CSE408
Divide and Conquer

Divide-and-Conquer

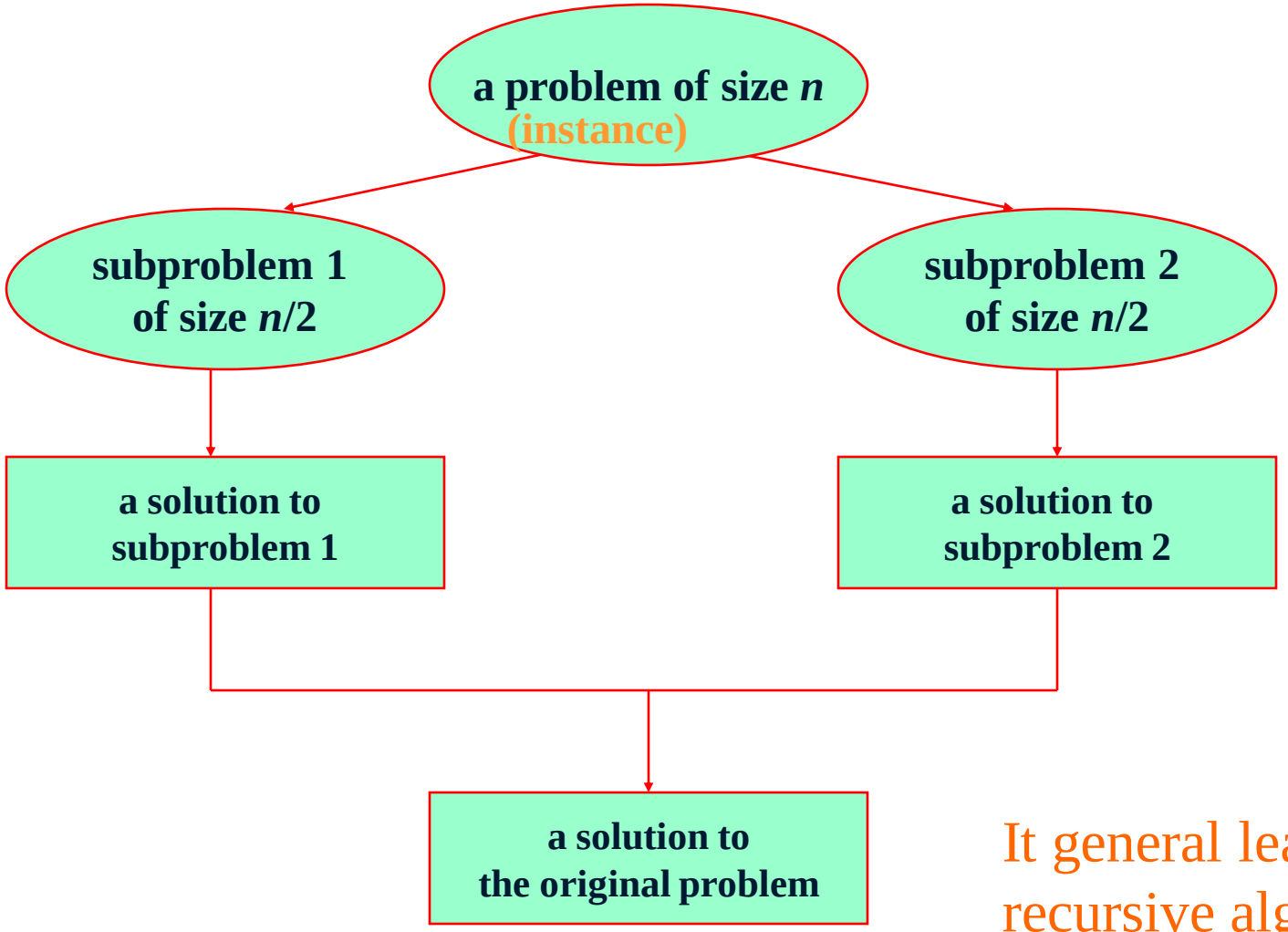
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)



It general leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ Θ(n^2)

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ Θ(n^2log n)

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ Θ(n^3)

Mergesort

- Split array A[0..*n*-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

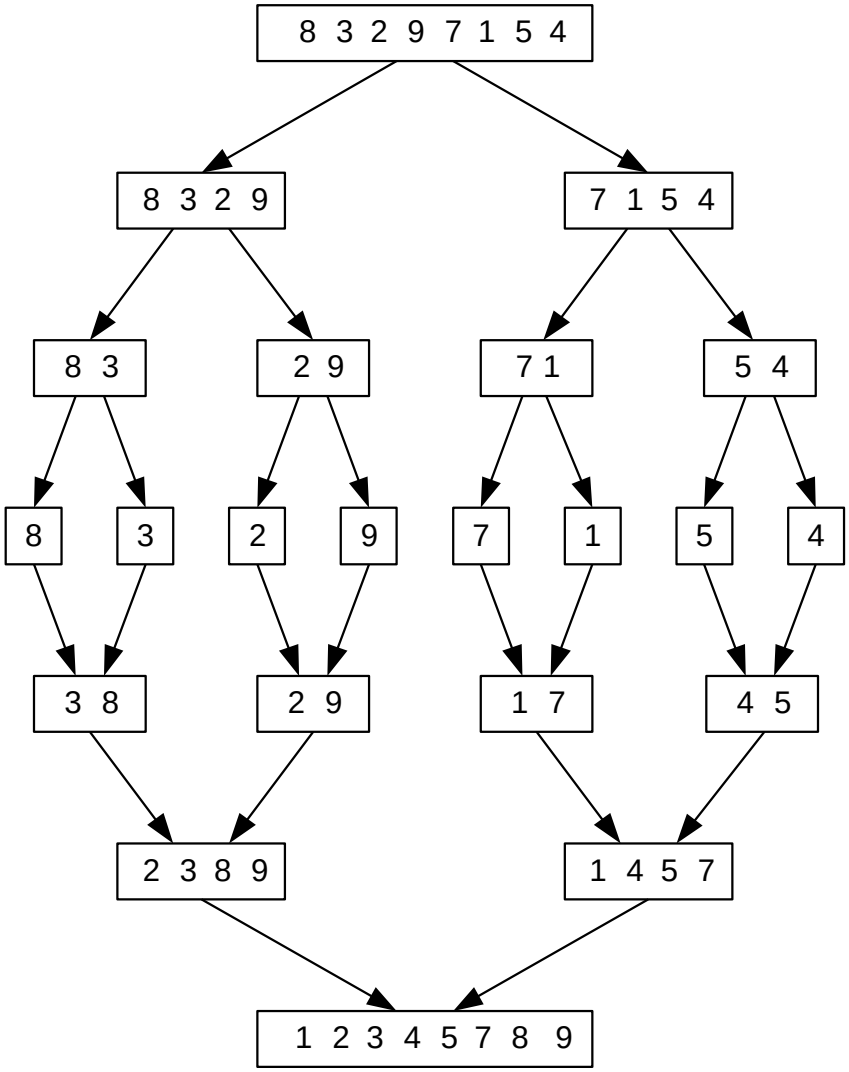
Pseudocode of Merge

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Time complexity: $\Theta(p+q) = \Theta(n)$ comparisons

Mergesort Example



The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$

$$T(n) = 2T(n/2) + \Theta(n),\ T(1) = 0$$

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

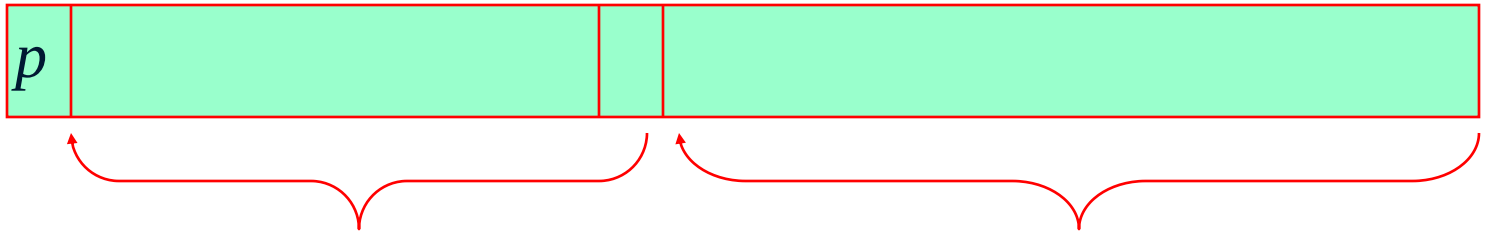
$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (not in-place)

- Can be implemented without recursion (bottom-up)

Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Partitioning Algorithm

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] < p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

or $i > r$

or $j = l$

Time complexity: $\Theta(r-l)$ comparisons

Quicksort Example

5 3 1 9 8 2 4 7

5

2 5 8

1 2 3 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$ $T(n) = T(n-1) + \Theta(n)$
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Divide: Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

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

The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
1  if p < r
2        q = PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```


i
p,j
r
(a)
2 8 7 1 3 5 6 4
p,i
j
r
(b)
2 8 7 1 3 5 6 4
p,i
j
r
(c)
2 8 7 1 3 5 6 4
p,i
j
r
(d)
2 8 7 1 3 5 6 4
p
i
j
r
(e)
2 1 7 8 3 5 6 4
p
i
j
r
(f)
2 1 3 8 7 5 6 4
p
i
j
r
(g)
2 1 3 8 7 5 6 4
p
i
r
(h)
2 1 3 8 7 5 6 4
p
i
r
(i)
2 1 3 4 7 5 6 8

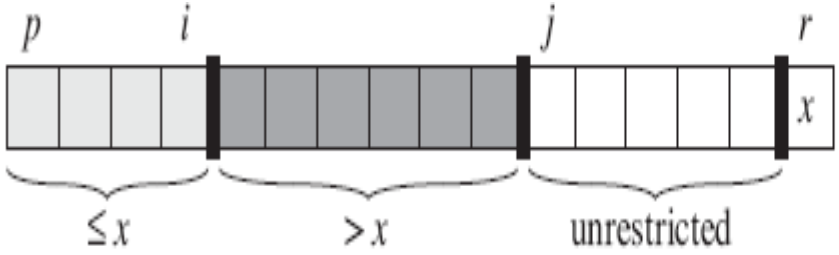
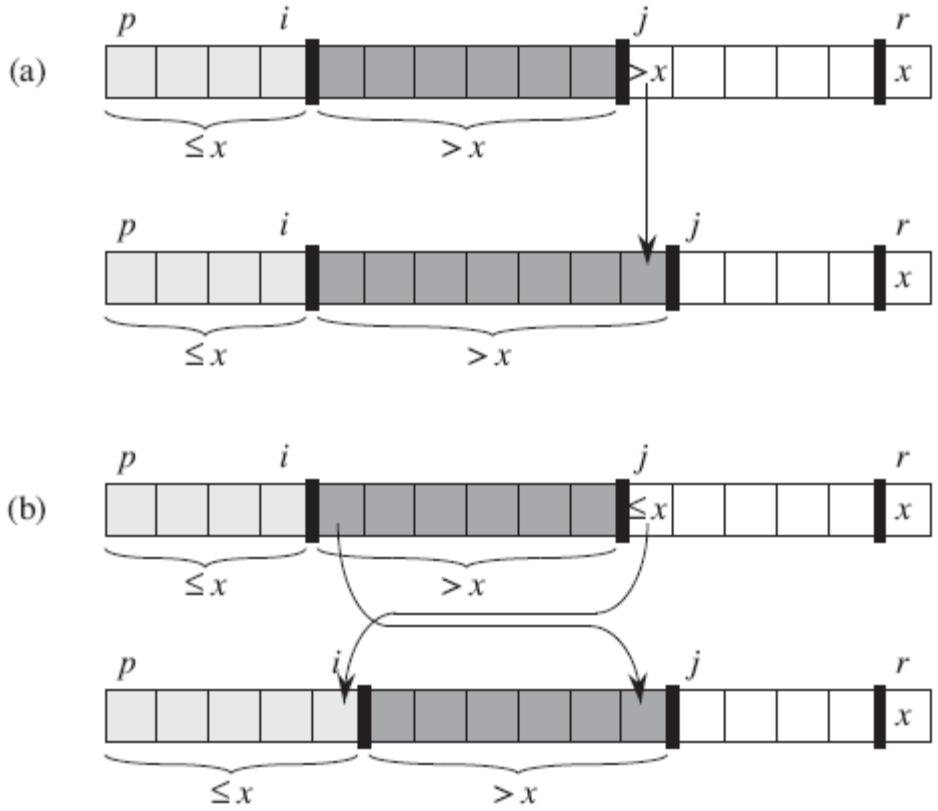


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j..r-1]$ can take on any values.


(a)
p
i
j
r
x
x
≤ x
> x
p
i
j
r
x
≤ x
> x
(b)
p
i
j
r
≤ x
x
≤ x
> x
p
i
j
r
x
≤ x
> x

WORST CASE

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n) \, .
\end{aligned}
$$

Best Case

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

$$\underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad \overset{K}{\updownarrow} A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}.$$

As an example, let us apply binary search to searching for $K = 70$ in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Binary Search

Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ return m
    else if $K < A[m]$ $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1

Analysis of Binary Search

- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$

- Optimal for searching a sorted array

- Limitations: must be a sorted array (not linked list)

- Bad (degenerate) example of divide-and-conquer
  because only one of the sub-instances is solved

- Has a continuous counterpart called *bisection method* for solving equations in one unknown $f(x) = 0$ (see Sec. 12.4)

Binary Tree Algorithms

Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

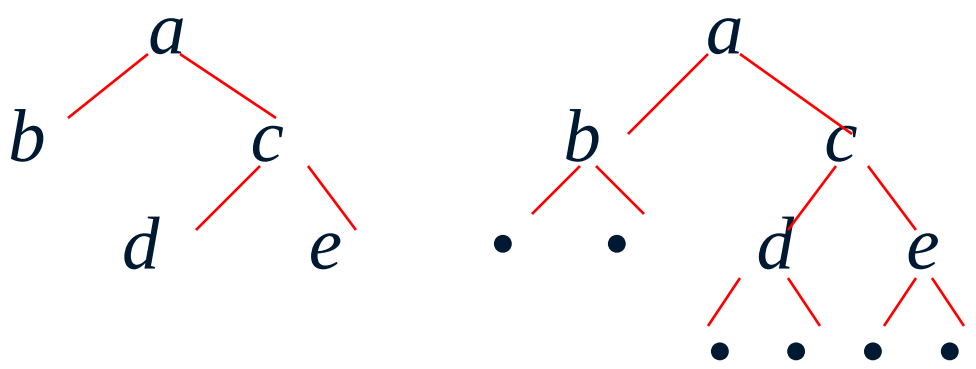
Algorithm *Inorder*(T)

if $T \neq \emptyset$

Inorder(T_{left})

print(root of T)

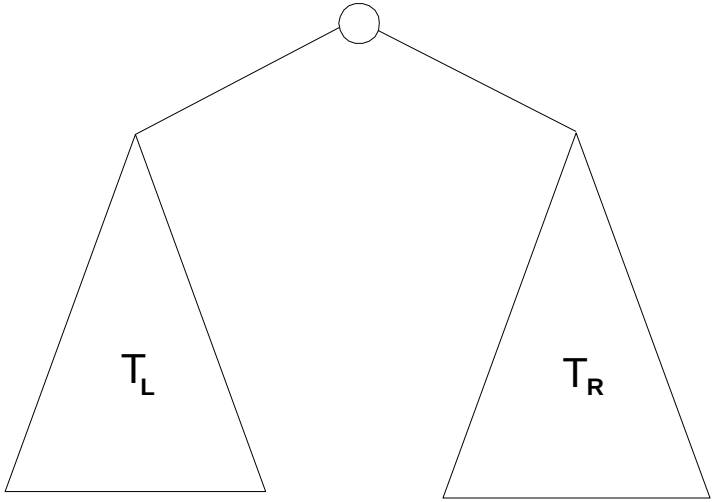
Inorder(T_{right})



Efficiency: $\Theta(n)$. Why?

Binary Tree Algorithms (cont.)

Ex. 2: Computing the height of a binary tree



$h(T) = \max\{h(T_L), h(T_R)\} + 1$ if $T \neq \varnothing$ and $h(\varnothing) = -1$

Efficiency: $\Theta(n)$. Why?

Multiplication of Large Integers

Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{rl}
 & a_1\ a_2 \ldots\ a_n \\
 & b_1\ b_2 \ldots\ b_n \\
\hline
(d_{10})\ & d_{11} d_{12} \ldots\ d_{1n} \\
(d_{20})\ & d_{21} d_{22} \ldots\ d_{2n} \\
\ldots & \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\
\hline
(d_{n0})\ & d_{n1} d_{n2} \ldots\ d_{nn}
\end{array}
$$

Efficiency: $\Theta(n^2)$ single-digit multiplications

First Divide-and-Conquer Algorithm

A small example: A ∗ B where A = 2135 and B = 4014

A = $(21 \cdot 10^2 + 35)$, B = $(40 \cdot 10^2 + 14)$

So, A ∗ B = $(21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if A = A_1A_2 and B = B_1B_2 (where A and B are *n*-digit, A_1, A_2, B_1, B_2 are *n*/2-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications M(*n*):

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

Solution: $M(n) = n^2$

23*14

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$
$$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$
$$= c_2 10^n + c_1 10^{n/2} + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Second Divide-and-Conquer Algorithm

$$\mathbf{A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2}$$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

I.e., $(A_1 * B_2 + A_2 * B_1) = \mathbf{(A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2}$, which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

What if we count both multiplications and additions?

Example of Large-Integer Multiplication

2135 * 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + 35*14

where c1 = (21+35)*(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.

This process requires 9 digit multiplications as opposed to 16.

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2})$$
$$= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k.$$

Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

Conventional Matrix Multiplication

- Brute-force algorithm

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}$$

8 multiplications

4 additions

Strassen's Matrix Multiplication

- Strassen's algorithm for two 2x2 matrices (1969):

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

- $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
- $m_2 = (a_{10} + a_{11}) * b_{00}$
- $m_3 = a_{00} * (b_{01} - b_{11})$
- $m_4 = a_{11} * (b_{10} - b_{00})$
- $m_5 = (a_{00} + a_{01}) * b_{11}$
- $m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
- $m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 multiplications

18 additions

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed in general as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Formulas for Strassen's Algorithm

$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$

Analysis of Strassen's Algorithm

If n is not a power of 2, matrices can be padded with zeros.

What if we count both multiplications and additions?

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ **vs.** n^3 **of brute-force alg.**

Algorithms with better asymptotic efficiency are known but they are even more complex and not used in practice.

CSE408
Recurrence equations

Lecture #12

Substitution method

$T(1) = \Theta(1)$

$O(n^3)$ O Ω

$T(k) \leq ck^3$ $k < n$

$T(n) \leq cn^3$

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

$$(c/2)n^3 - n \ge 0$$

$$c \ge 2 \qquad n \ge 1$$

residual

Example (continued)

