ a + a \circ a \circ a + \dots = 1 + a + a^2 + a^3 + \dots$$

By infinite idempotency of $+$, this infinite sum exists and is unique.

- For $(\{0,1\}, \vee, \wedge, 0, 1)$,

$$a^* = 1 \vee a \vee a^2 \dots = 1 \text{ for } a = 0 \text{ or } 1$$

- For $(R_{\geq 0} \cup \{+\infty\}, \text{MIN}, +, +\infty, 0)$,

$$a^* = \text{MIN}(0, a, 2a, 3a, \dots) = 0 \text{ for } a \in R_{\geq 0} \cup \{+\infty\}$$

- For $(F_{\Sigma}, \cup, \circ, \emptyset, \{\epsilon\})$,

$$S^* = \{\epsilon\} \cup S \cup (S \circ S) \cup (S \circ S \circ S) \dots = \bigcup_{i \geq 0} \{x_1 x_2 \dots x_i \mid x_j \in S, 1 \leq j \leq i\}$$

Closed semi-ring matrices

- Define M_n be set of $n \times n$ matrices where elements are from a closed semi-ring $(S, +, \circ, 0, 1)$
- $(M_n, +_n, *_n, 0_n, I_n)$ is a closed semi-ring
 - $+_n$ - addition of $n \times n$ matrices, ($+$ is closed semi-ring idempotent operation)
 - $*_n$ - multiplication of $n \times n$ matrices, ($+, \circ$ closed semi-ring operations) ; distributes over $+_n$
 - 0_n - $n \times n$ matrix with all 0 (identity for $+$)
 - I_n - $n \times n$ identity matrix with 0 identity for $+$ and 1 identity for \circ - 0_n is annihilator for $*_n$
- Infinite sum of matrices exists and is unique

Digraph Matrix Closure

- For a digraph with n vertices, define an $n \times n$ matrix L where $L(i,j) = I((v_i, v_j))$
- L matrix is an element of closed semi-ring $(M_n, +_n, *_n, 0_n, I_n)$
- Define $L^k = L *_n L \dots$ multiplied k times. $L^0 = I_n$
- What does $L^2(i,j)$ indicate ?

$$L^2(i,j) = L(i,1) \circ L(1,j) + L(i,2) \circ L(2,j) + \dots + L(i,n) \circ L(n,j)$$

Sum of products of paths of length 2 from v_i to v_j

- L^k matrix gives for all vertex pairs sum of k length path products between these vertices
- Closure matrix $L^* = \sum_{k=0}^{\infty} L^k$ exists and is unique where sum is $+_n$ and $L^0 = I_n$
- It gives exactly what we need in all pairs-path problem.

DP algorithm for all-pair paths

- Due to Floyd-Warshall
- Let $D_w(u,v)$ = sum of path products from u to v that go through w

$$= S_1(u,w) \circ S_2(w,w) \circ S_3(w,v) \text{ where}$$

$S_1(u,w)$ = sum of paths from u to w that do not go through w

$S_1(w,w)$ = sum of paths from w to w

$S_3(w,v)$ = sum of paths from w to v that do not go through w

Distributivity of \circ over $+$ for finite and infinite sums

- Define $D_k(i,j)$ as sum of paths from v_i to v_j that do not go through vertices other than v_1, v_2, \dots, v_k

Floyd-Warshall Algorithm

AllPair(G, I):

Input : Digraph G = (V,E) with vertex numbered 1,2..n arbitrarily and a labeling function I : E → S where (S, +, ∘, 0, 1) is a closed semi-ring

Output : Compute L^* matrix

for i ← 1 to n

 for j ← 1 to n

 I ← $(v_i, v_j) \in E ? I((v_i, v_j) : 0$

$D_o(i,j) \leftarrow i = j ? 1 + I : I - - (1)$

for k ← 1 to n

 for i ← 1 to n

 for j ← 1 to n

$D_k(i,j) = D_{k-1}(i,j) + D_{k-1}(i,k) \circ (D_{k-1}(k,k))^* \circ D_{k-1}(k,j) --$

(2)

Return D_n

Floyd-Warshall alg. time complexity

- Initialization takes $O(n^2)$ operations (assignment)
- There are n iterations of matrix computations
- Each iteration computes n^2 values
- Each computation of matrix value requires constant number of closed semi-ring ops $+$, \circ and $*$
- Total complexity is $O(n^3)$ closed semi-ring ops $+$, \circ and $*$