Base: $T(n) = \Theta(1)$ $n < n_0$ n_0

$1 \le n < n_0$ $\Theta(1) \le cn^3$

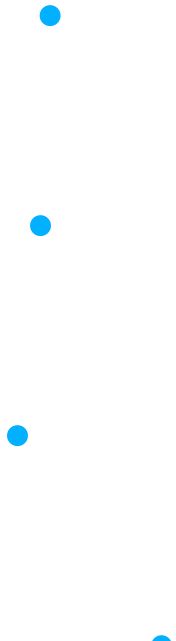
c

Base: $T(n) = \Theta(1)$ $n < n_0$ n_0

$1 \leq n < n_0$ $\Theta(1) \leq cn^3$

c

Recursion-tree method



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

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

$$T(n)$$

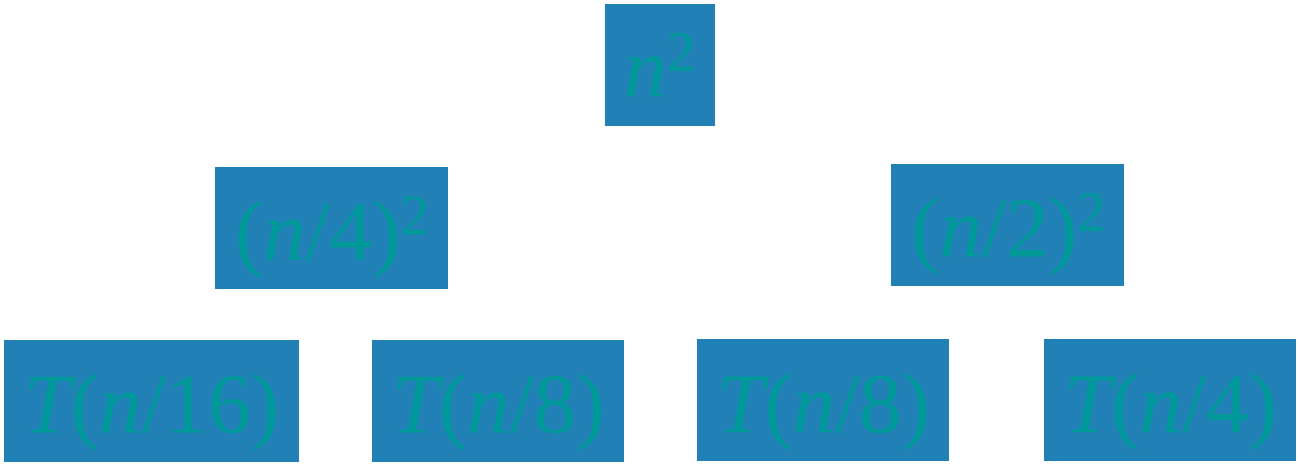
$$T(n) = T(n/4) + T(n/2) + n^2$$

n^2

$T(n/4)$

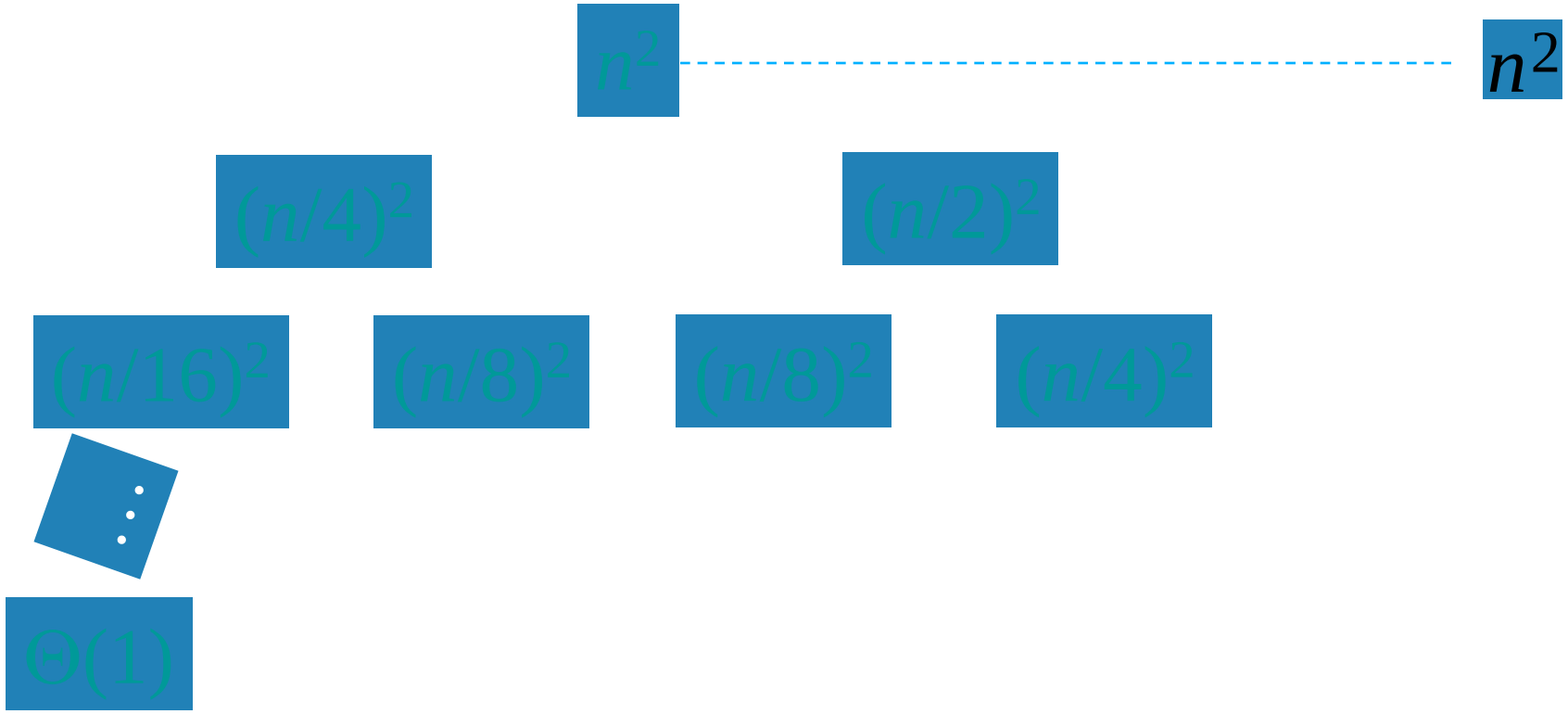
$T(n/2)$

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

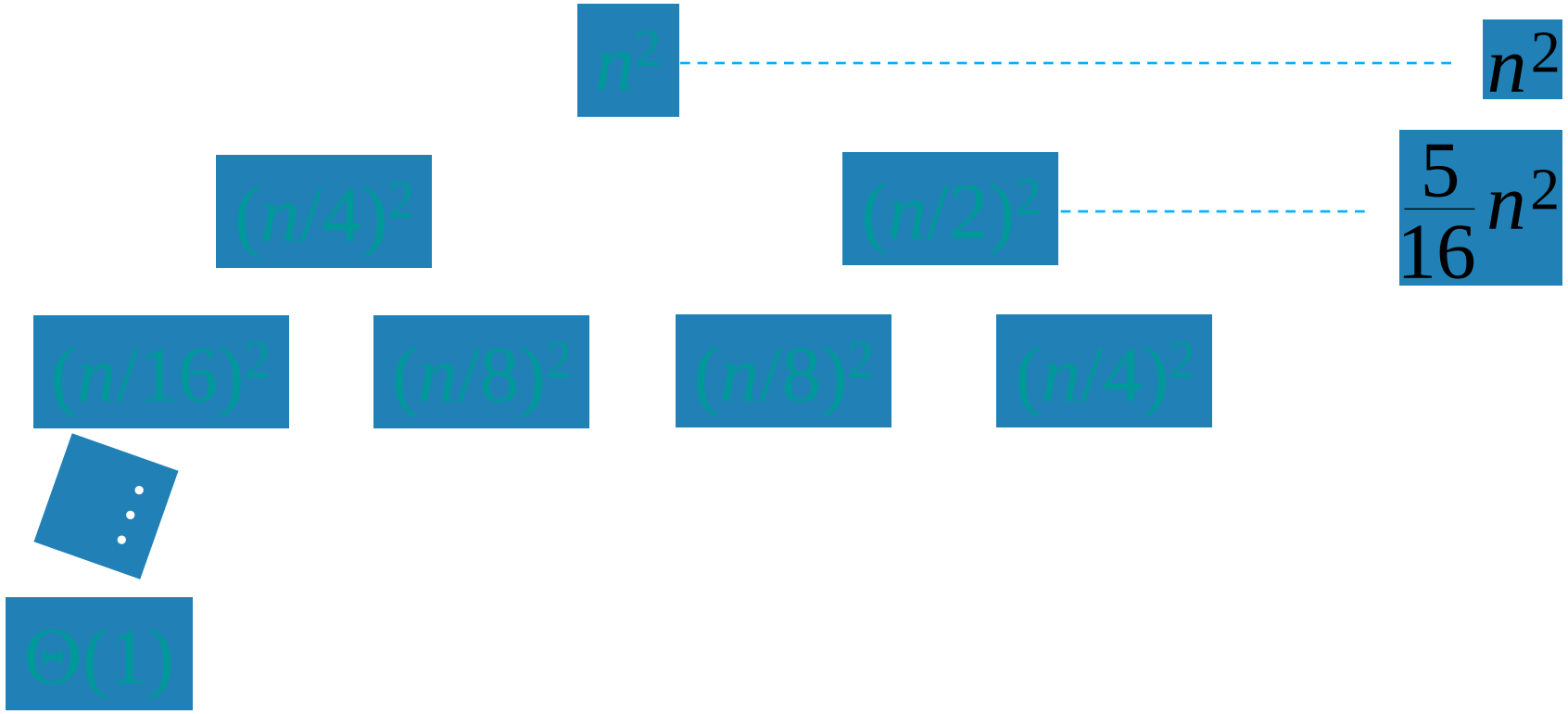


$$T(n) = T(n/4) + T(n/2) + n^2$$

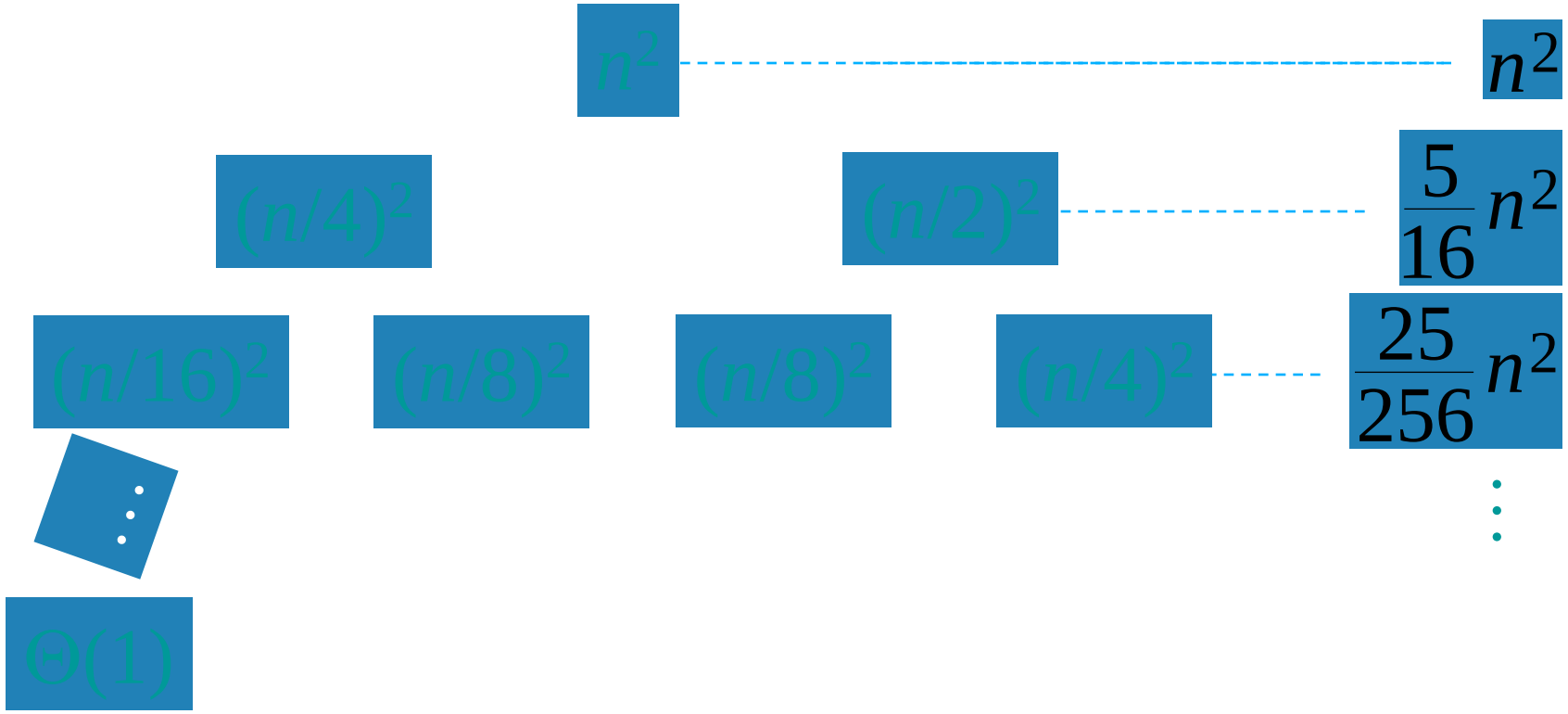
$$T(n) = T(n/4) + T(n/2) + n^2$$



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

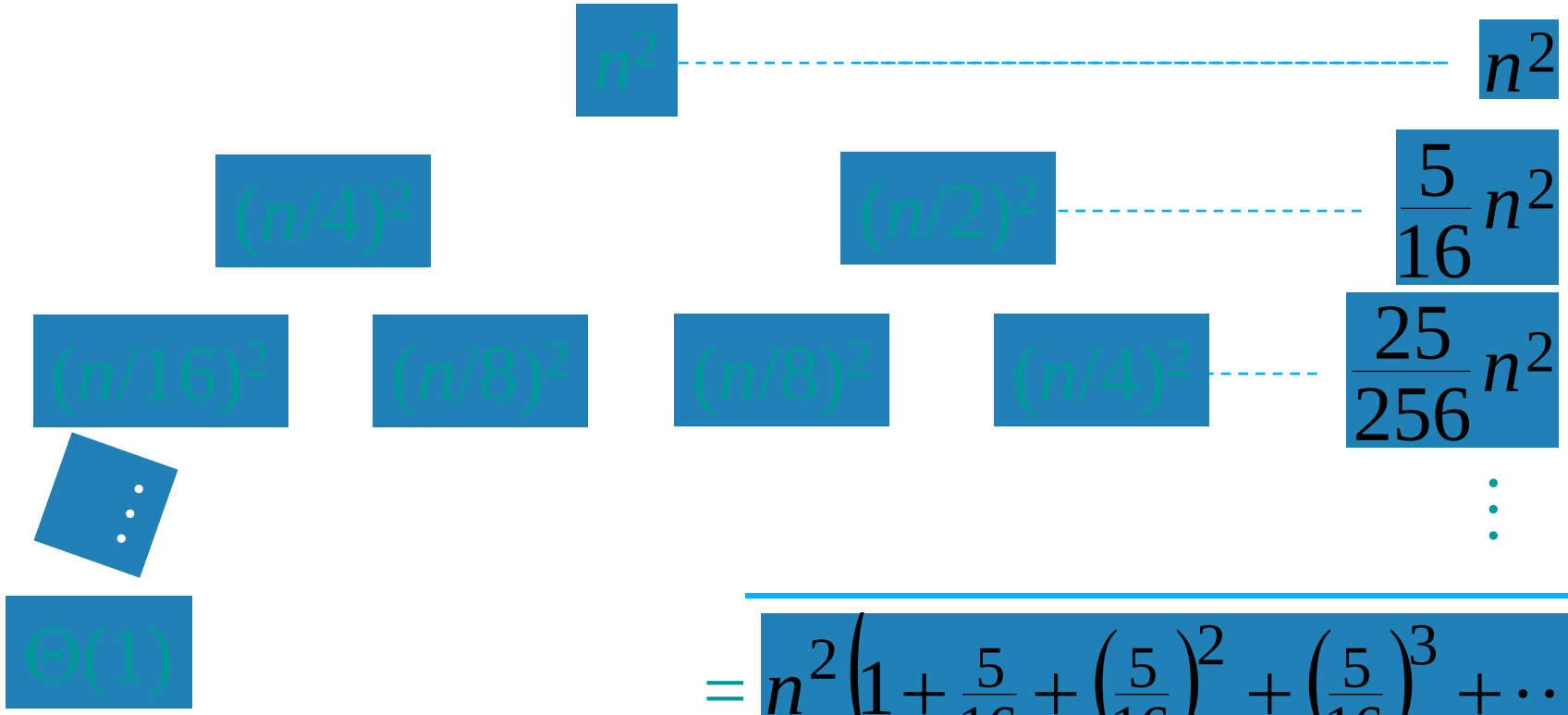


$$T(n) = T(n/4) + T(n/2) + n^2$$



Example of recursion tree

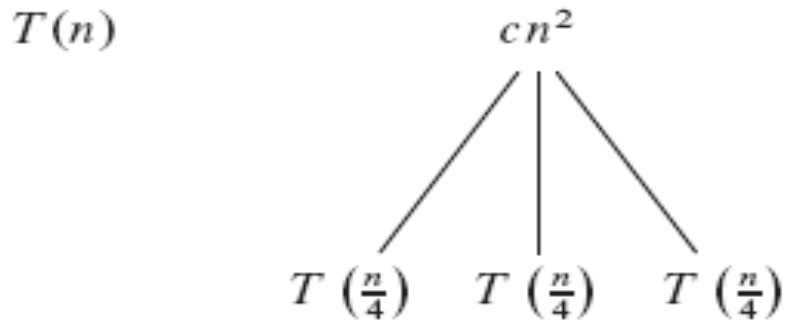
$$T(n) = T(n/4) + T(n/2) + n^2$$

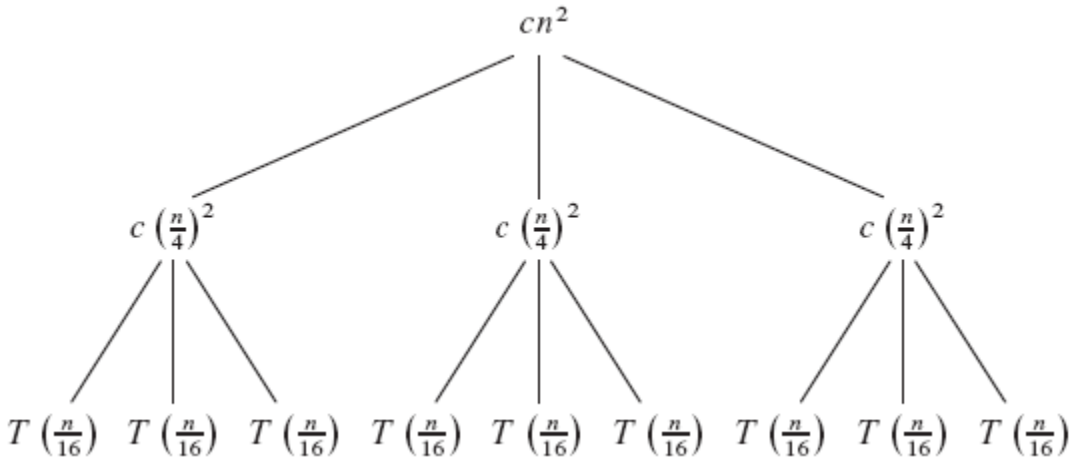


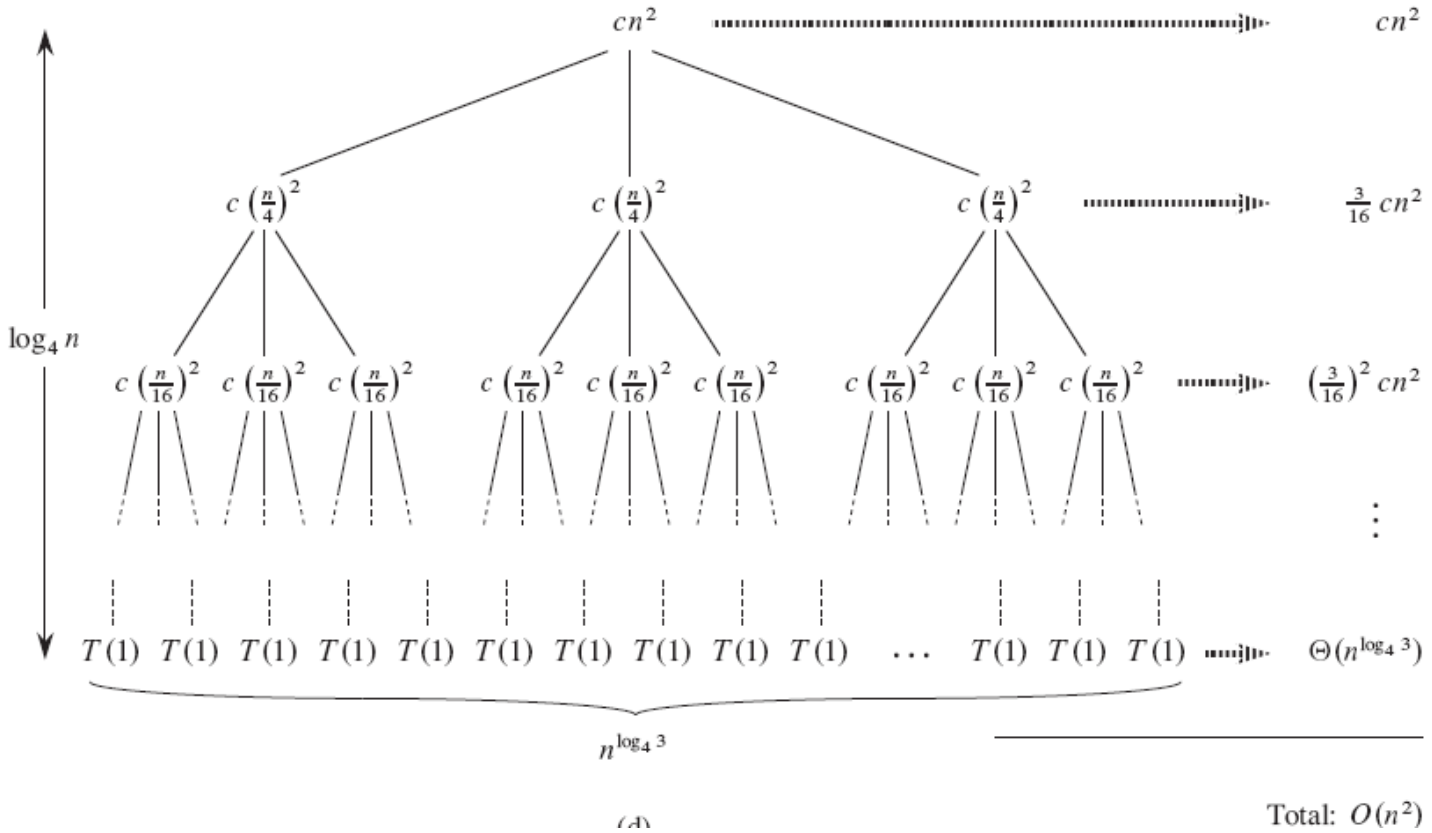
$$= n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$$= \Theta(n^2)$$ *geometric series*

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


T(n)
cn²
T (n/4)
T (n/4)
T (n/4)


cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$


cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
Total: $O(n^2)$

Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition. How far from the root do we reach one? The subproblem size for a node at depth i is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$).

Next we determine the cost at each level of the tree. Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i.

Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, which is $\Theta(n^{\log_4 3})$, since we assume that $T(1)$ is a constant.

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(by equation (A.5))} .
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) \, .
\end{aligned}
$$

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

$a \geq 1$ $b > 1$ f

Three common cases

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

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

n^{ε}

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

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

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

n^{ε}

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

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

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

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

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ $\varepsilon > 0$

- $f(n)$ $n^{\log_b a}$

n^{ε}

$f(n)$ ***regularity condition***

$a f(n/b) \leq c f(n)$ $c < 1$

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

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

$a = 4 \quad b = 2 \qquad n^{\log_b a} = n^2 \quad f(n) = n$

$f(n) = O(n^{2-\varepsilon}) \qquad = 1$

$T(n) = \Theta(n^2)$

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

$a = 4 \quad b = 2 \qquad n^{\log_b a} = n^2 \quad f(n) = n$

$f(n) = O(n^{2-\varepsilon}) \qquad = 1$

$T(n) = \Theta(n^2)$

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

$a = 4 \quad b = 2 \qquad n^{\log_b a} = n^2 \quad f(n) = n^2$

$f(n) = \Theta(n^2 \lg^0 n) \qquad k = 0$

$T(n) = \Theta(n^2 \lg n)$

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

$a = 4 \quad b = 2 \qquad n^{\log_b a} = n^2 \quad f(n) = n^3$

$f(n) = \Omega(n^{2+\varepsilon}) \qquad \varepsilon = 1$

$4(n/2)^3 \le cn^3 \qquad c = 1/2$

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

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

$a = 4 \quad b = 2 \qquad n^{\log_b a} = n^2 \quad f(n) = n^3$

$f(n) = \Omega(n^{2+\varepsilon}) \qquad \varepsilon = 1$

$4(n/2)^3 \le cn^3 \qquad c = 1/2$

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

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

$a = 4 \quad b = 2 \qquad n^{\log_b a} = n^2 \quad f(n) = n^2/\lg n$

$\varepsilon > 0 \qquad n^{\varepsilon} = \omega(\lg n)$

Lecture #13

Divide-and-Conquer

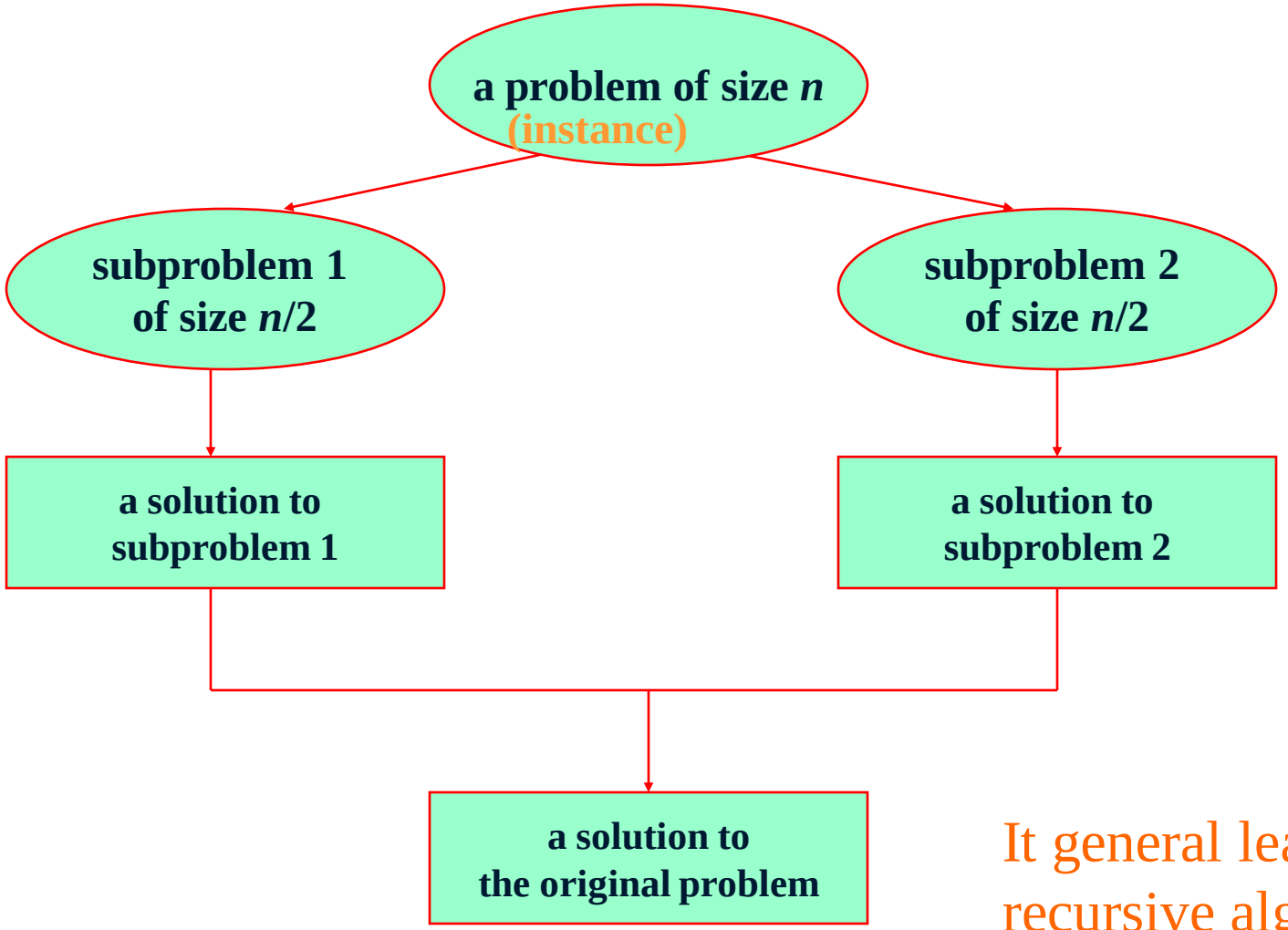
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)

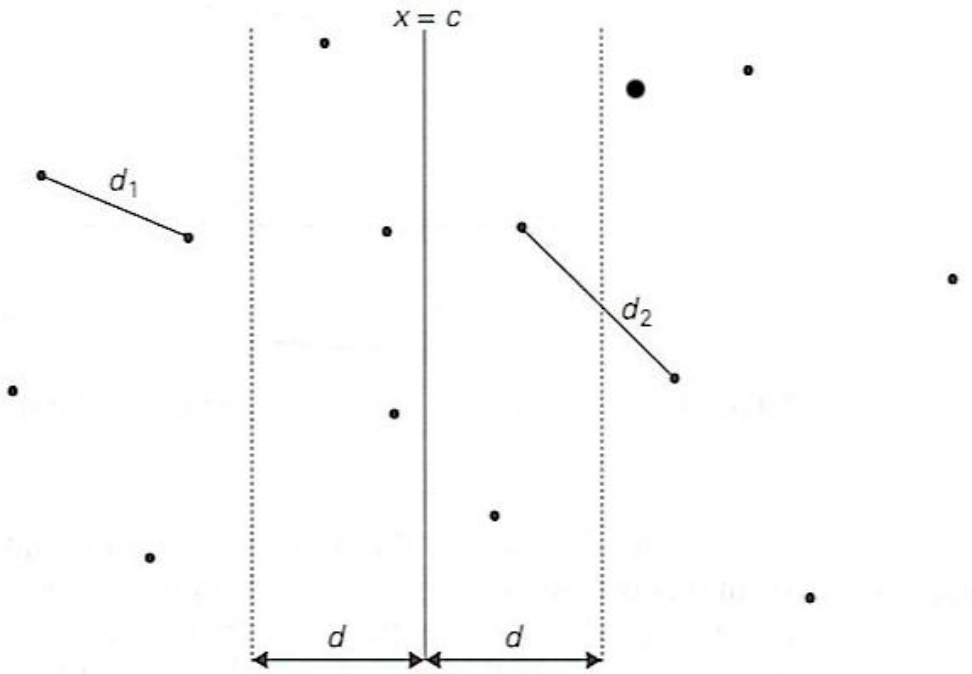


It general leads to a recursive algorithm!

Closest-Pair Problem by Divide-and-Conquer

Step 0 Sort the points by x (list one) and then by y (list two).

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Closest Pair by Divide-and-Conquer (cont.)

Step 2 Find recursively the closest pairs for the left and right subsets.

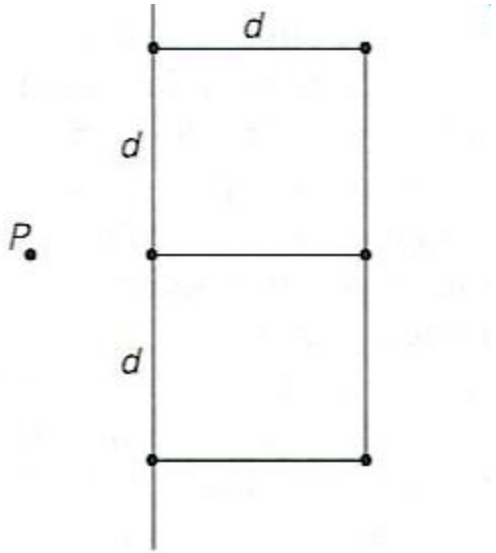
Step 3 Set $d = \min\{d_1, d_2\}$

We can limit our attention to the points in the symmetric vertical strip of width $2d$ as possible closest pair. Let C_1 and C_2 be the subsets of points in the left subset S_1 and of the right subset S_2, respectively, that lie in this vertical strip. The points in C_1 and C_2 are stored in increasing order of their y coordinates, taken from the second list.

Step 4 For every point $P(x,y)$ in C_1, we inspect points in C_2 that may be closer to P than d. There can be no more than 6 such points (because $d \leq d_2$)!

Closest Pair by Divide-and-Conquer: Worst Case

The worst case scenario is depicted below:



Efficiency of the Closest-Pair Algorithm

Running time of the algorithm (without sorting) is:

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in \Theta(n)$$

By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

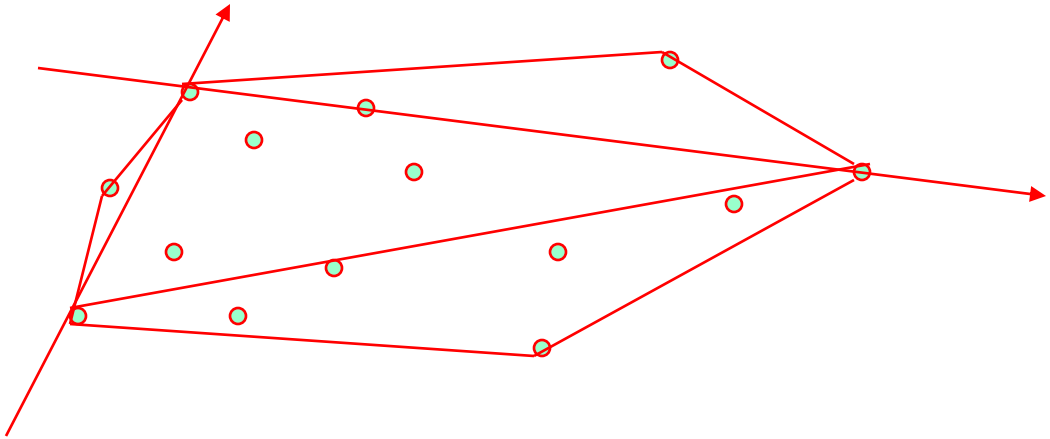
$$T(n) \in \Theta(n \log n)$$

So the total time is $\Theta(n \log n)$.

Convex hull Algorithm

Convex hull: smallest convex set that includes given points. An O(n^3) bruteforce time is given in Levitin, Ch 3.

- Assume points are sorted by *x*-coordinate values
- Identify *extreme points* P_1 and P_2 (leftmost and rightmost)
- Compute *upper hull* recursively:
  - find point P_{max} that is farthest away from line P_1P_2
  - compute the upper hull of the points to the left of line P_1P_{max}
  - compute the upper hull of the points to the left of line $P_{max}P_2$
- Compute *lower hull* in a similar manner



Efficiency of Convex hull Algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency: T(n) = T(x) + T(y) + T(z) + T(v) + O(n), where x + y + z +v <= n.
  - worst case: $\Theta(n^2)$
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- If points are not initially sorted by *x*-coordinate value, this can be accomplished in O(*n* log *n*) time
- Several O(*n* log *n*) algorithms for convex hull are known

Insertion Sort

- Iteration i.

- Property.

Array index	0	1	2	3	4	5	6	7	8	9
	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 0: step 0.

Insertion Sort

- Iteration i.

- Property.

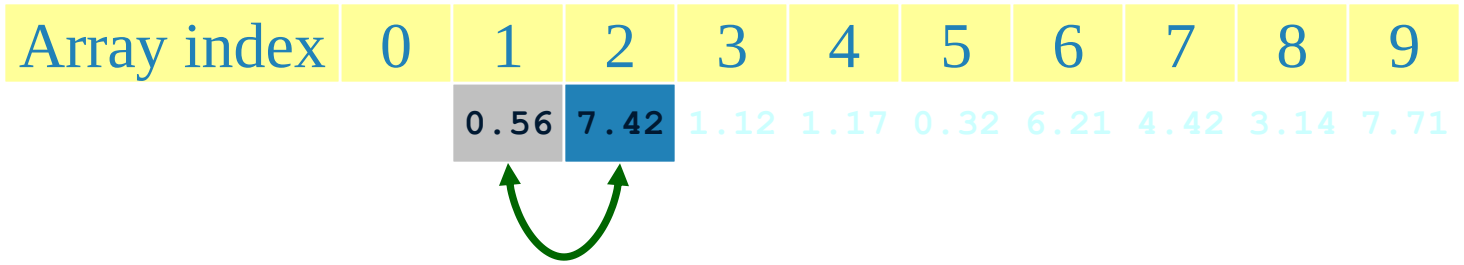
Array index	0	1	2	3	4	5	6	7	8	9
		7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 1: step 0.

Insertion Sort

- Iteration i.

- Property.

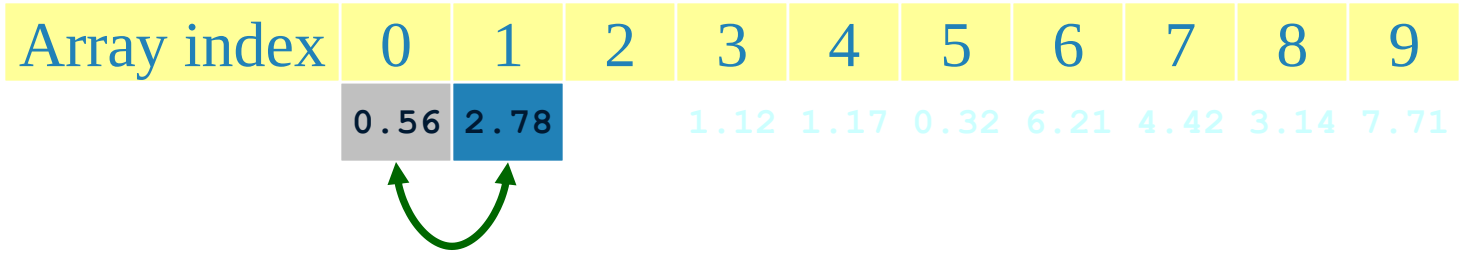


Iteration 2: step 0.

Insertion Sort

- Iteration i.

- Property.



Iteration 2: step 1.

Insertion Sort

- Iteration i.

- Property.

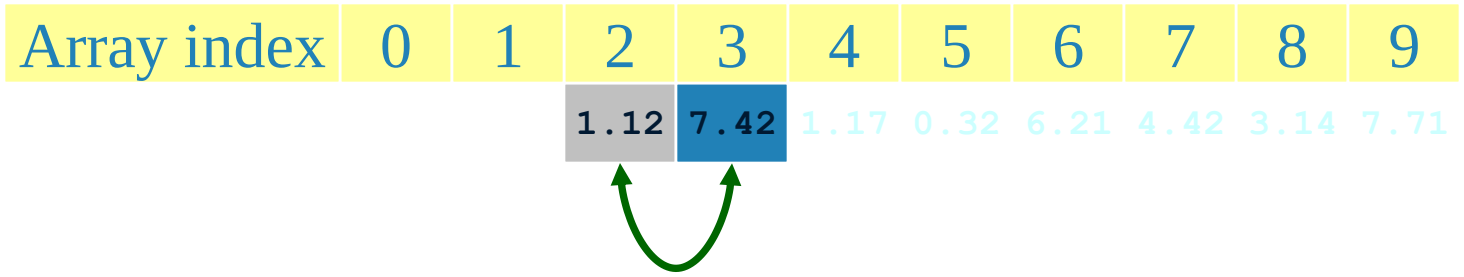
Array index	0	1	2	3	4	5	6	7	8	9
	0.56			1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 2.

Insertion Sort

- Iteration i.

- Property.

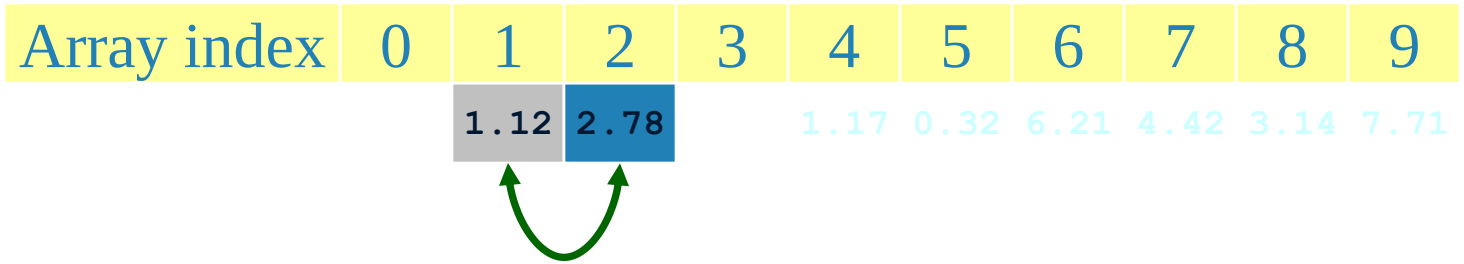


Iteration 3: step 0.

Insertion Sort

- Iteration i.

- Property.



Iteration 3: step 1.

Insertion Sort

- Iteration i.

- Property.

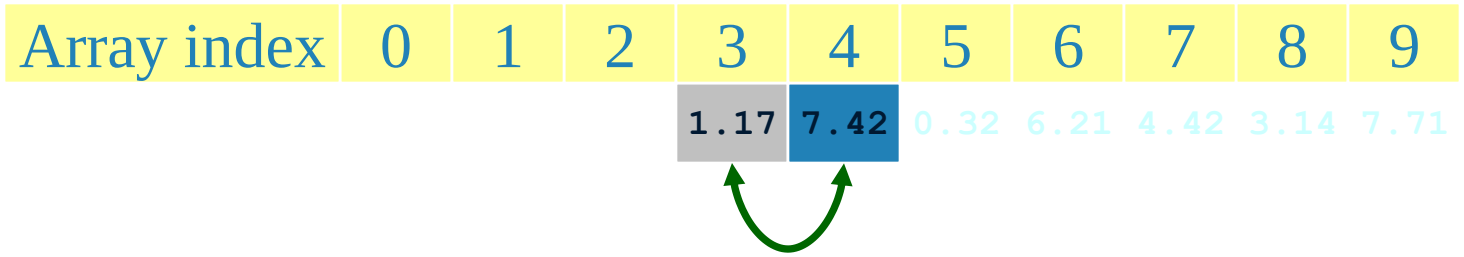
Array index	0	1	2	3	4	5	6	7	8	9
		1.12			1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 2.

Insertion Sort

- Iteration i.

- Property.

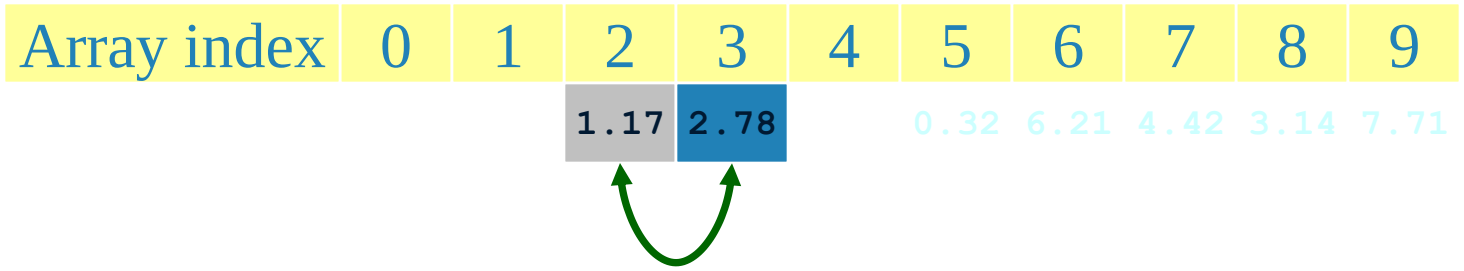


Iteration 4: step 0.

Insertion Sort

- Iteration i.

- Property.



Iteration 4: step 1.

Insertion Sort

- Iteration i.

- Property.

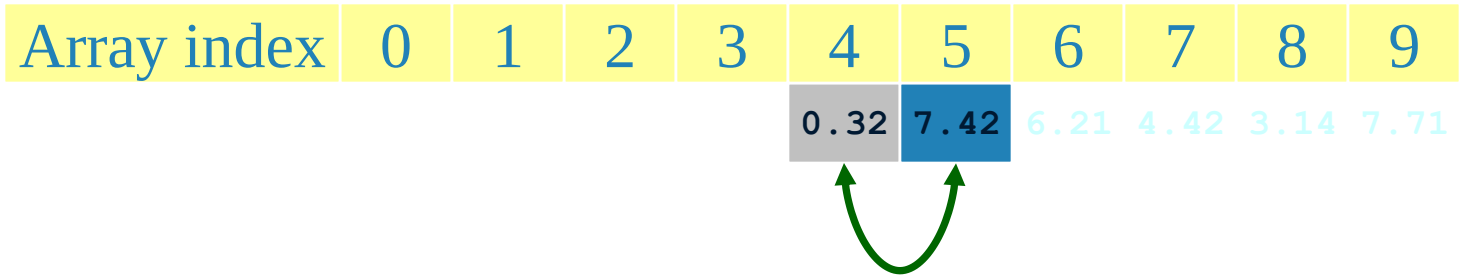
Array index	0	1	2	3	4	5	6	7	8	9
			1.17			0.32	6.21	4.42	3.14	7.71

Iteration 4: step 2.

Insertion Sort

- Iteration i.

- Property.

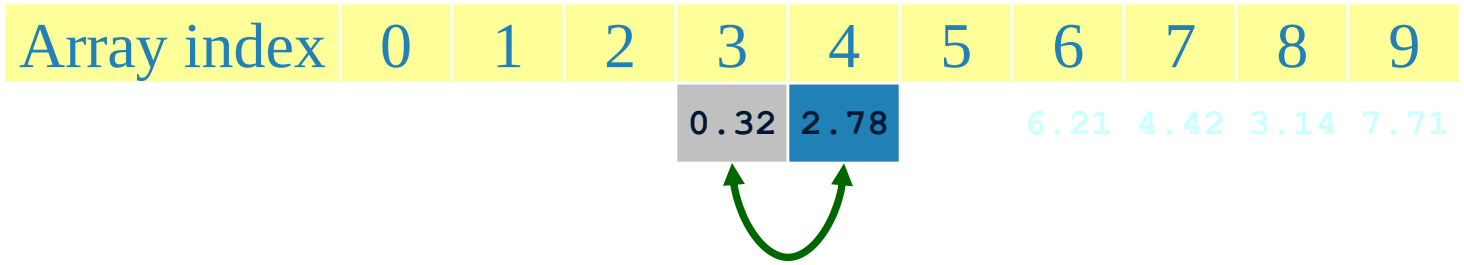


Iteration 5: step 0.

Insertion Sort

- Iteration i.

- Property.

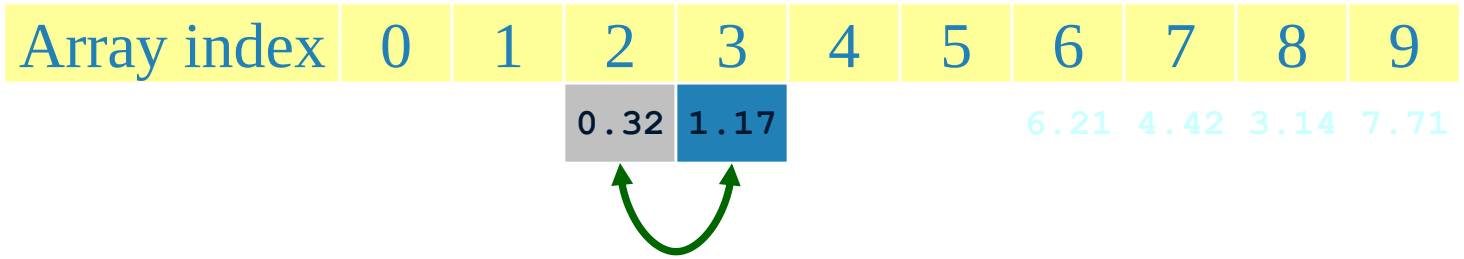


Iteration 5: step 1.

Insertion Sort

- Iteration i.

- Property.

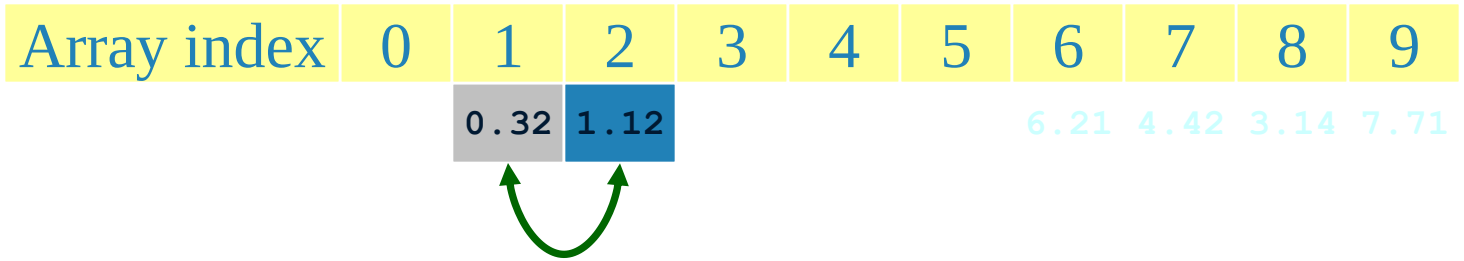


Iteration 5: step 2.

Insertion Sort

- Iteration i.

- Property.

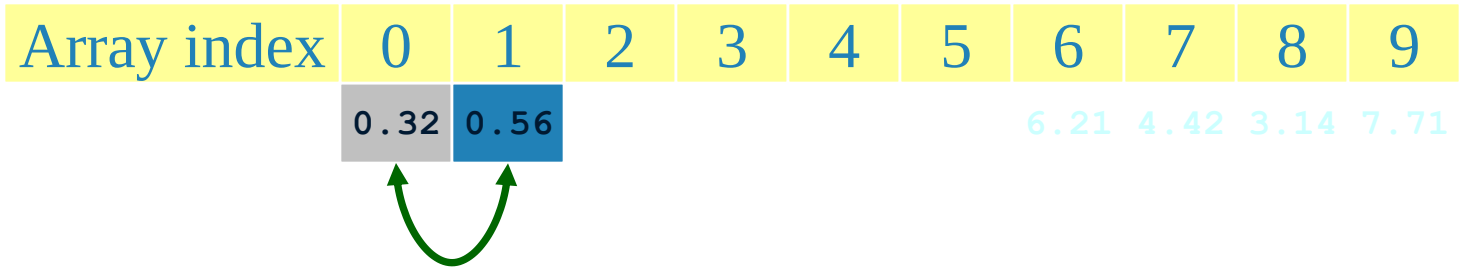


Iteration 5: step 3.

Insertion Sort

- Iteration i.

- Property.



Iteration 5: step 4.

Insertion Sort

- Iteration i.

- Property.

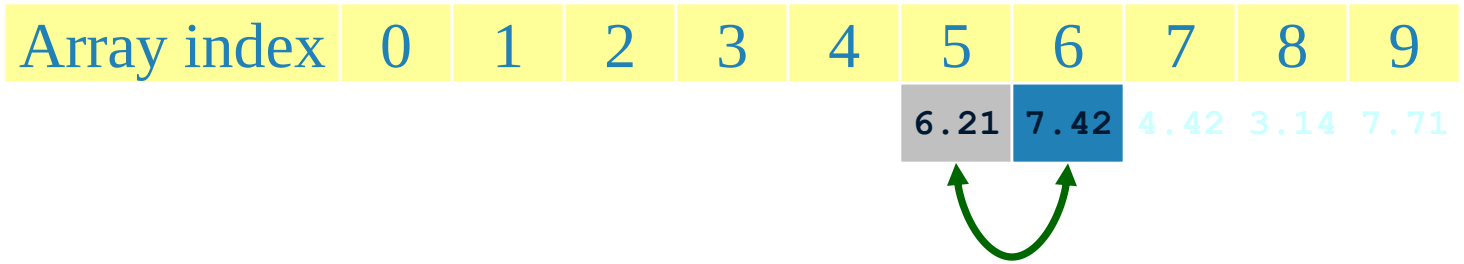
Array index	0	1	2	3	4	5	6	7	8	9
	0.32						6.21	4.42	3.14	7.71

Iteration 5: step 5.

Insertion Sort

- Iteration i.

- Property.



Iteration 6: step 0.

Insertion Sort

- Iteration i.

- Property.

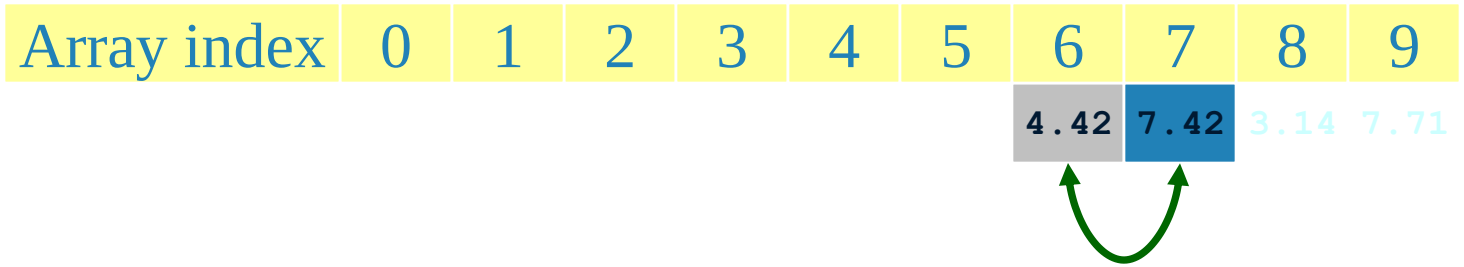
Array index	0	1	2	3	4	5	6	7	8	9
						6.21		4.42	3.14	7.71

Iteration 6: step 1.

Insertion Sort

- Iteration i.

- Property.

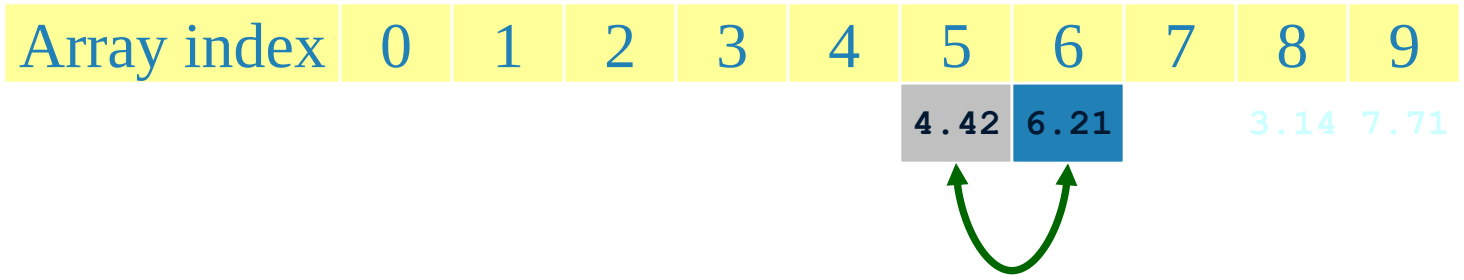


Iteration 7: step 0.

Insertion Sort

- Iteration i.

- Property.



Iteration 7: step 1.

Insertion Sort

- Iteration i.

- Property.

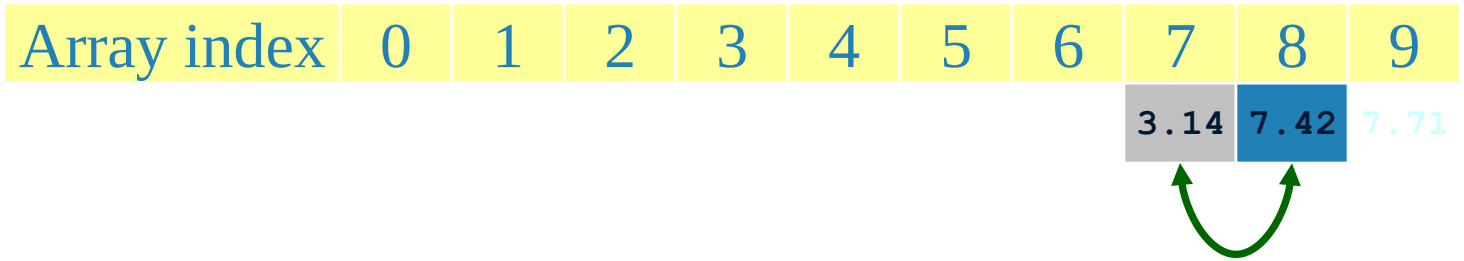
Array index	0	1	2	3	4	5	6	7	8	9
						4.42			3.14	7.71

Iteration 7: step 2.

Insertion Sort

- Iteration i.

- Property.

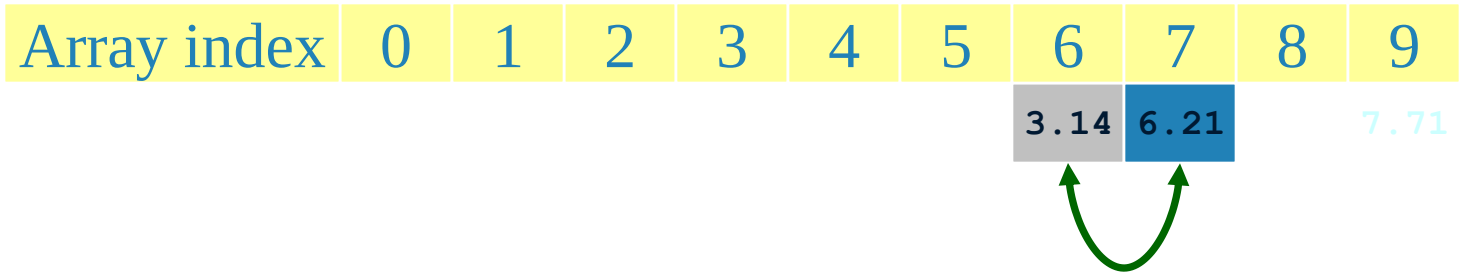


Iteration 8: step 0.

Insertion Sort

- Iteration i.

- Property.

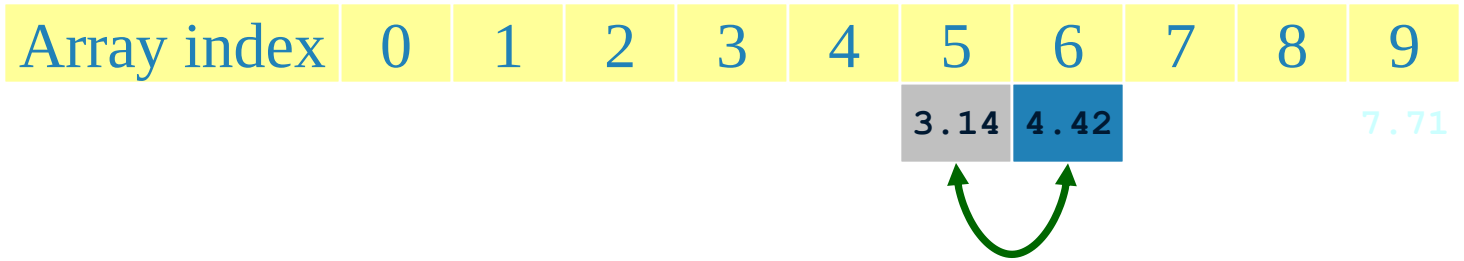


Iteration 8: step 1.

Insertion Sort

- Iteration i.

- Property.



Iteration 8: step 2.

Insertion Sort

- Iteration i.

- Property.

Array index	0	1	2	3	4	5	6	7	8	9
						3.14				7.71

Iteration 8: step 3.