Transitive Closure Algorithm

- **Problem :** Given a directed graph $G = (V, E)$, return a $n \times n$ matrix T where $T[i, j] = 1$ if there exists a directed path from v_i to v_j , 0 otherwise
- Solve all-pairs problem where closed semi-ring is : $(\{0,1\}, \vee, \wedge, 0, 1)$ with adjacency matrix A . D_n gives value of T
- Step 1 becomes :

$$D_0(i, j) \leftarrow i = j ? 1 : A[i, j]$$

- $0^* = 1^* = 1$ so we can remove the closure element in the algorithm.
- Hence step 2 becomes :
$$D_k(i, j) = D_{k-1}(i, j) \vee (D_{k-1}(i, k) \wedge D_{k-1}(k, j))$$
- Using Floyd-Warshall alg, we can compute it in $O(n^3)$ semi-ring operations \vee, \wedge .
- Another way is to compute A^* matrix is to compute A^n which requires $O(n^3 \log n)$ time. Think about path doubling.

All-pair shortest path alg.

- **Problem** : Given a directed graph $G = (V, E)$ and a length function $w : E \rightarrow R_{\geq 0}$, return a $n \times n$ matrix D where $D(i,j) = \text{length of shortest path from } i, j$
- Solve all-pairs problem where closed semi-ring is : $(R_{\geq 0}, \text{MIN}, +, +\infty, 0)$ and matrix L where $L[i,j]$ is length of edge (v_i, v_j) if it exists else $+\infty$. D_n gives value of D
- Step 1 becomes (note $\text{Min}(0, L[i,i]) = 0$):

$$D_0(i,j) \leftarrow i = j ? 0 : L[i,j]$$

- Note that $a^* = \text{MIN}(0, a, 2a, \dots) = 0$. So we can omit it in step 2
- Step2 then becomes:

$$D_k(i,j) = \text{Min}(D_{k-1}(i,j), (D_{k-1}(i,k) + D_{k-1}(k,j)))$$

- Using Floyd-Warshall alg, we can compute it in $O(n^3)$ semi-ring operations MIN , $+$.

Closed semi-ring matrix closure

- Given a $n \times n$ matrix A of elements from a closed semi-ring $(S, +, \circ, 0, 1)$, the closure matrix $A^* = \sum_{k=0}^{\infty} A^k$ can be computed by divide-and-conquer strategy
- Let $A = \begin{bmatrix} B & C \\ D & E \end{bmatrix}$ where B, C, D and E are $n/2 \times n/2$ submatrices;
Let V_1 and V_2 be partition of vertices
- Then $A^* = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix}$
 W = closure of paths that either (i) stay in V_1 or (ii) stay in V_1 for some time, jump to V_2 , stay in V_2 for sometime and then jump back to V_1 , this pattern repeated 1 or more time
 $= (B + C \cdot E^* \cdot D)^*$ -- + and . are $n \times n$ matrix semi-ring operations
 X = closure of paths that start in V_1 and end in V_2
 $= W \cdot C \cdot E^*$

Closed semi-ring matrix closure

Y = closure of paths that start in V_2 and end in V_1

$$= E^* \cdot D \cdot W$$

$$Z = E^* + Y \cdot C \cdot E^*$$

If we have $T = C \cdot E^*$, then

$$W = (B + T \cdot D)^*$$

$$X = W \cdot T$$

$$Y = E^* \cdot D \cdot W$$

$$Z = E^* + Y \cdot T$$

Requires 2 recursive $n/2 \times n/2$ matrix closures and 6 $n/2 \times n/2$ matrix multiplications and 2 $n/2 \times n/2$ matrix additions

Time complexity $T(n)$ – number of closed-semi ring operations

$$T(n) \leq 2 T(n/2) + c g(n), n > 1 \text{ and } T(1) = b \text{ for constants } b \text{ and } c$$

$g(n)$ time to multiply 2 $n/2 \times n/2$ matrices

$T(n)$ is $O(g(n))$ provided $g(n)$ satisfies $g(2n) \geq 4 g(n)$ (normally the case)

→ Matrix closure can be computed in the same order of time as matrix multiplication using closed semi-ring.