Insertion Sort

- Iteration i.

- Property.

Array index	0	1	2	3	4	5	6	7	8	9
										7.71

Iteration 9: step 0.

Insertion Sort

- Iteration i.
- Property.
-
-

Array index	0	1	2	3	4	5	6	7	8	9

Iteration 10: DONE.

- Insertion Sort(n)
- {
- For j=2 to A.length
- Key=A[j]
- i=j-1
- While(i>0 and A[i]>key)
- A[i+1]=A[i]
- i=i-1
- A[i+1]=key
- }

CSE408
BFS, DFS, Connected components

Lecture #14

Graph Traversal

Topics

- Depth First Search (DFS)
- Breadth First Search (BFS)

Graph Search (traversal)

- How do we search a graph?
  - At a particular vertices, where shall we go next?
- Two common framework:
    - the depth-first search (DFS)
    - the breadth-first search (BFS) and
  - In DFS, go as far as possible along a single path until reach a dead end (a vertex with no edge out or no neighbor unexplored) then backtrack
  - In BFS, one explore a graph level by level away (explore all neighbors first and then move on)

Depth-First Search (DFS)

- The basic idea behind this algorithm is that it traverses the graph using recursion
  - Go as far as possible until you reach a deadend
  - Backtrack to the previous path and try the next branch
  - The graph below, started at node a, would be visited in the following order: a, b, c, g, h, i, e, d, f, j

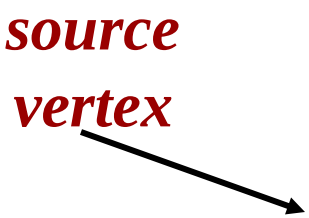
DFS: Color Scheme

- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

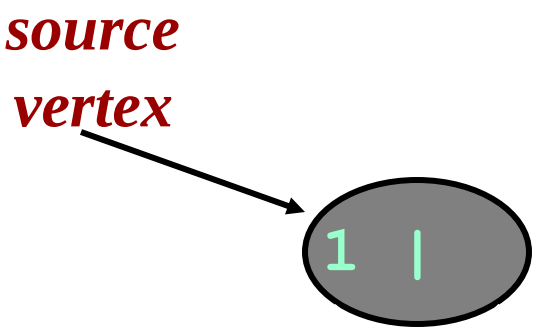
DFS: Time Stamps

- Discover time d[u]: when u is first discovered
- Finish time f[u]: when backtrack from u
- d[u] < f[u]

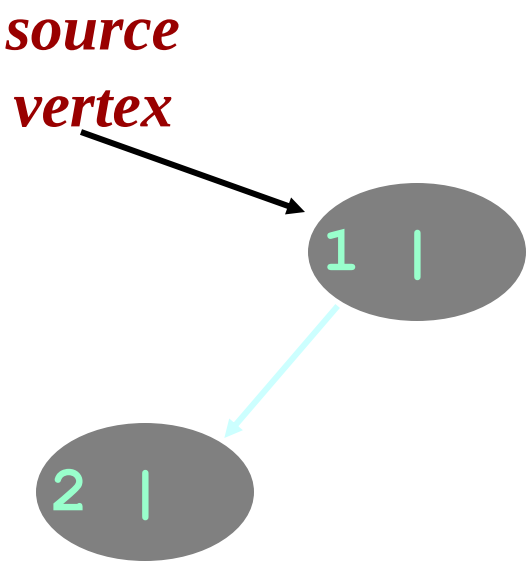
DFS Example



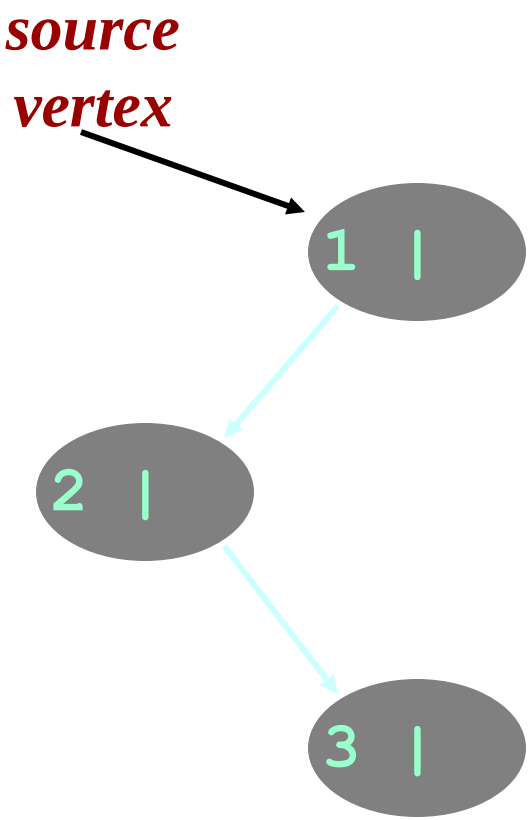
DFS Example



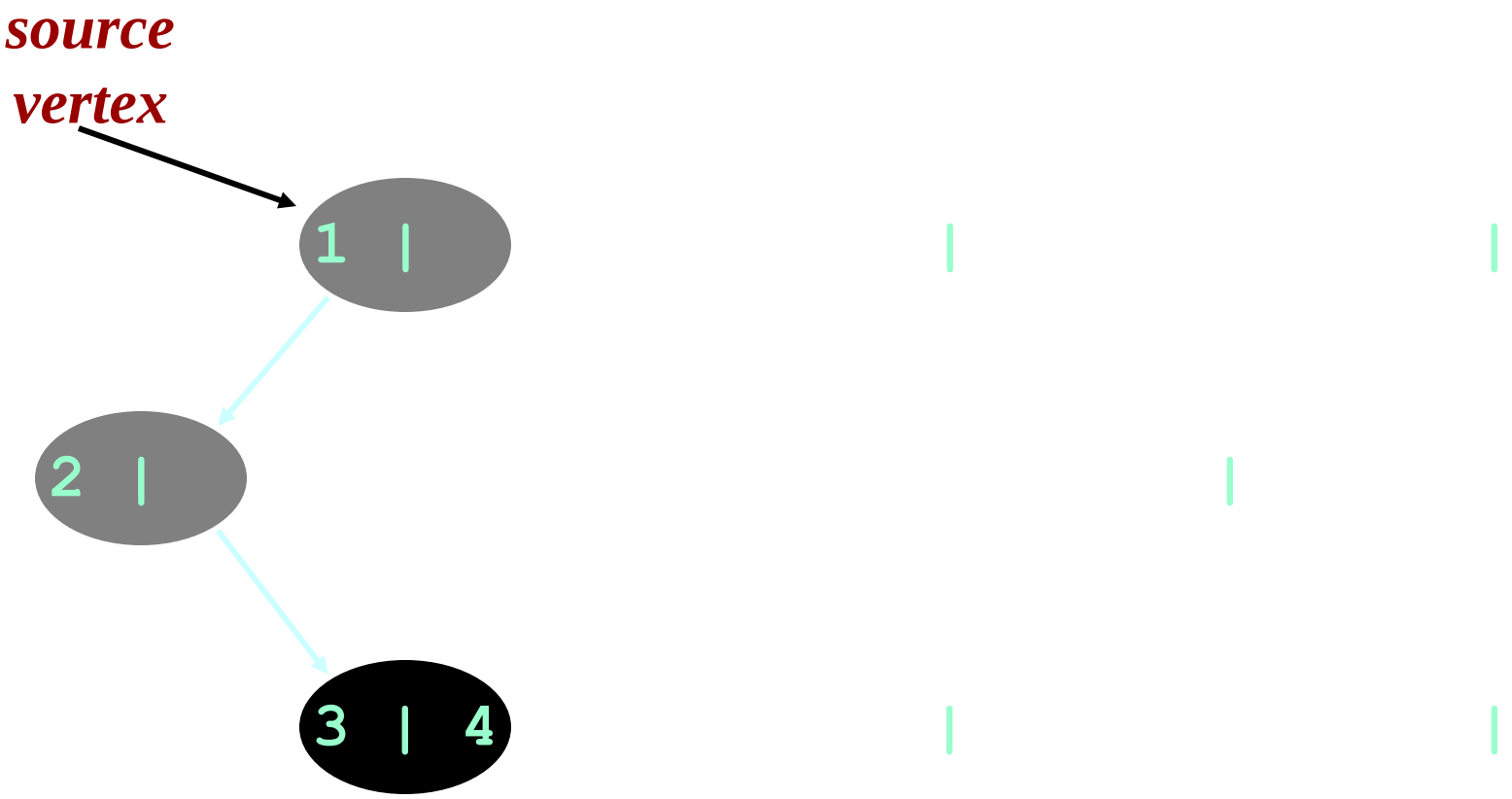
DFS Example



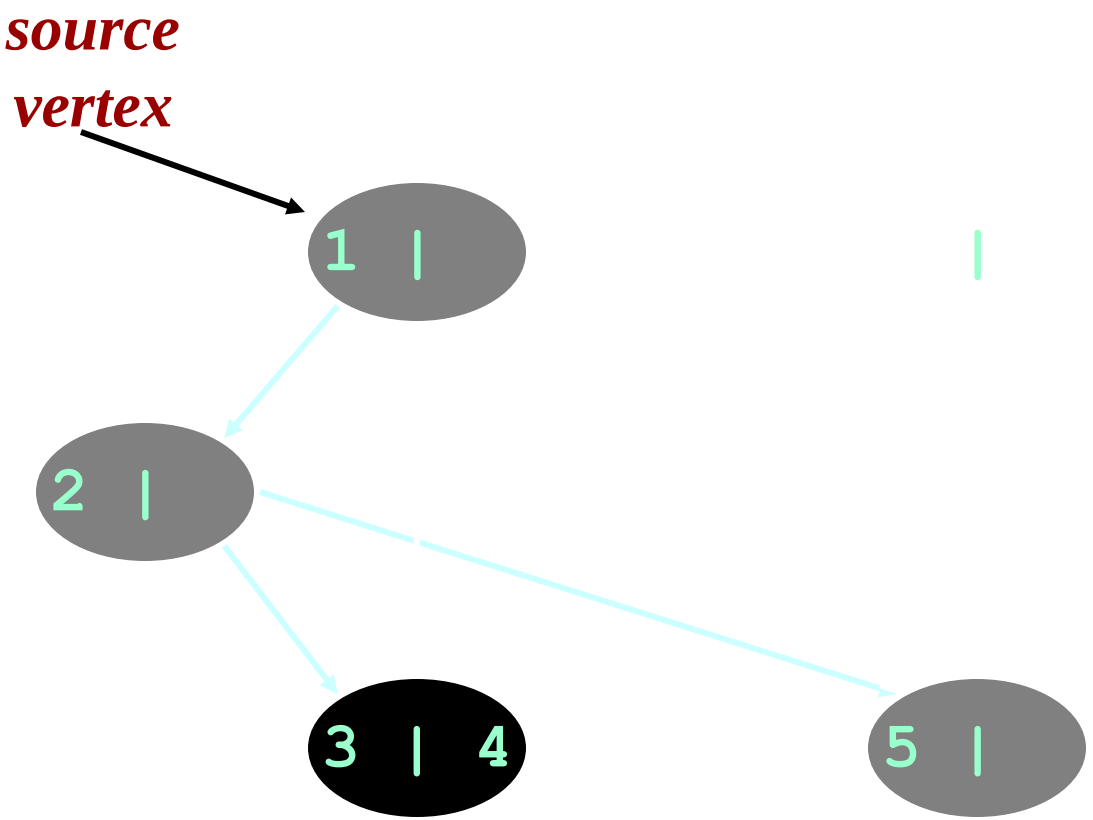
DFS Example



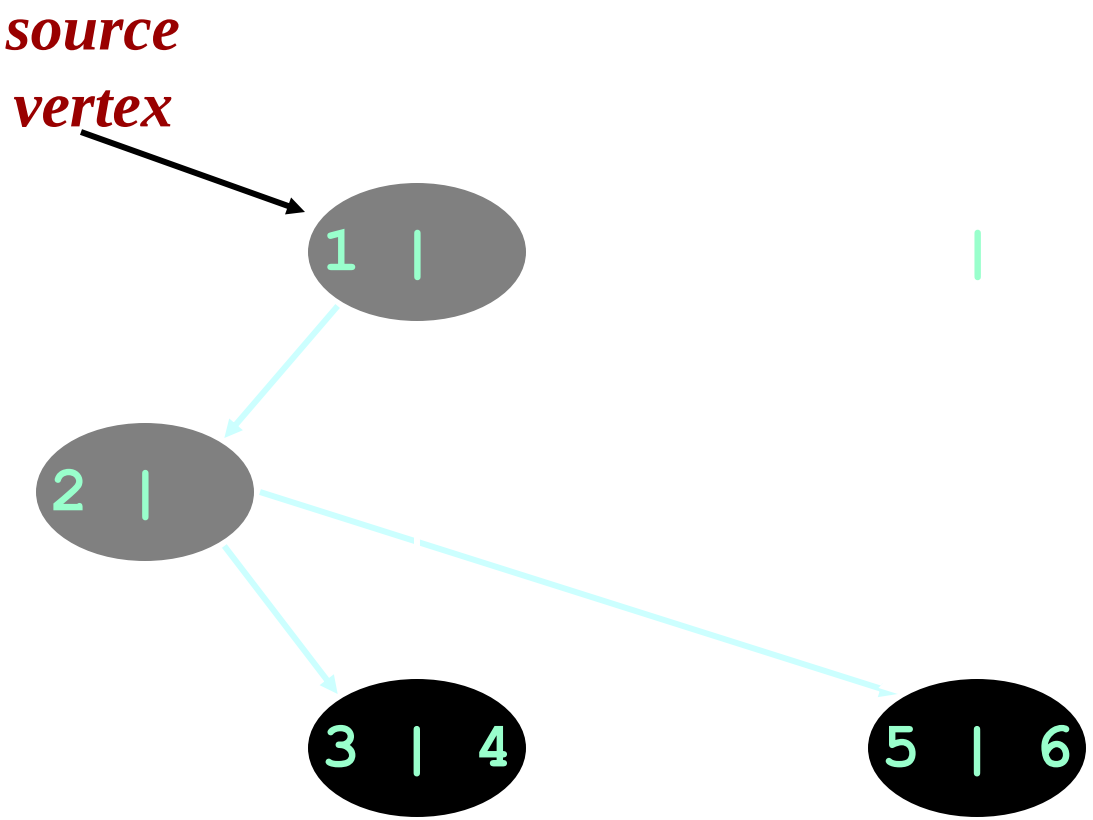
DFS Example



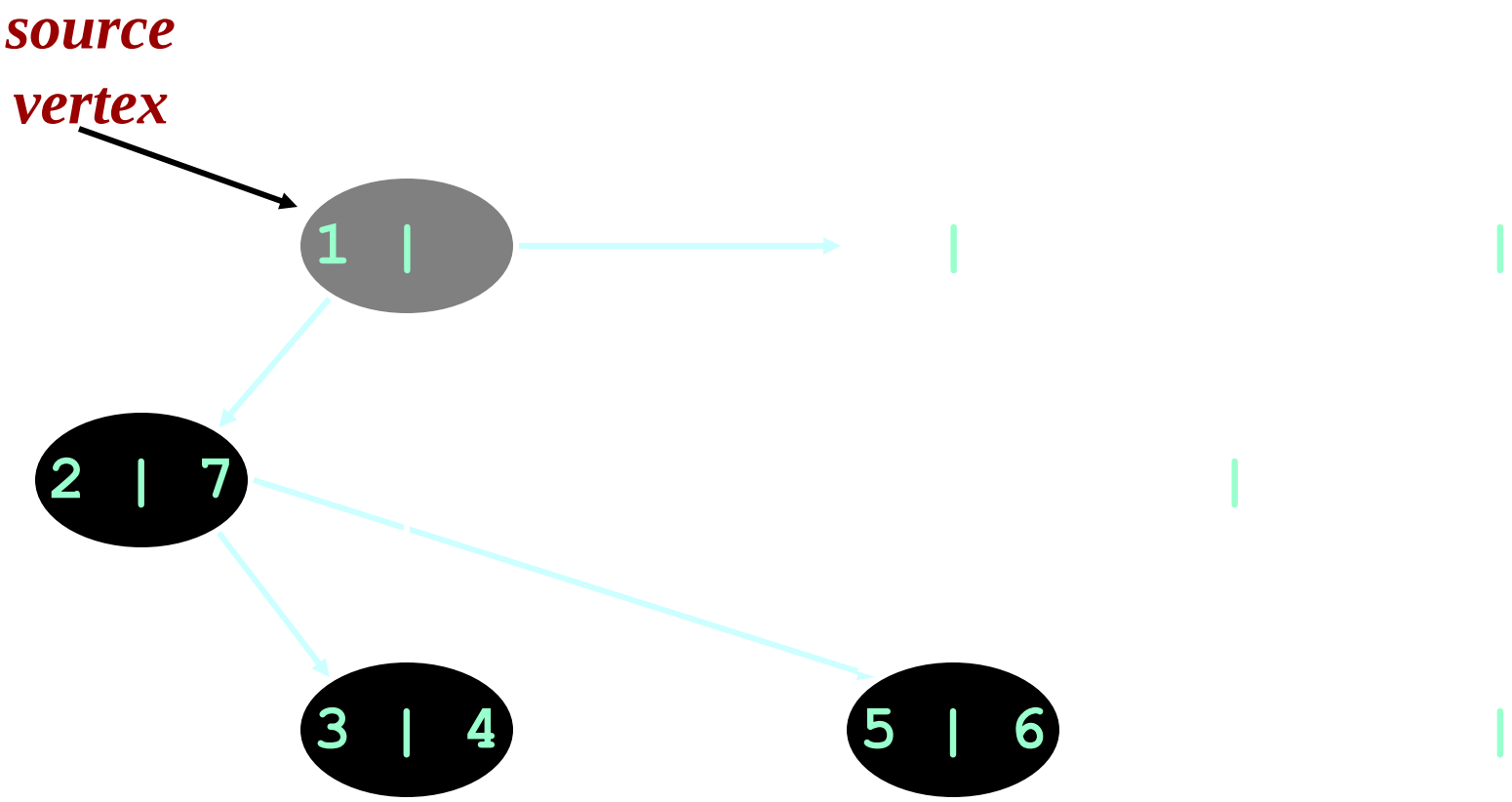
DFS Example



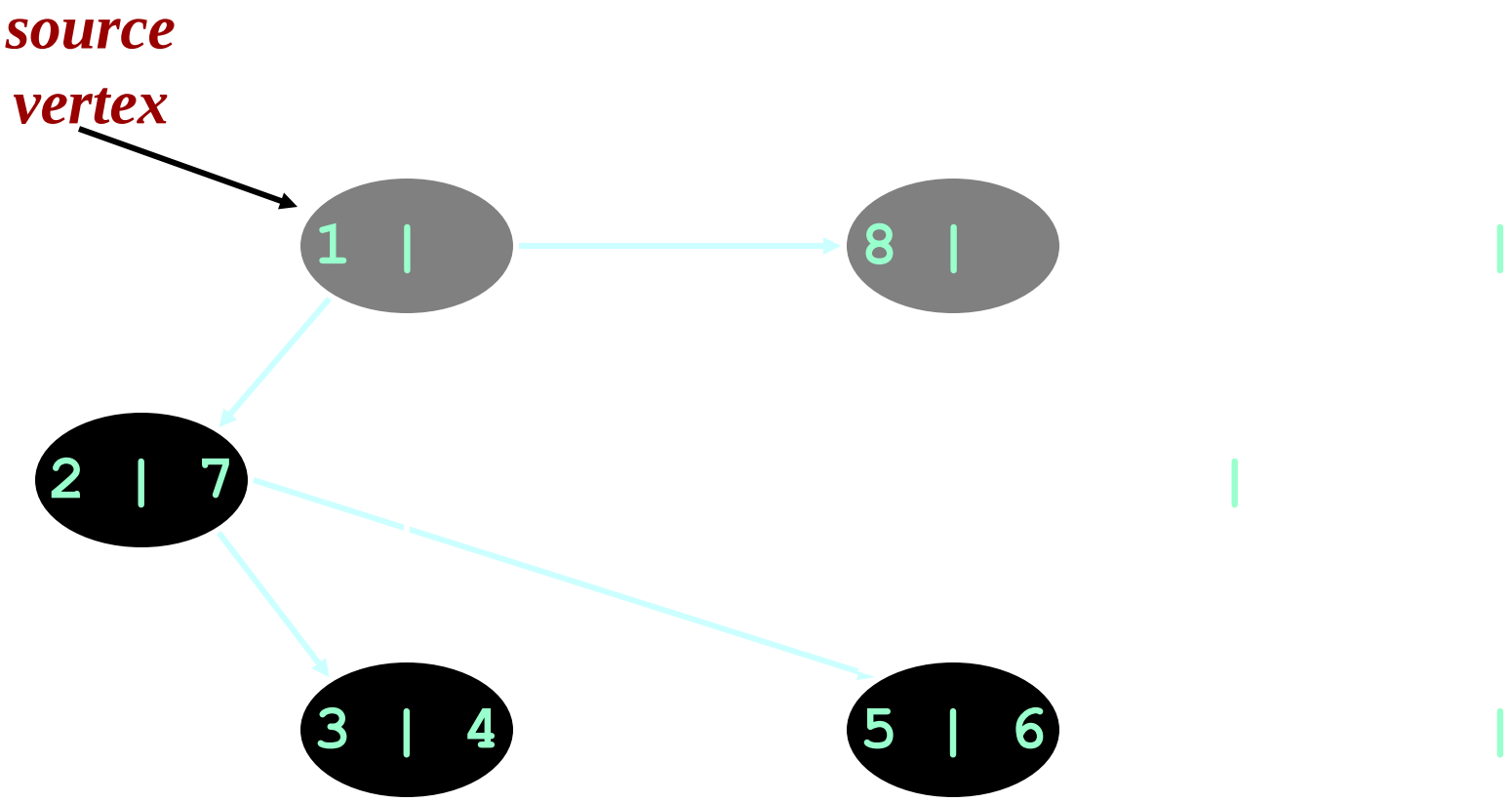
DFS Example



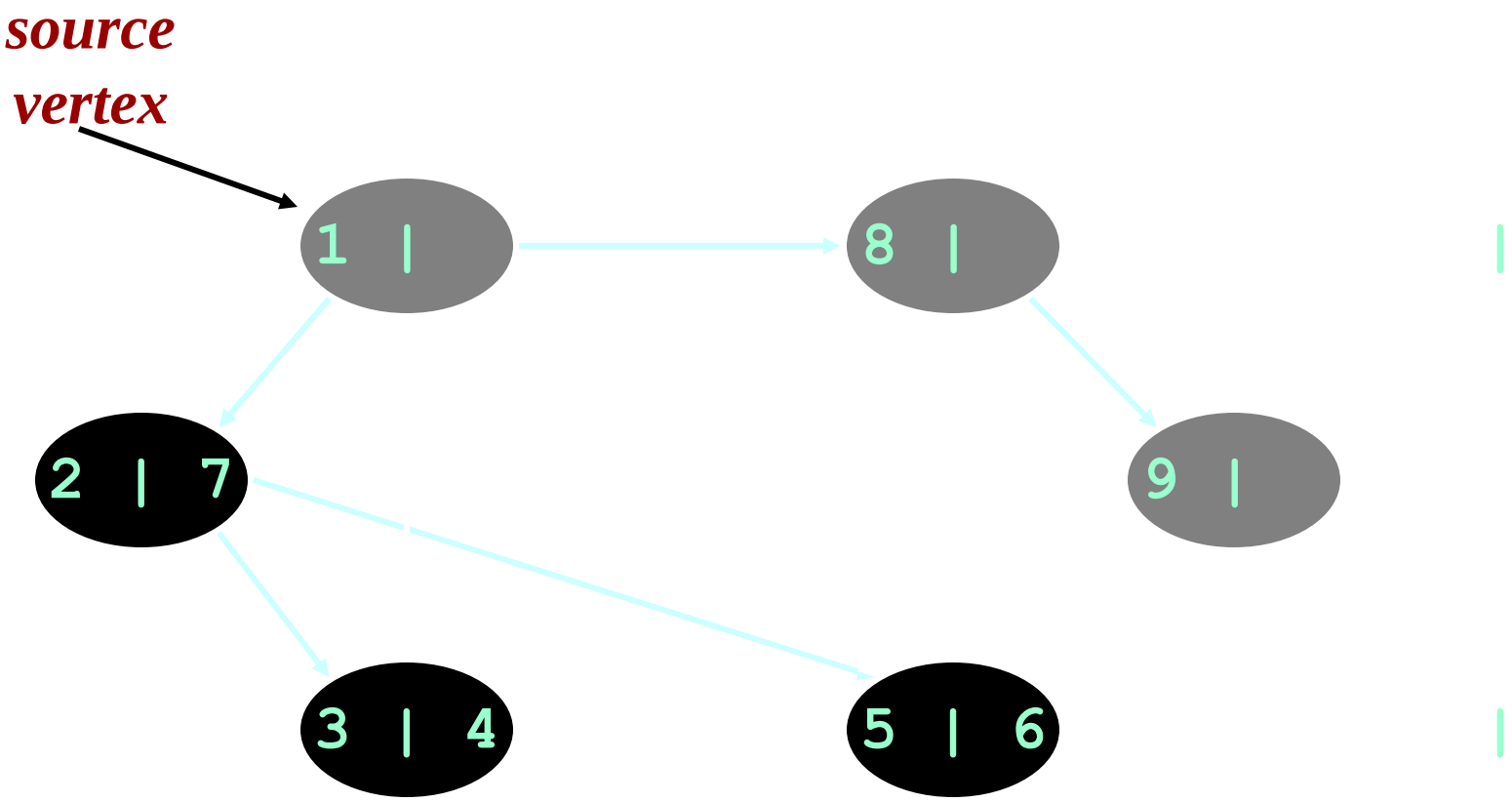
DFS Example



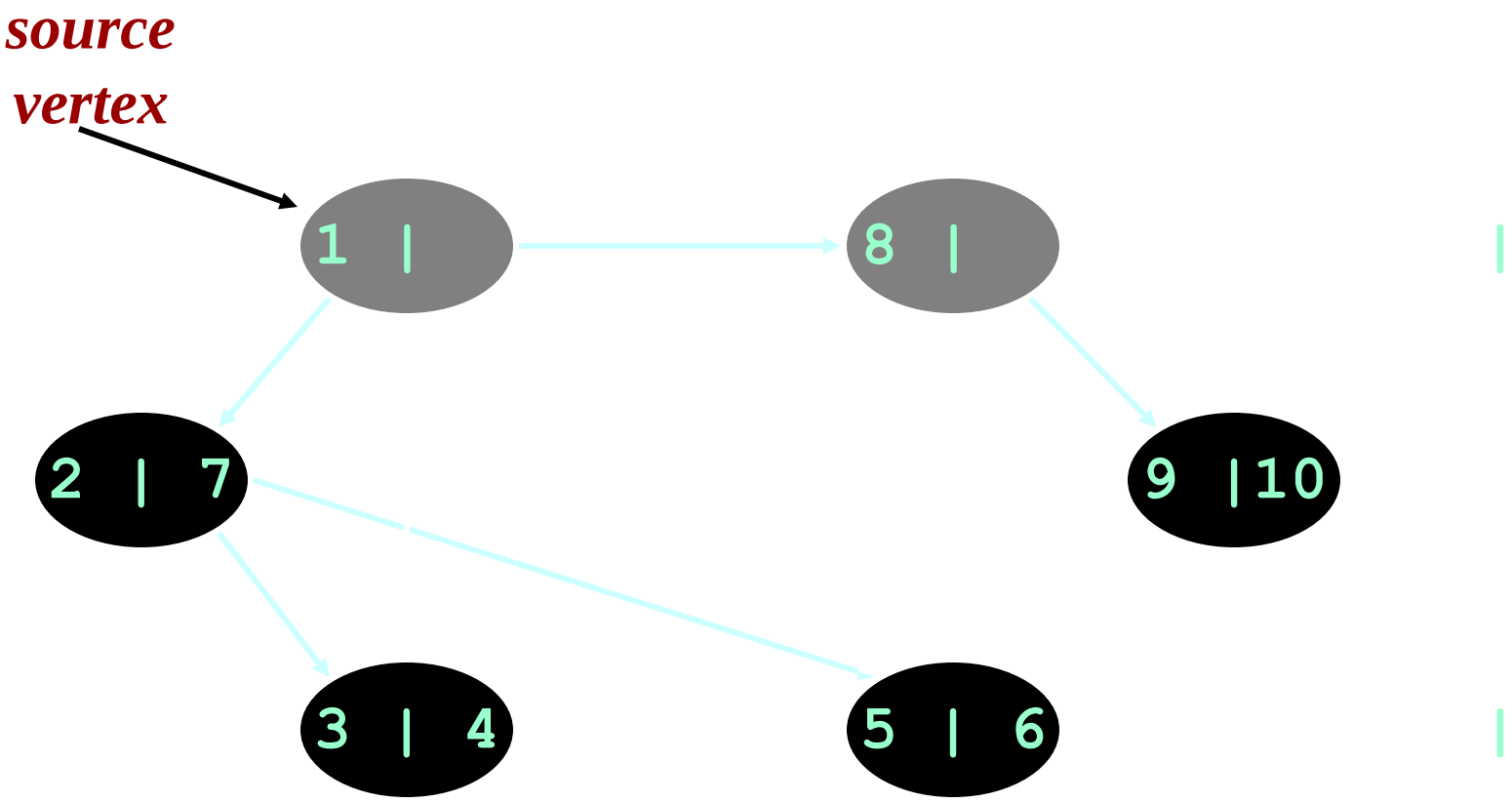
DFS Example



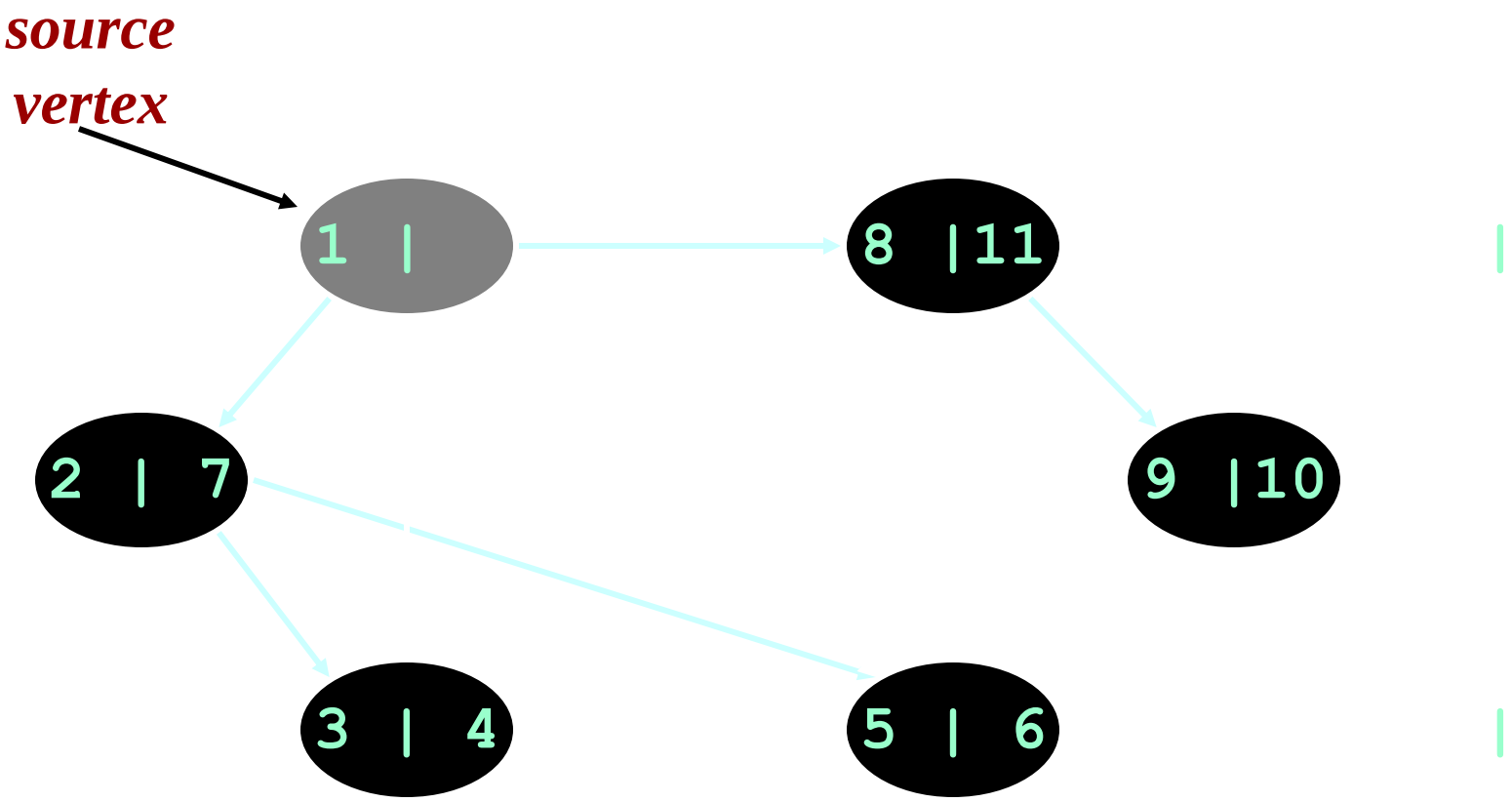
DFS Example



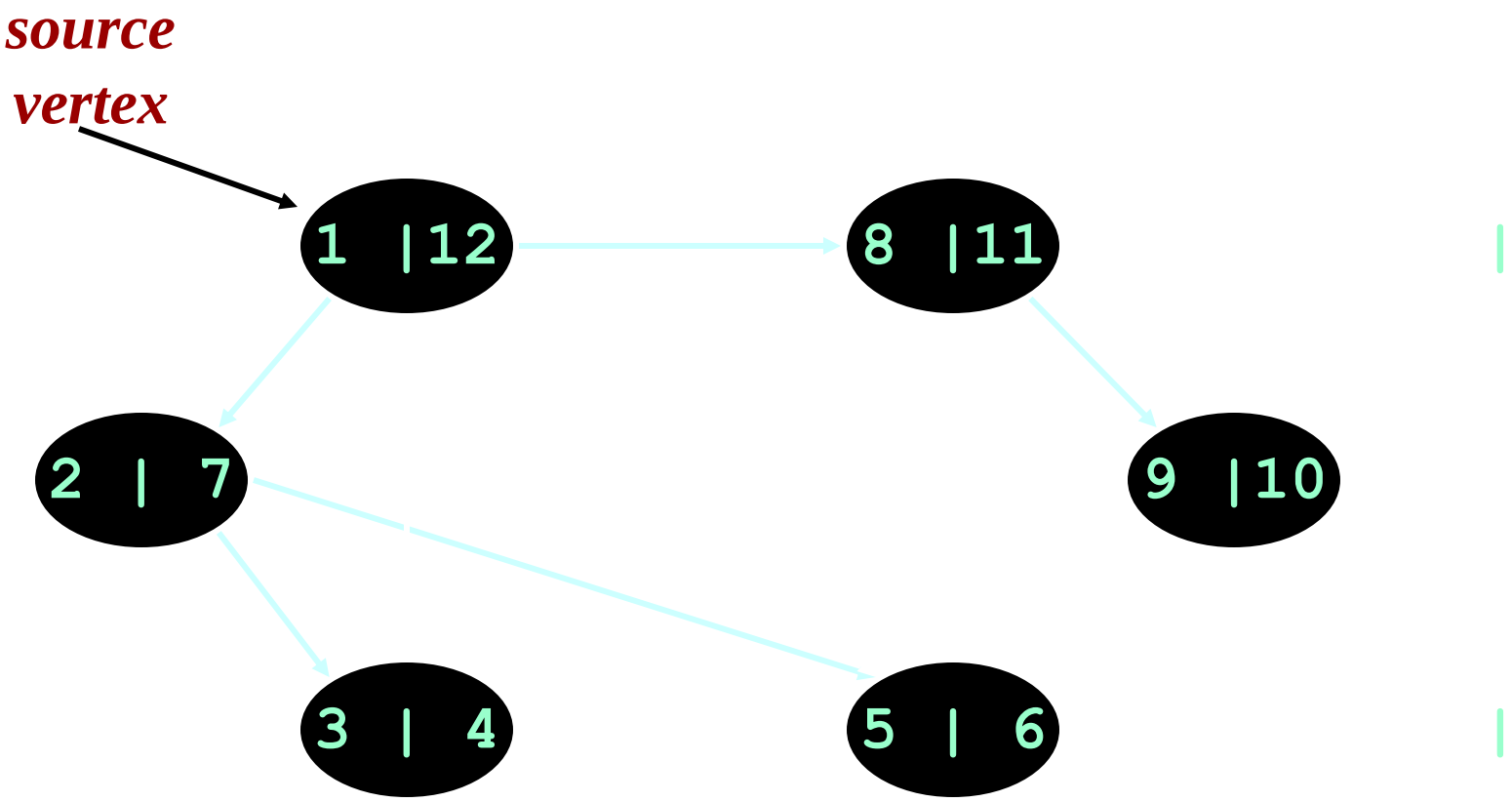
DFS Example



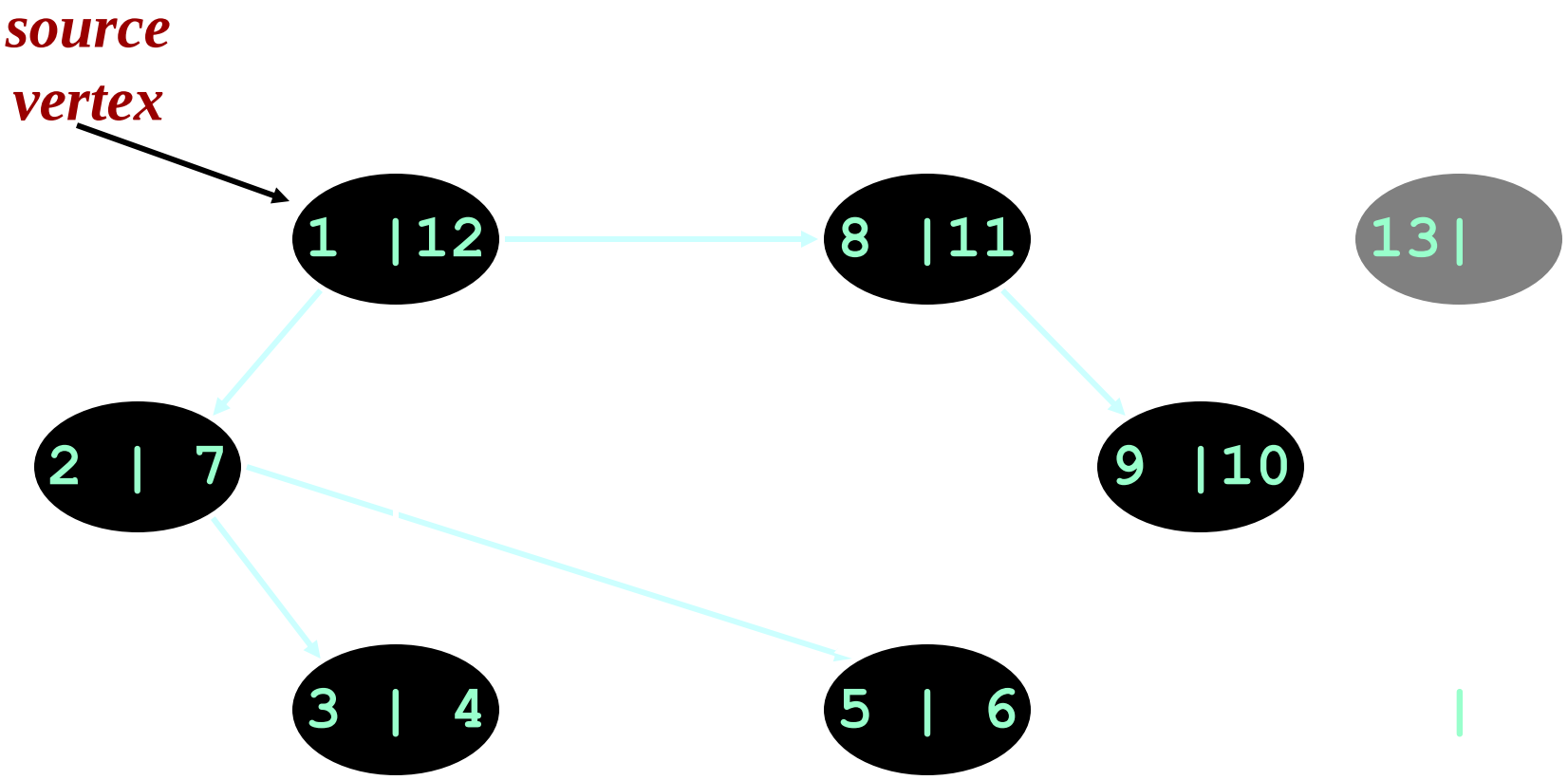
DFS Example



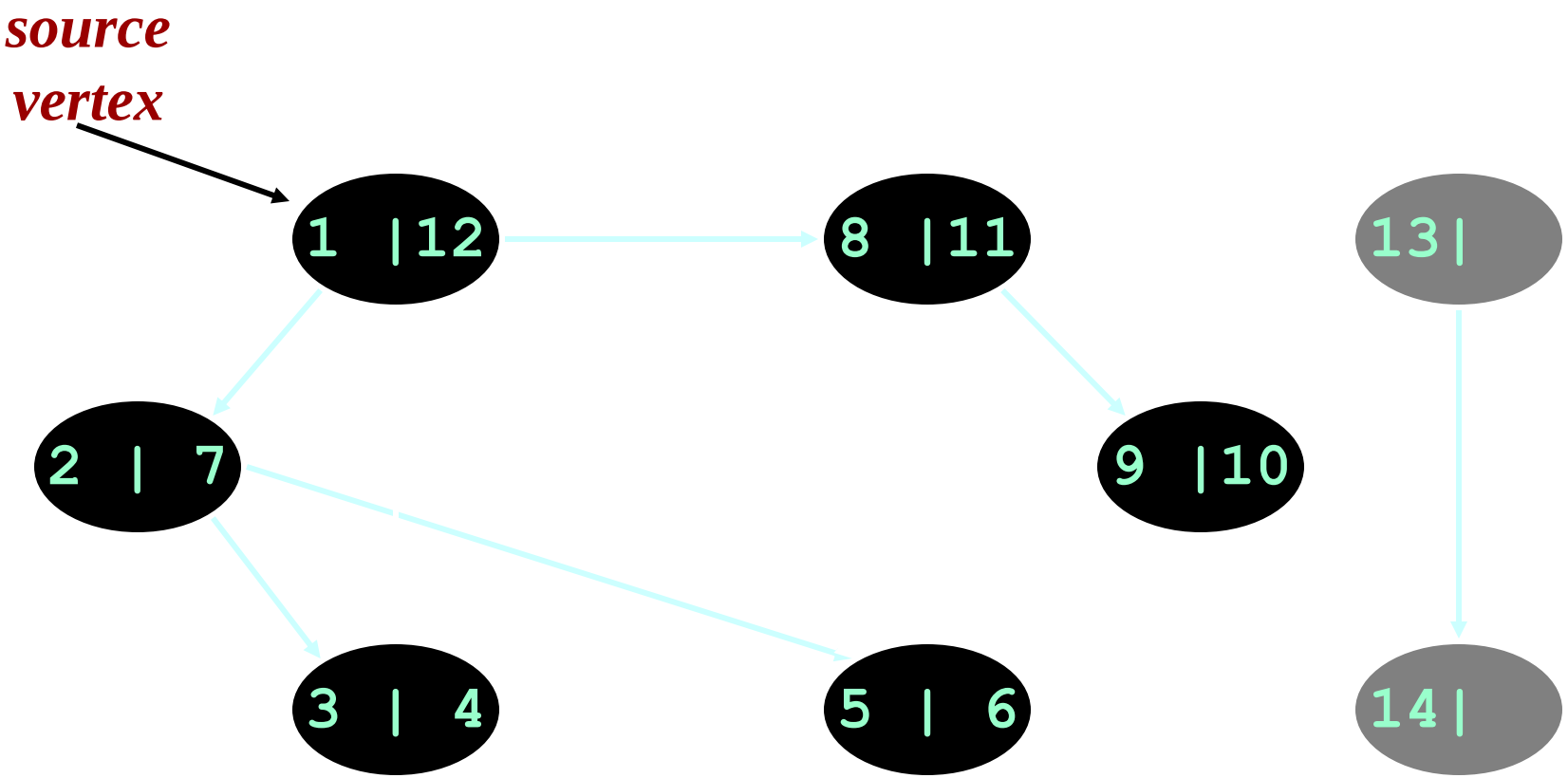
DFS Example



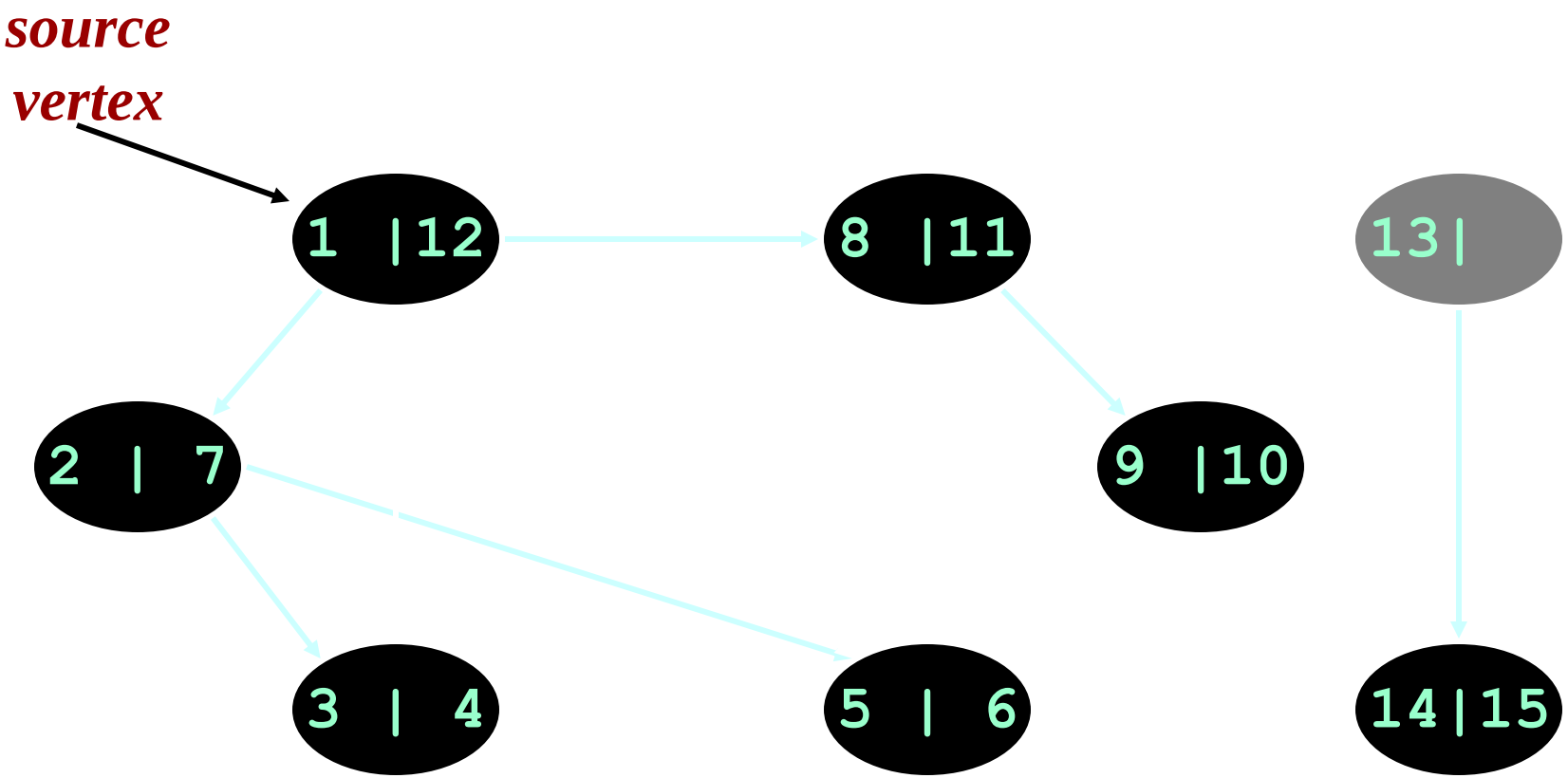
DFS Example



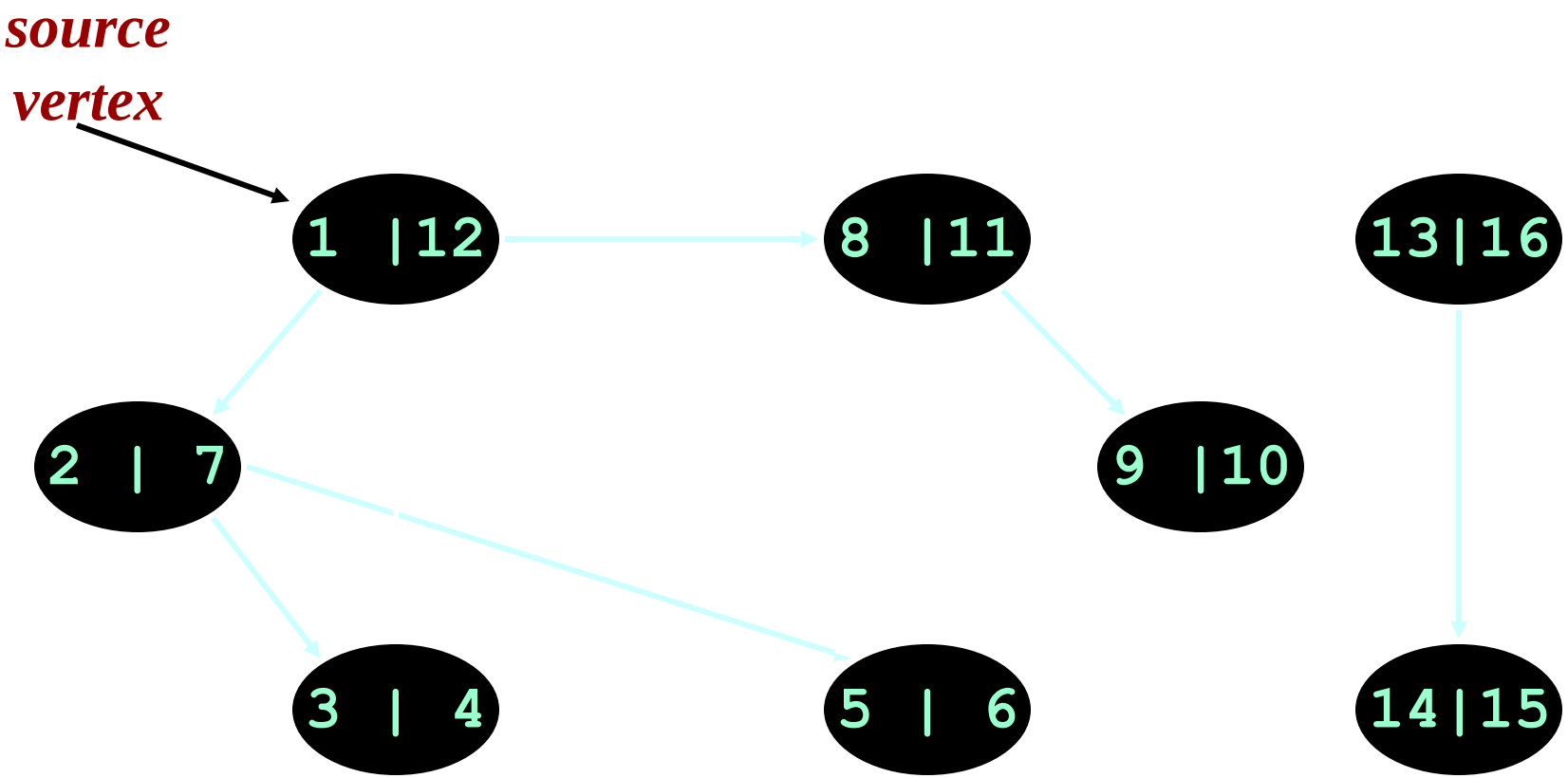
DFS Example



DFS Example



DFS Example



DFS: Algorithm

DFS: Algorithm (Cont.)

DFS: Complexity Analysis

DFS: Application

- Topological Sort
- Strongly Connected Component

Breadth-first Search (BFS)

- Search for all vertices that are directly reachable from the root (called level 1 vertices)
- After mark all these vertices, visit all vertices that are directly reachable from any level 1 vertices (called level 2 vertices), and so on.
- In general, level k vertices are directly reachable from a level k – 1 vertices

BFS: the Color Scheme

- White vertices have not been discovered
  - All vertices start out white
- Grey vertices are discovered but not fully explored
  - They may be adjacent to white vertices
- Black vertices are discovered and fully explored
  - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

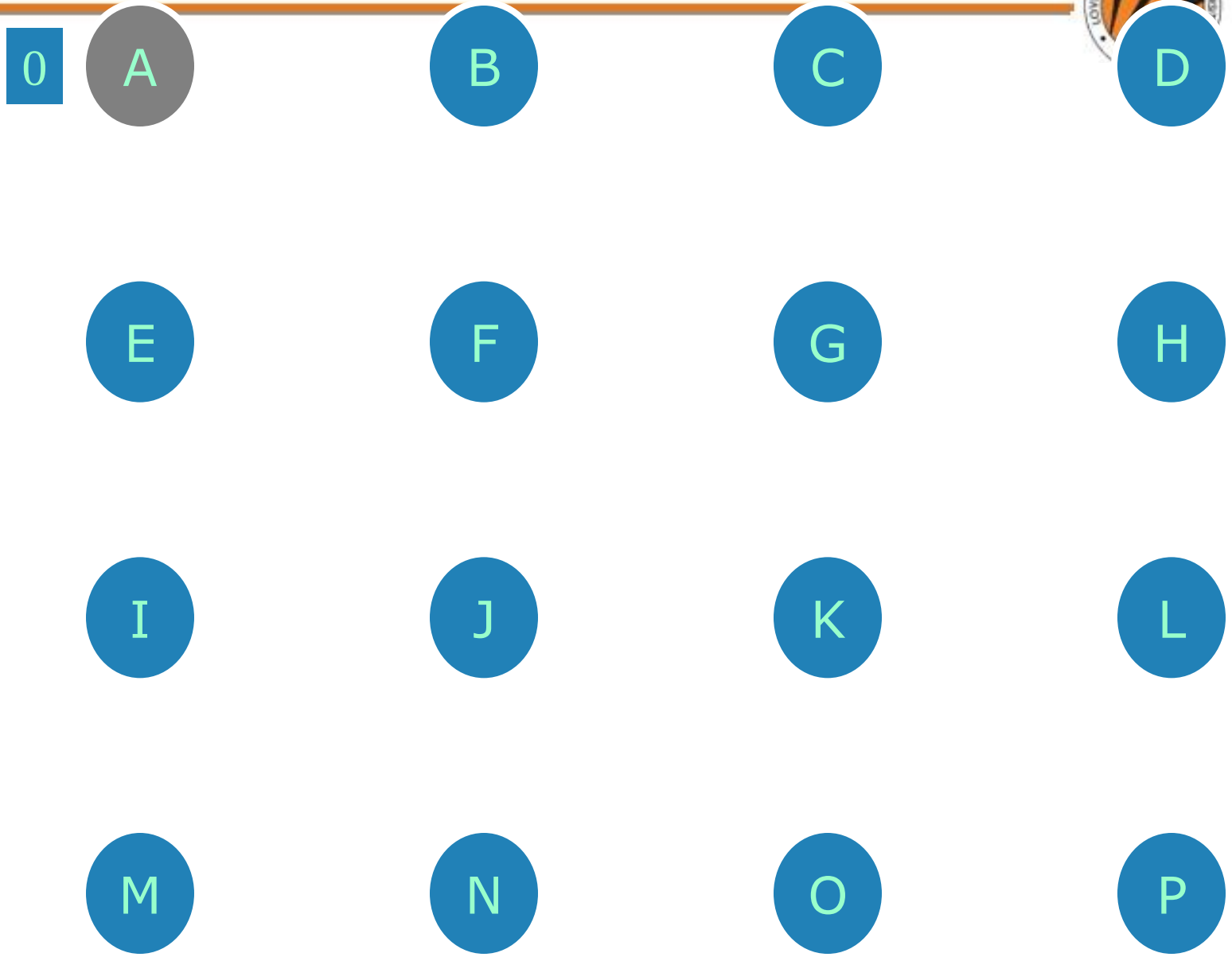
An Example

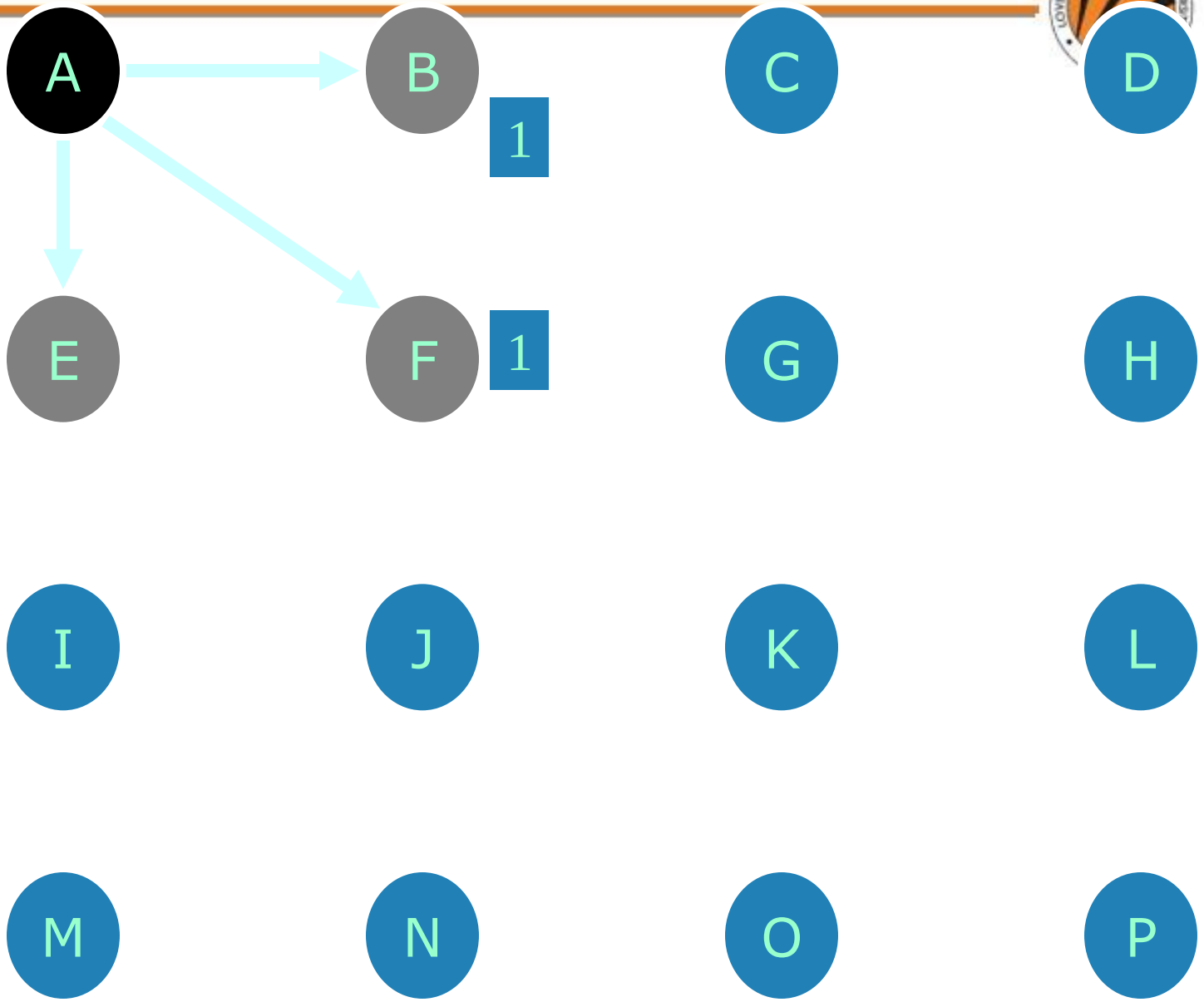
A B C D

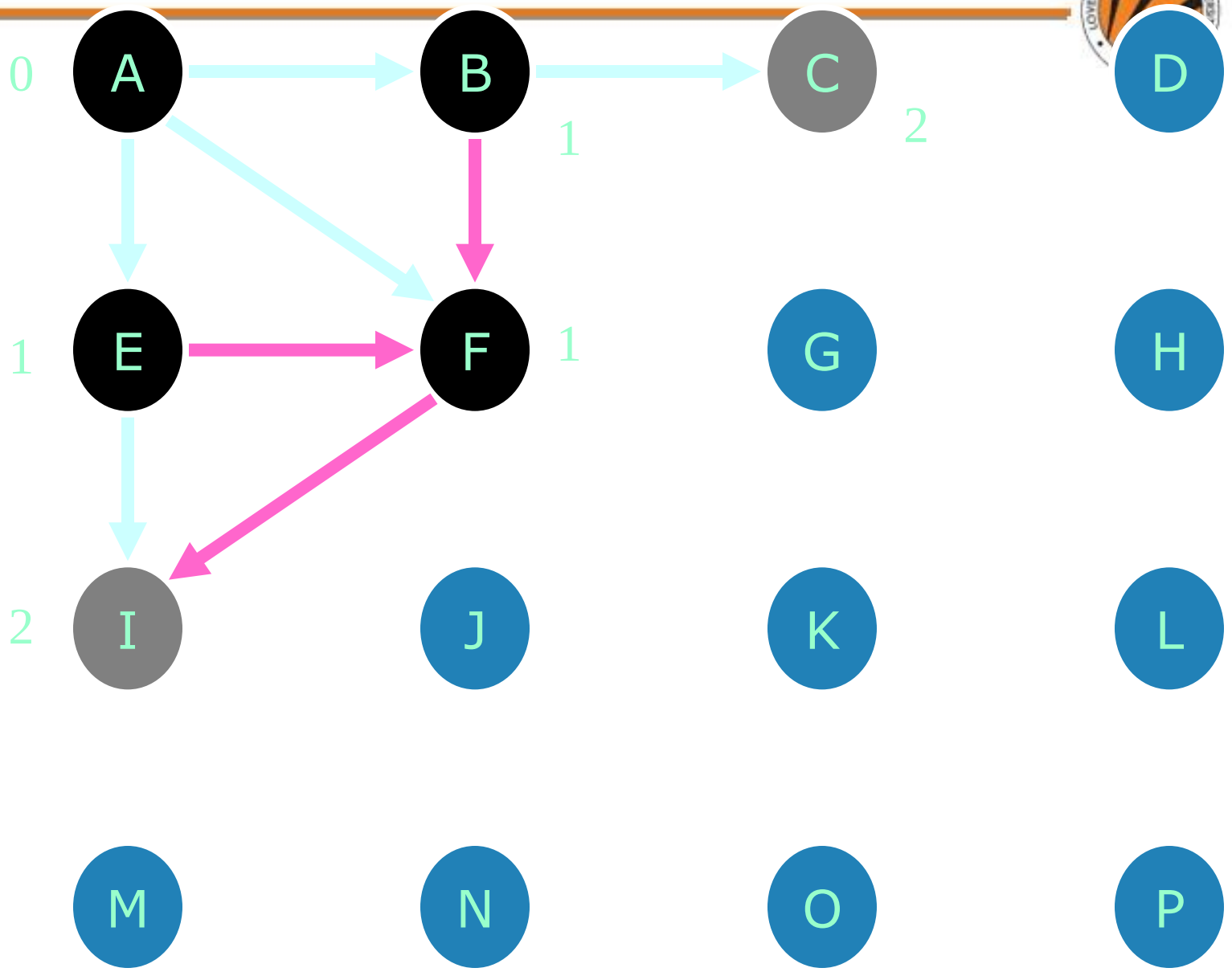
E F G H

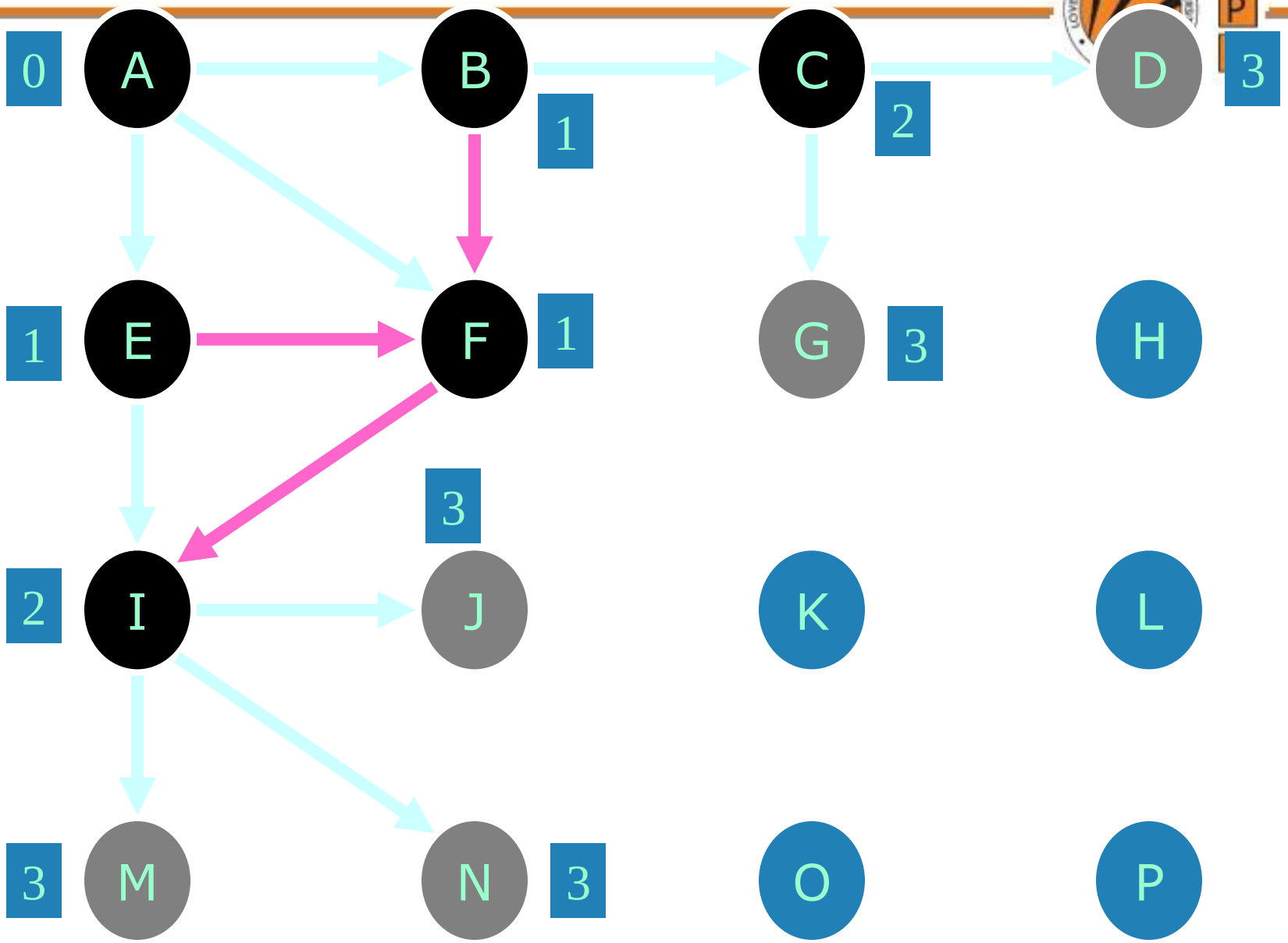
I J K L

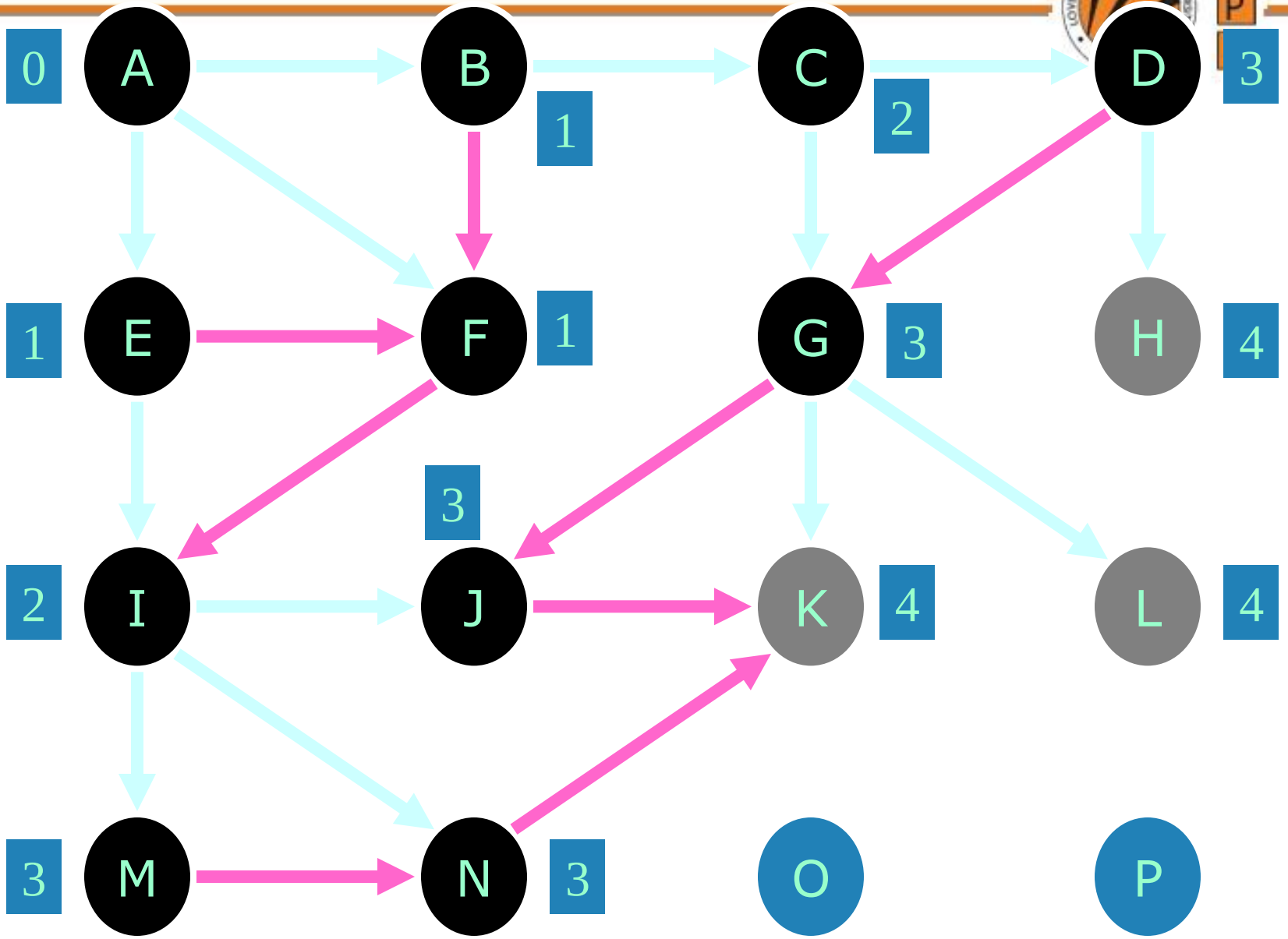
M N O P

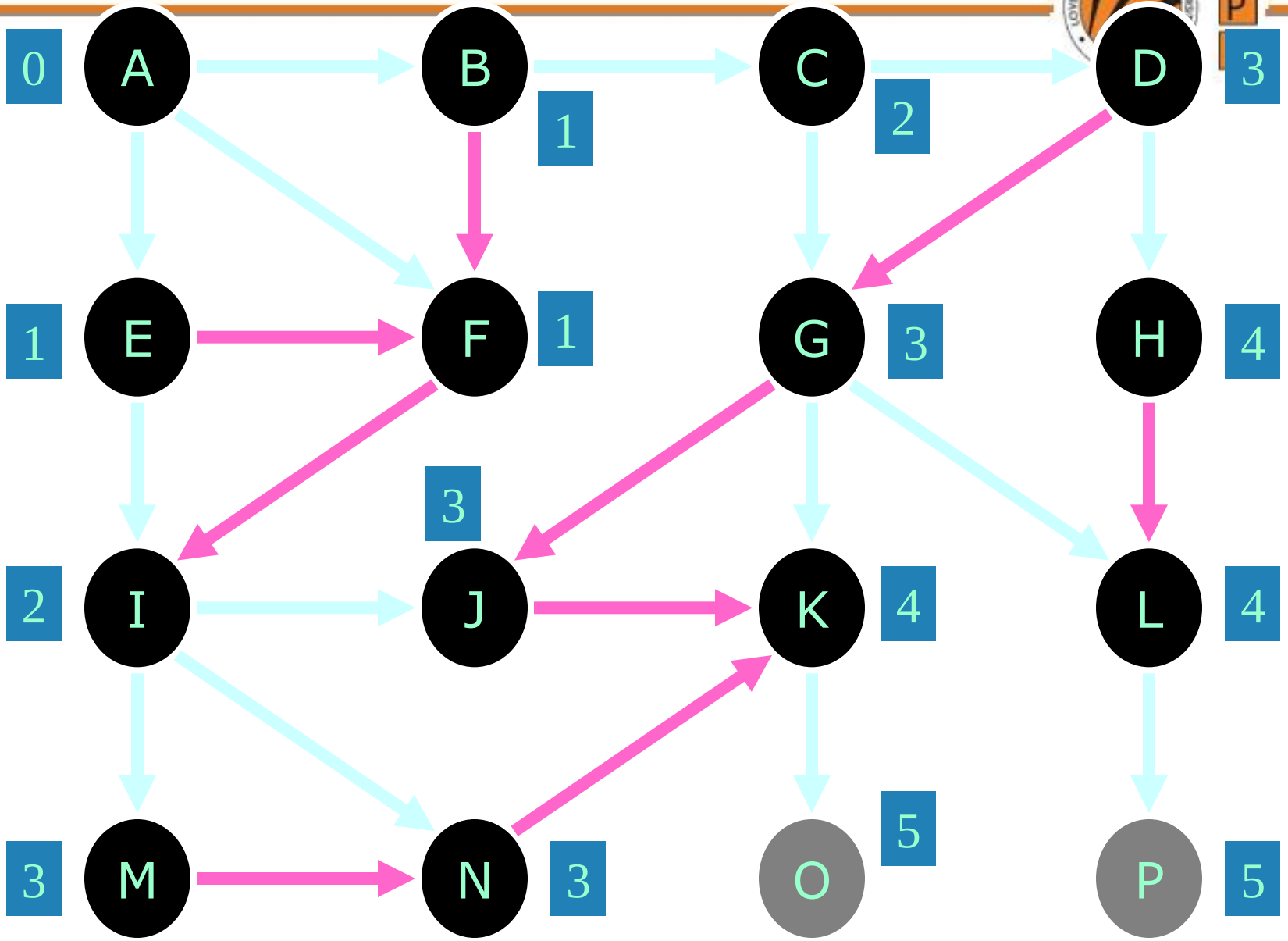

0
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

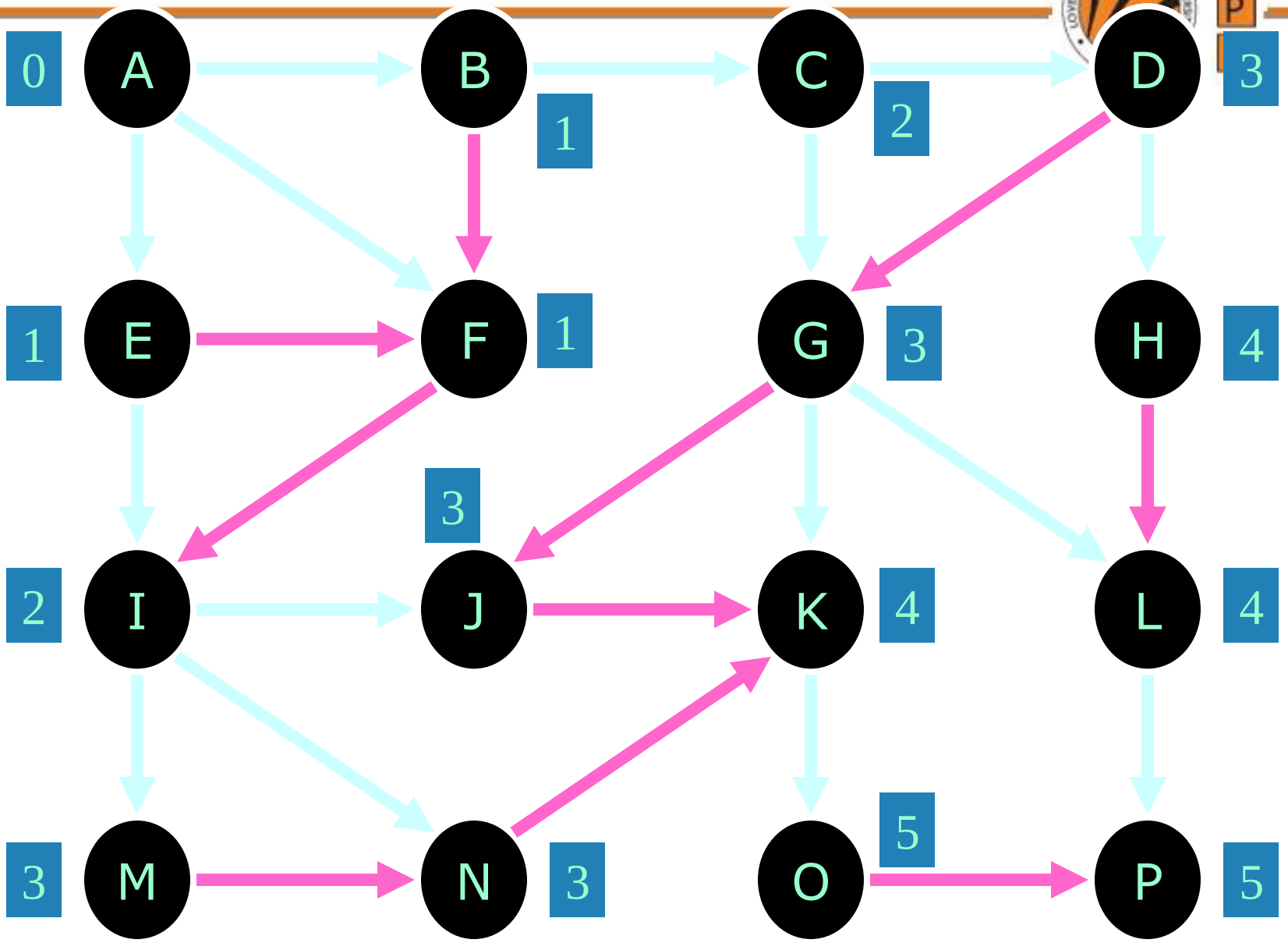

A
B
1
C
D
E
F
1
G
H
I
J
K
L
M
N
O
P


0
A
B
C
D
1
2
1
E
F
G
H
1
2
I
J
K
L
M
N
O
P


0
A
B
1
C
2
D
3
1
E
F
1
G
3
H
3
2
I
J
K
L
3
M
N
3
O
P


0
A
B
1
C
2
D
3
1
E
F
1
G
3
H
4
3
2
I
J
K
4
L
4
3
M
N
3
O
P


0
A
B
1
C
2
D
3
1
E
F
1
G
3
H
4
3
2
I
J
K
4
L
4
3
M
N
3
O
5
P
5


0
A
B
1
C
2
D
3
1
E
F
1
G
3
H
4
3
2
I
J
K
4
L
4
3
M
N
3
O
5
P
5

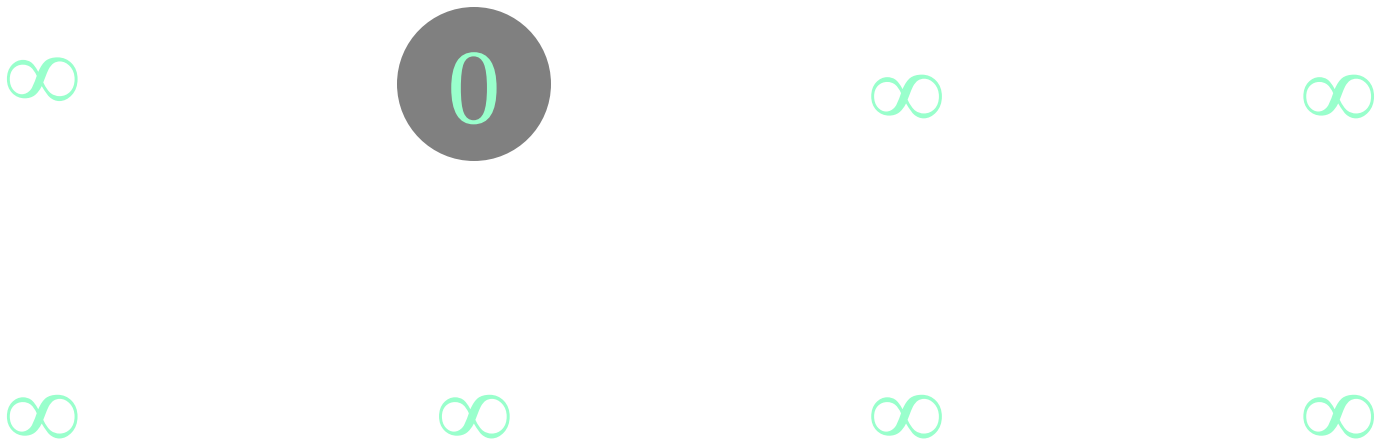
BFS: Algorithm

BFS(G, s)

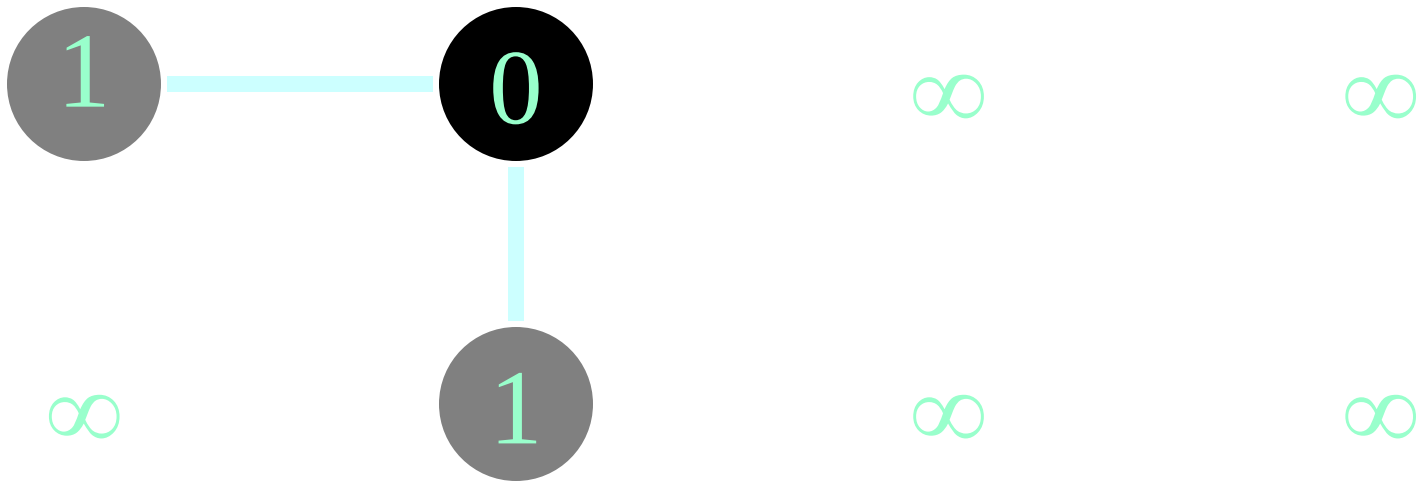
- For each vertex u in V – {s},
  - color[u] = white;
  - d[u] = infinity;
  - π[u] = NIL
- color[s] = GRAY; d[s] = 0; π[s] = NIL; Q = empty queue
- ENQUEUE(Q,s)
- while (Q not empty)
- u = DEQUEUE(Q)
- for each v ∈ Adj[u]
- if color[v] = WHITE
- then color[v] = GREY
- d[v] = d[u] + 1; π[v] = u
- ENQUEUE(Q, v);
- color[u] = BLACK;
-

Example

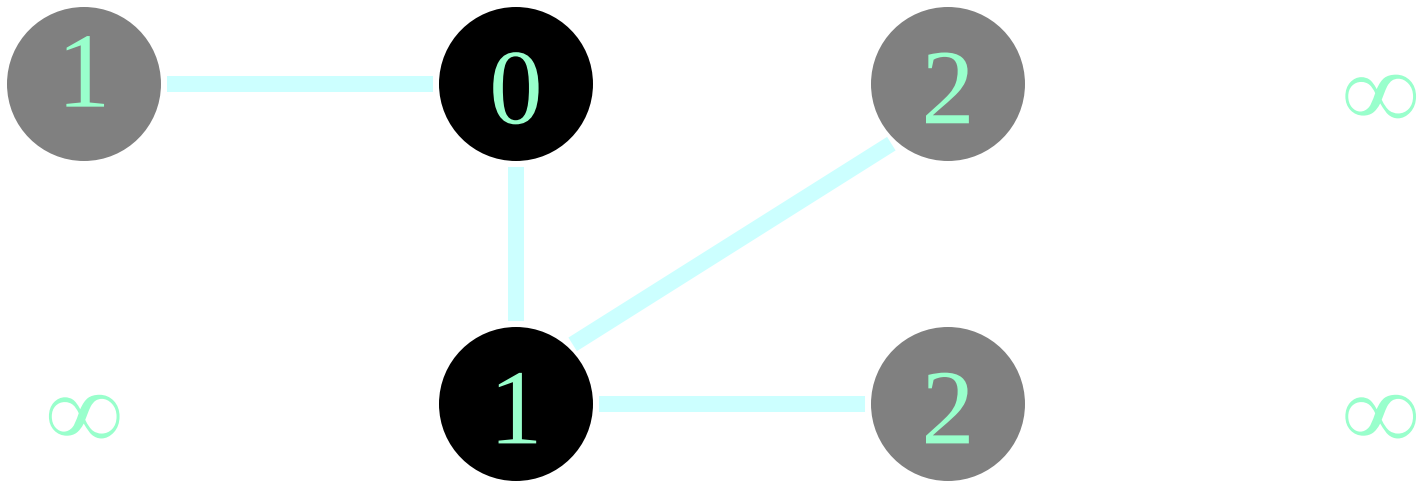
Example



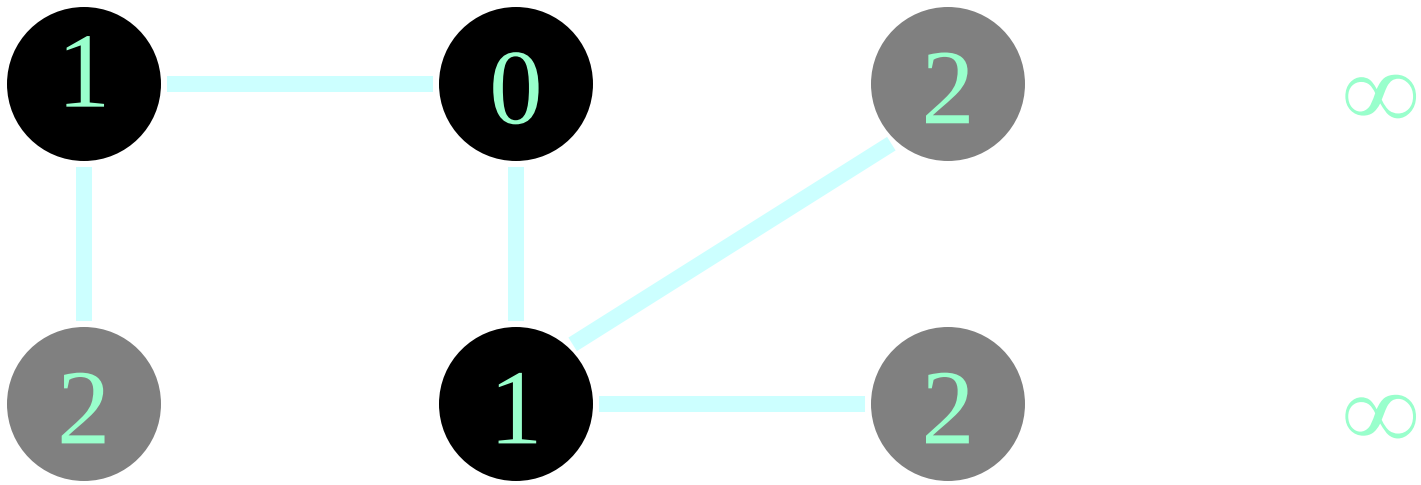
Example



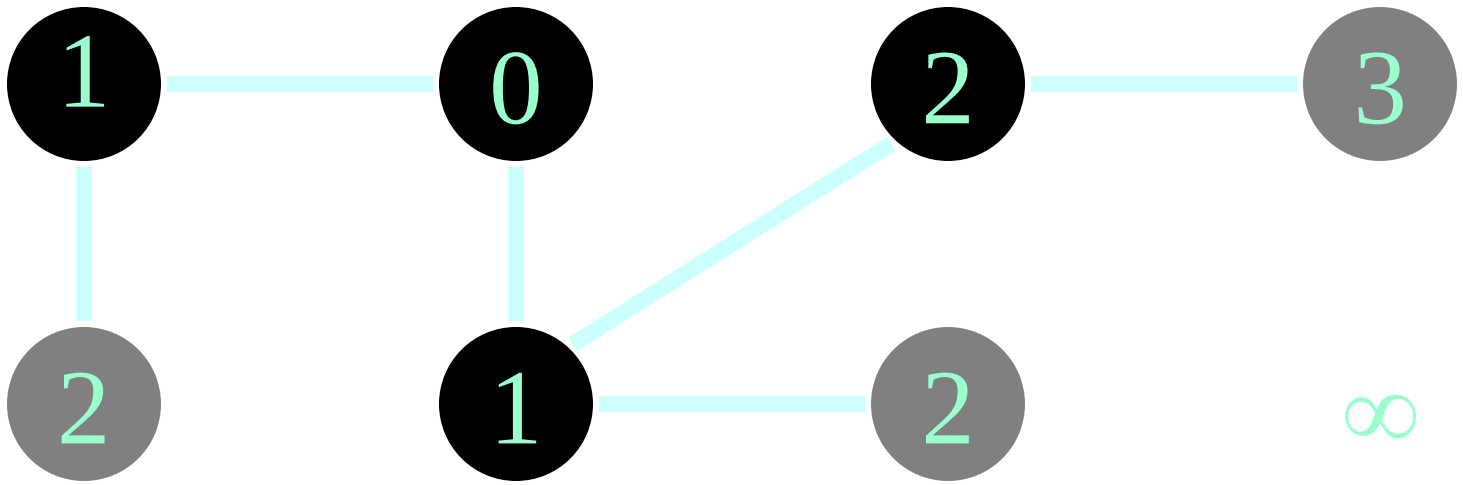
Example



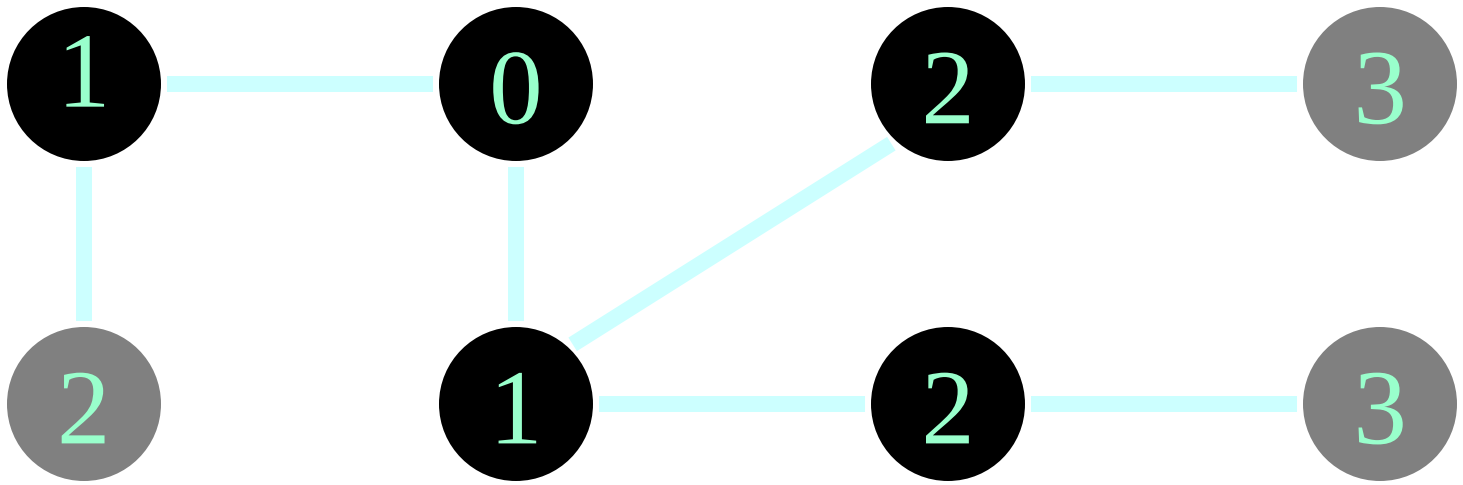
Example



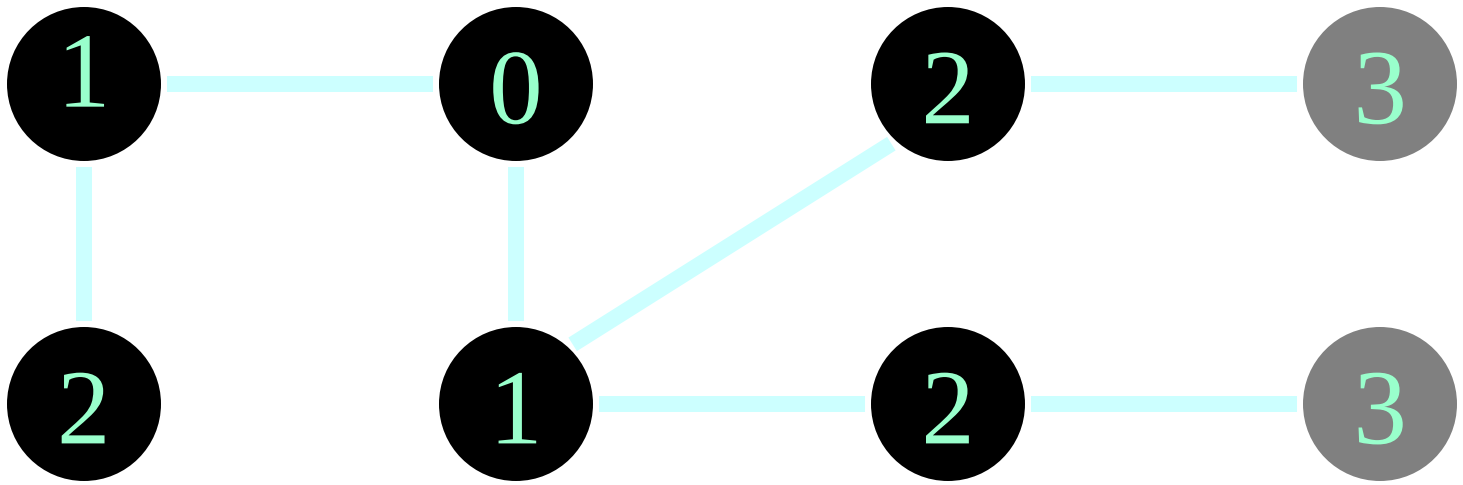
Example



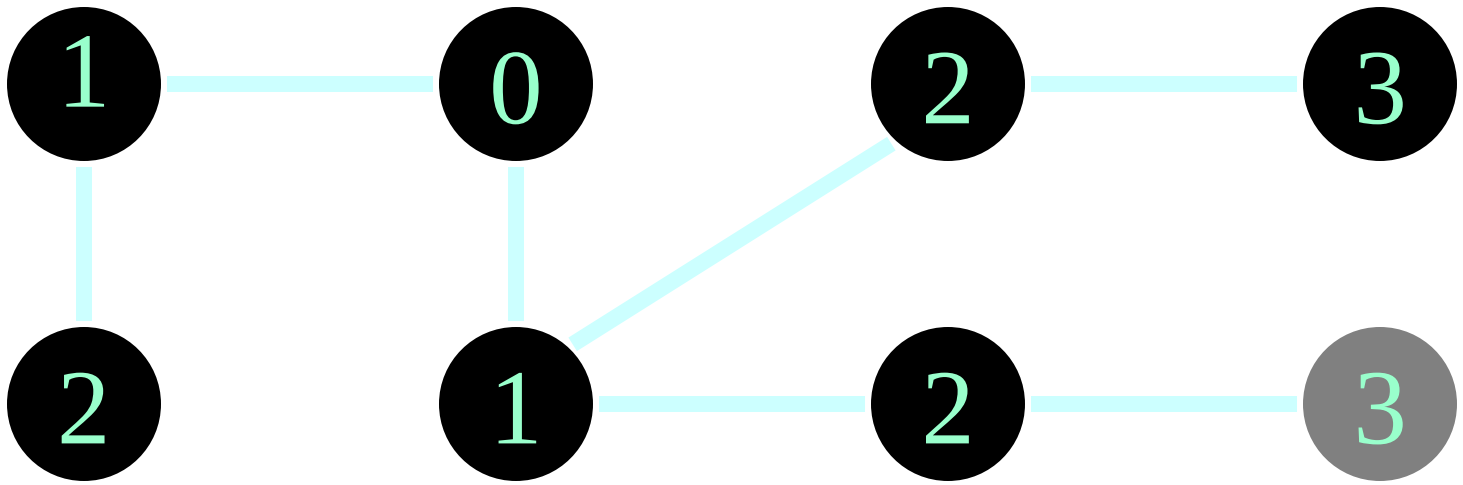
Example



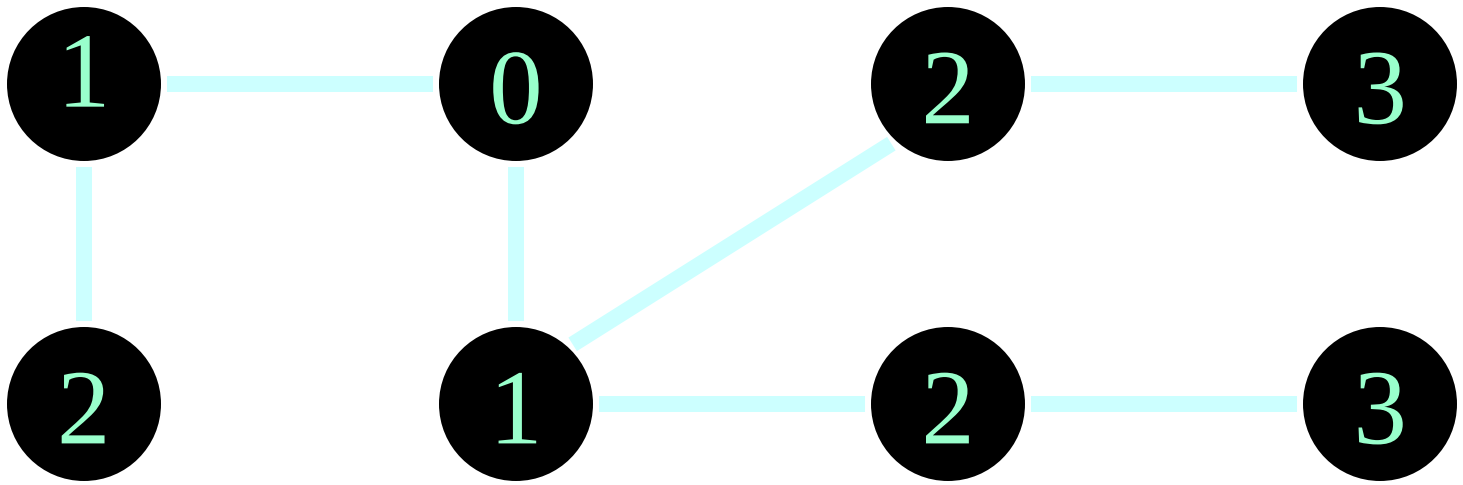
Example



Example



Example



BFS: Complexity Analysis

- Queuing time is O(V) and scanning all edges requires O(E)
- Overhead for initialization is O (V)
- So, total running time is O(V+E)

BFS: Application

- Shortest path problem

Thank You !!!

CSE408
Presorting& Balanced Search Tree

Transform and Conquer

- Algorithms based on the idea of transformation
  - Transformation stage
    - Problem instance is modified to be more amenable to solution
  - Conquering stage
    - Transformed problem is solved
- Major variations are for the transform to perform:
  - Instance simplification
  - Different representation
  - Problem reduction

Presorting

- Presorting is an old idea, you sort the data and that allows you to more easily compute some answer
  - Saw this with quickhull, closest point
- Some other simple presorting examples
  - Element Uniqueness
  - Computing the mode of *n* numbers

Element Uniqueness

- Given a list *A* of *n* orderable elements, determine if there are any duplicates of any element

Computing a mode

- A mode is a value that occurs most often in a list of numbers
  - e.g. the mode of [5, 1, 5, 7, 6, 5, 7] is 5
  - If several different values occur most often any of them can be considered the mode
- "Count Sort" approach: (assumes all values > 0; what if they aren't?)

Presort Computing Mode

AVL Animation Link

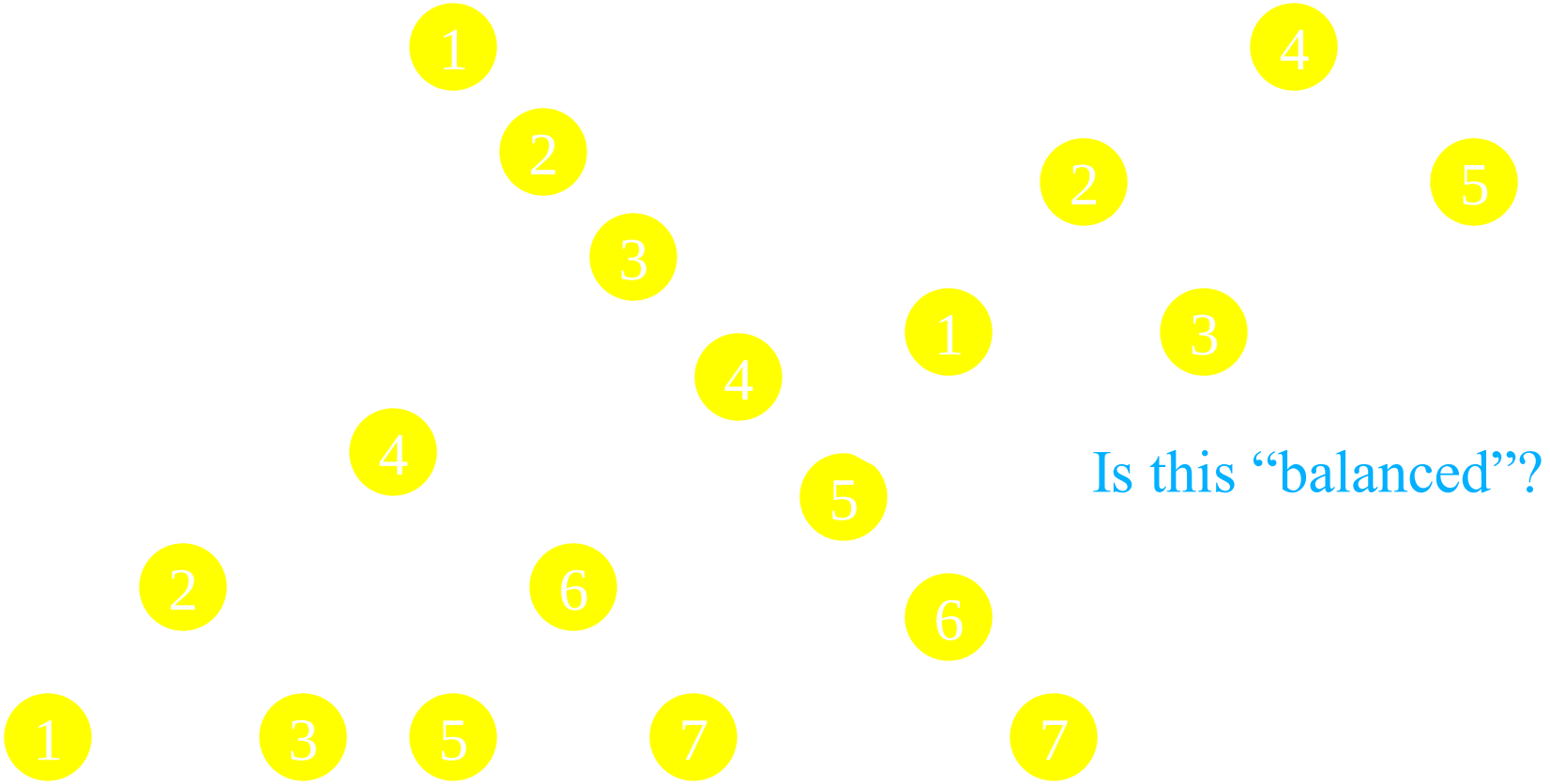
Binary Search Tree - Best Time

- All BST operations are O(d), where d is tree depth
- minimum d is $d = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

- Worst case running time is O(N)
  - What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of "balance":
    - compare depths of left and right subtree
  - Unbalanced degenerate tree

Balanced and unbalanced BST



Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree

AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

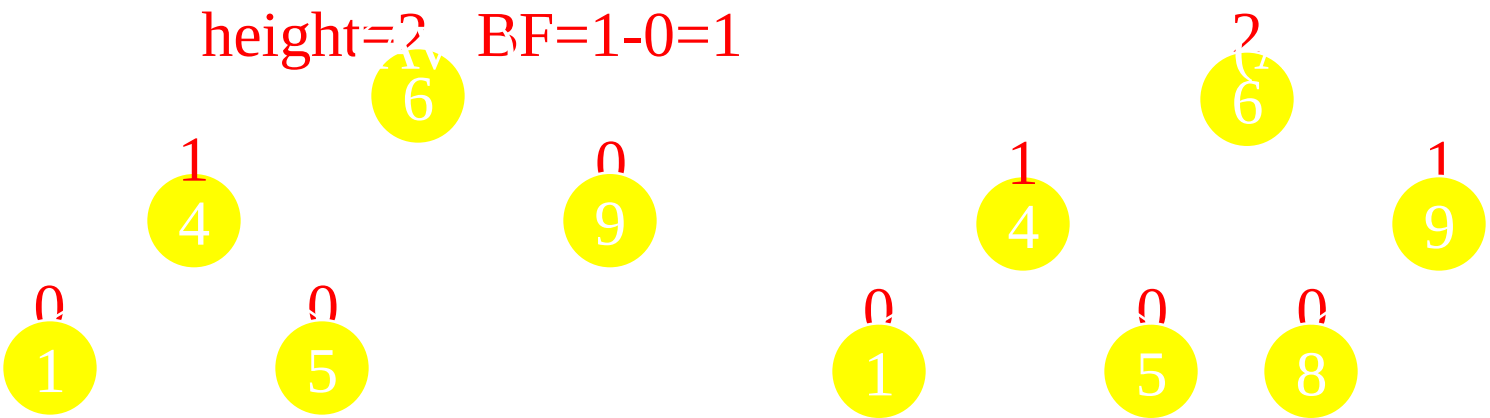
Height of an AVL Tree

- N(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - N(0) = 1, N(1) = 2
- Induction
  - N(h) = N(h-1) + N(h-2) + 1
- Solution (recall Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$

Height of an AVL Tree

- $N(h) \geq \phi^h \quad (\phi \approx 1.62)$
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(logn))

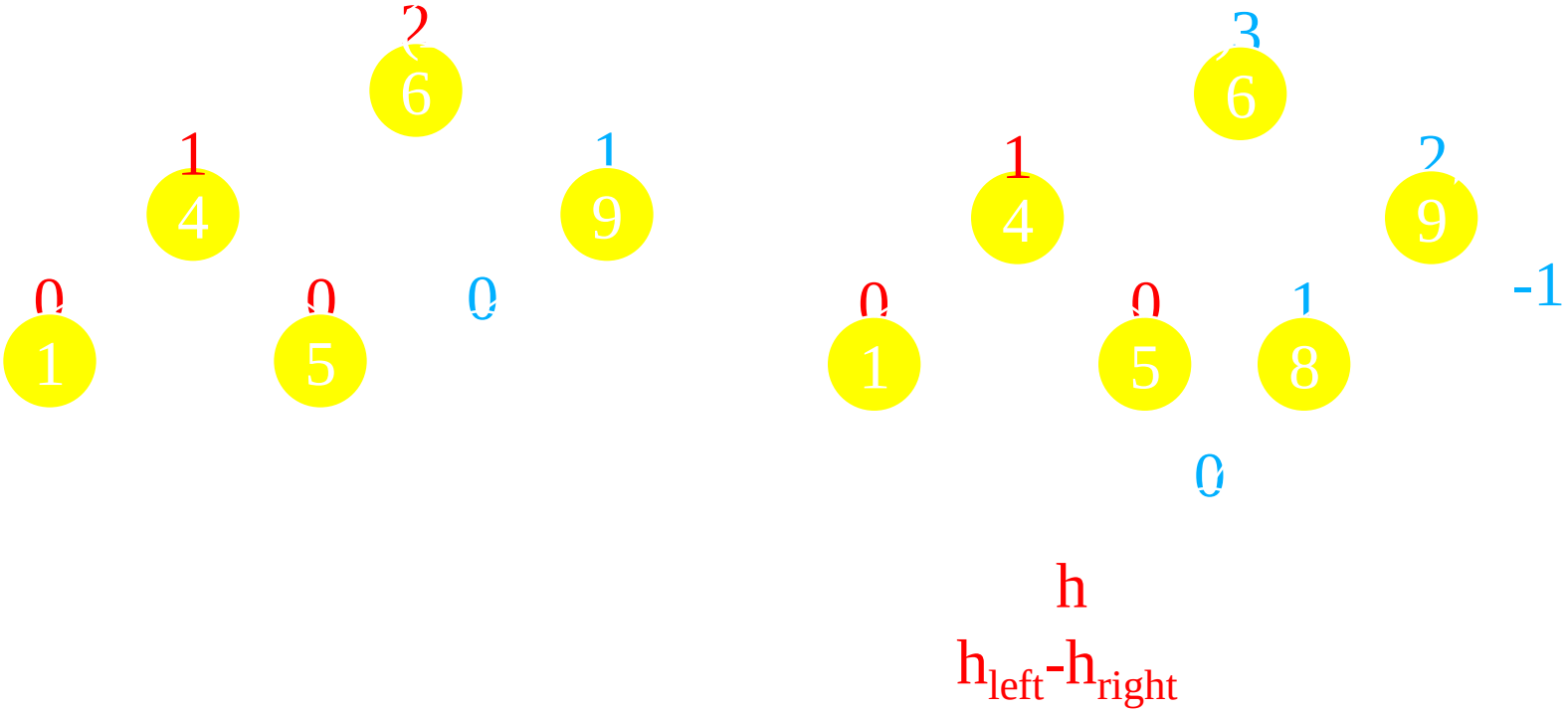
Node Heights



h

h_{left}-h_{right}

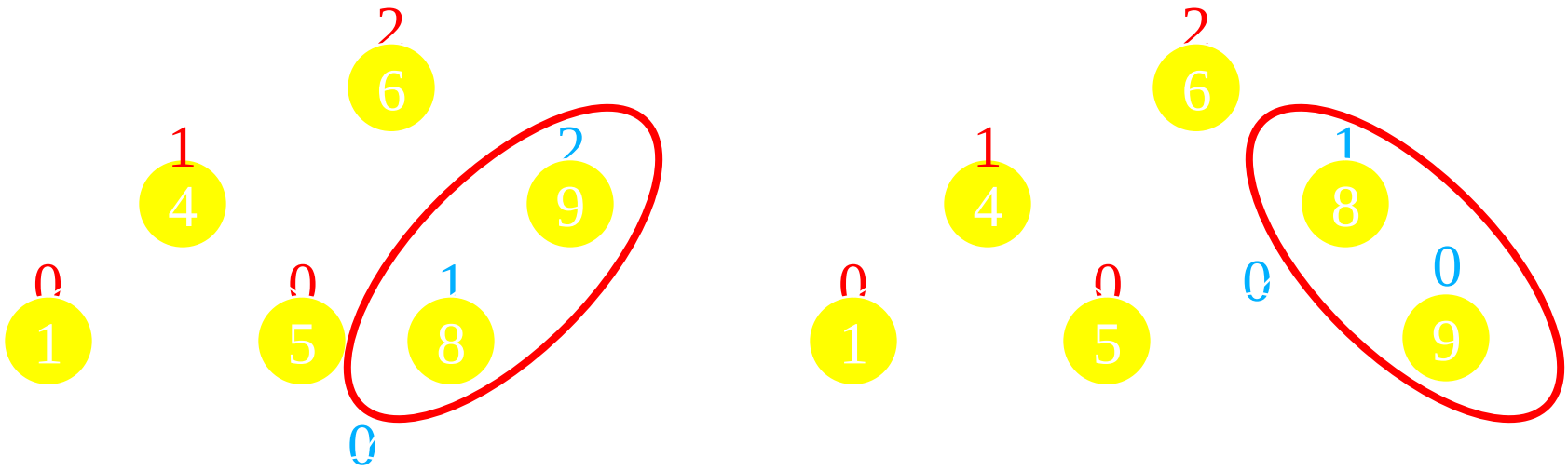
Node Heights after Insert 7



Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree



Insertions in AVL Trees

α

Outside Cases

left of left

right of right

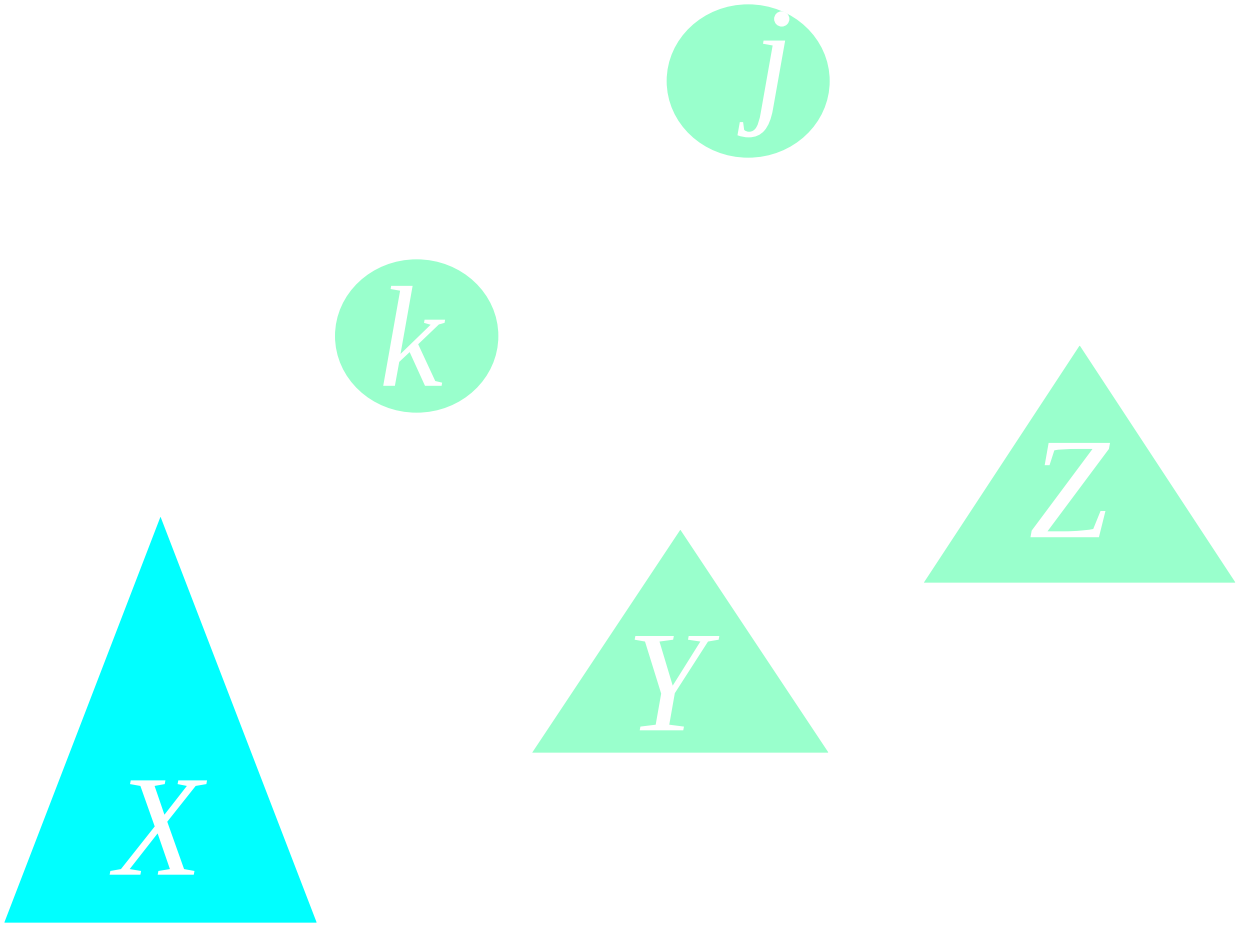
Inside Cases

right of left

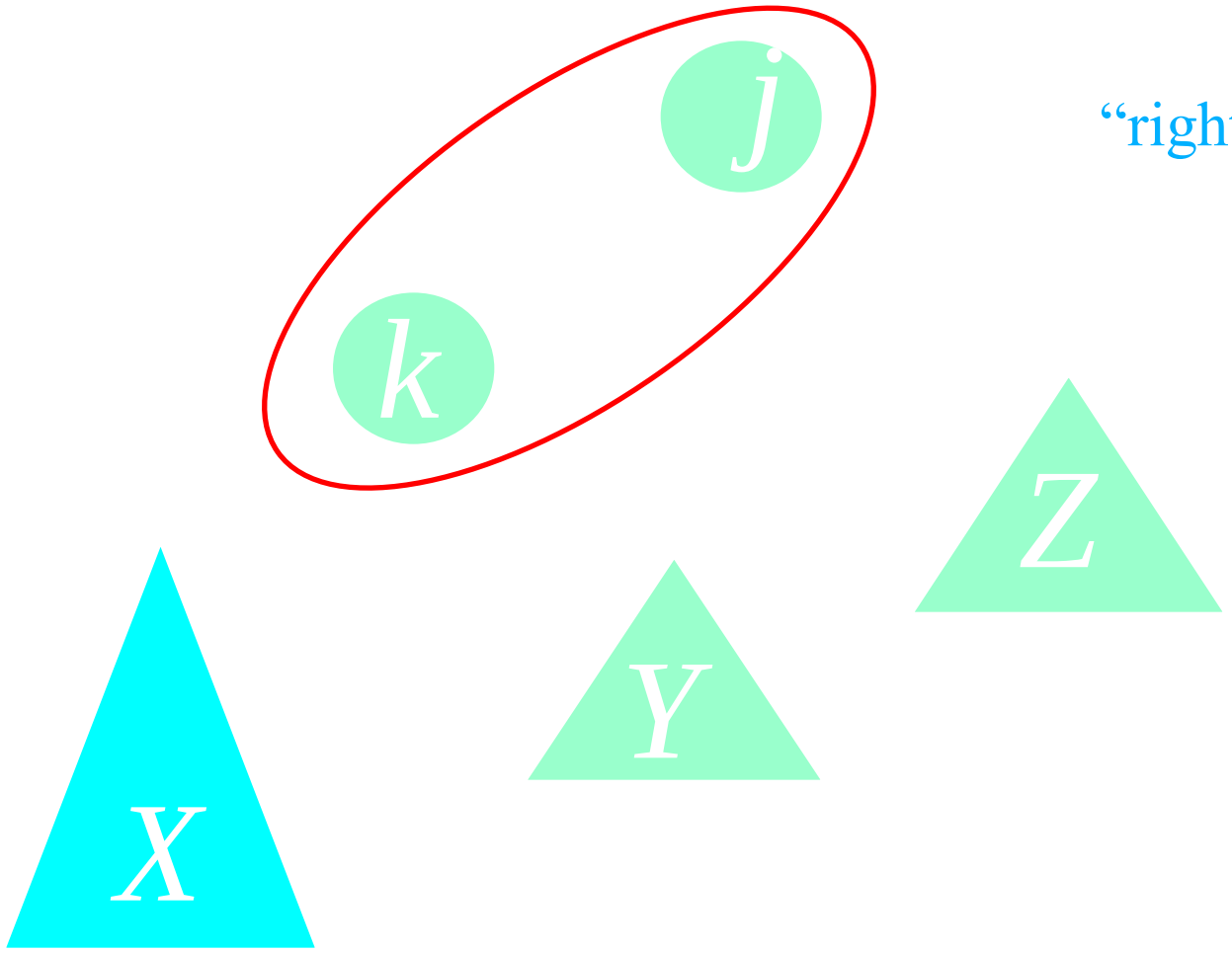
left of right

AVL Insertion: Outside Case

AVL Insertion: Outside Case

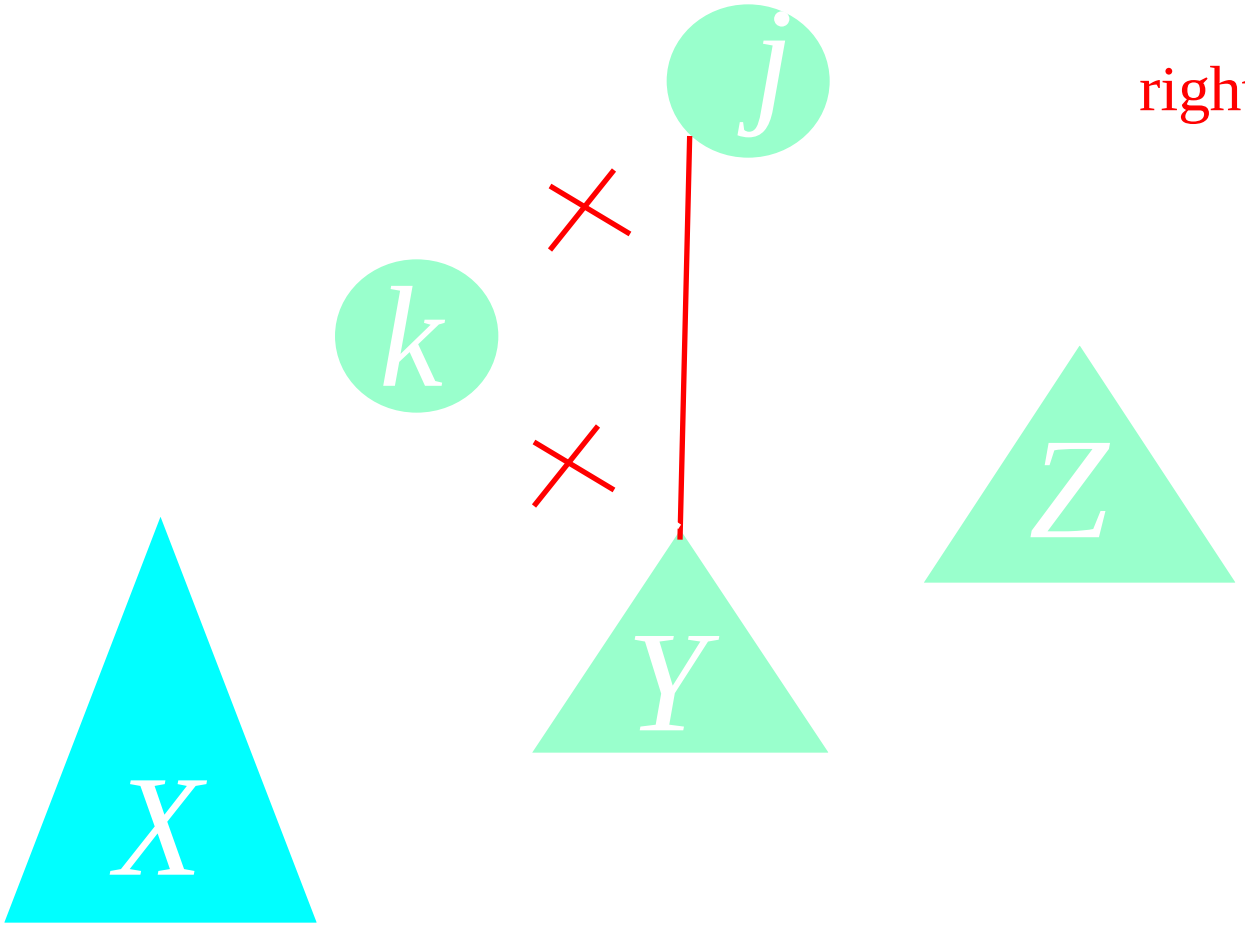


AVL Insertion: Outside Case



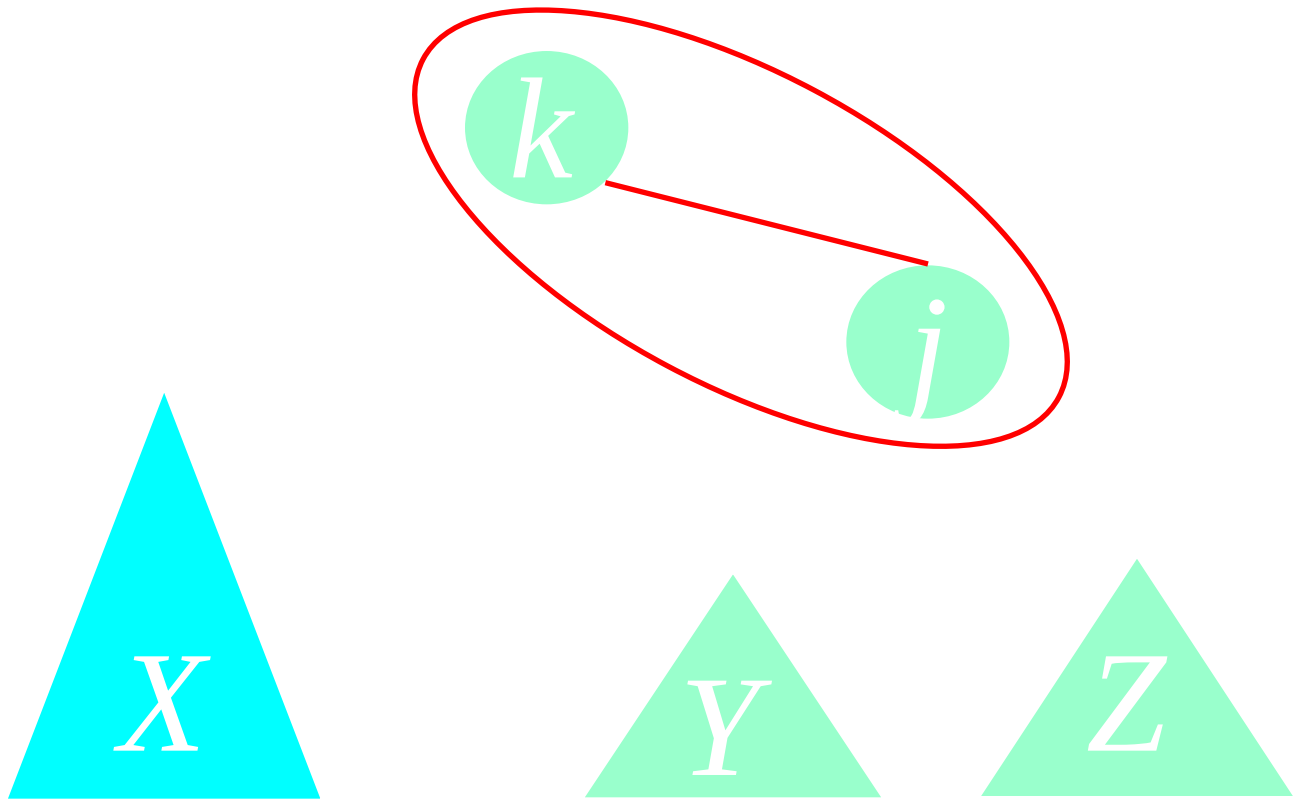
"right rotation"

Single right rotation



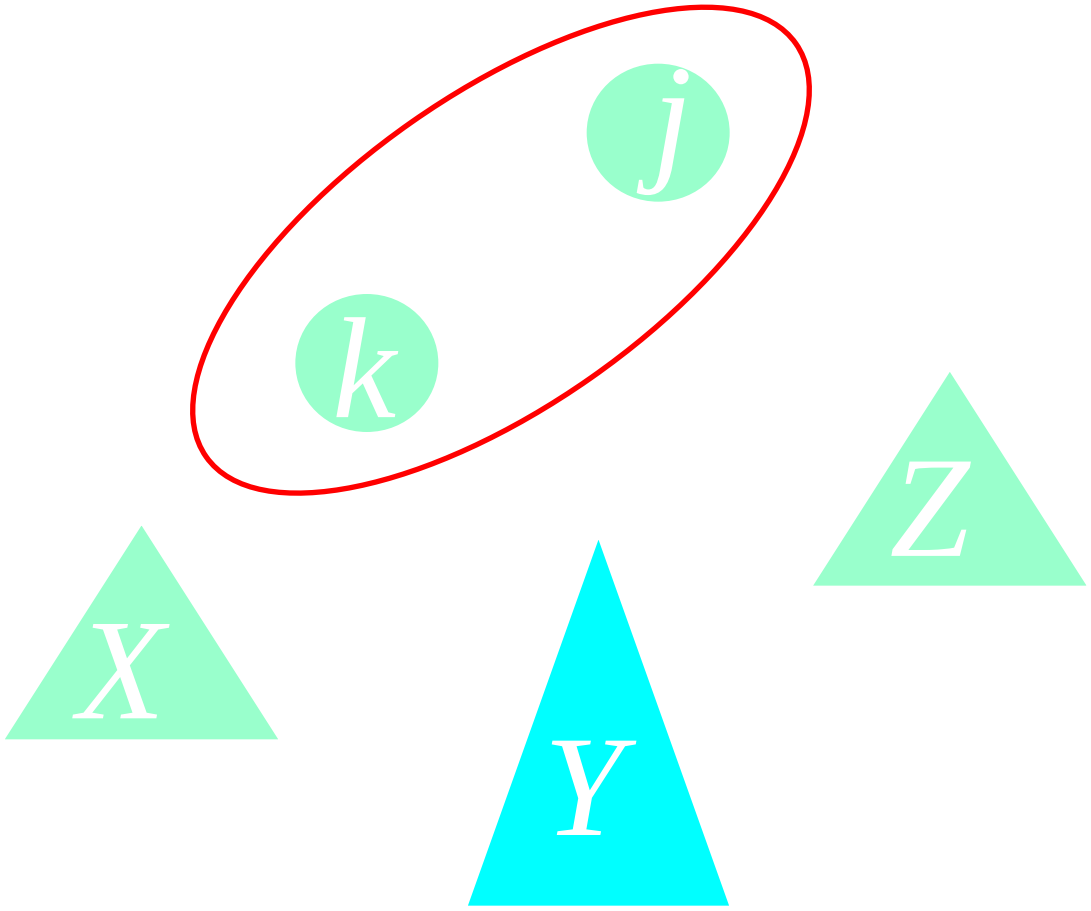
right rotation

Outside Case Completed

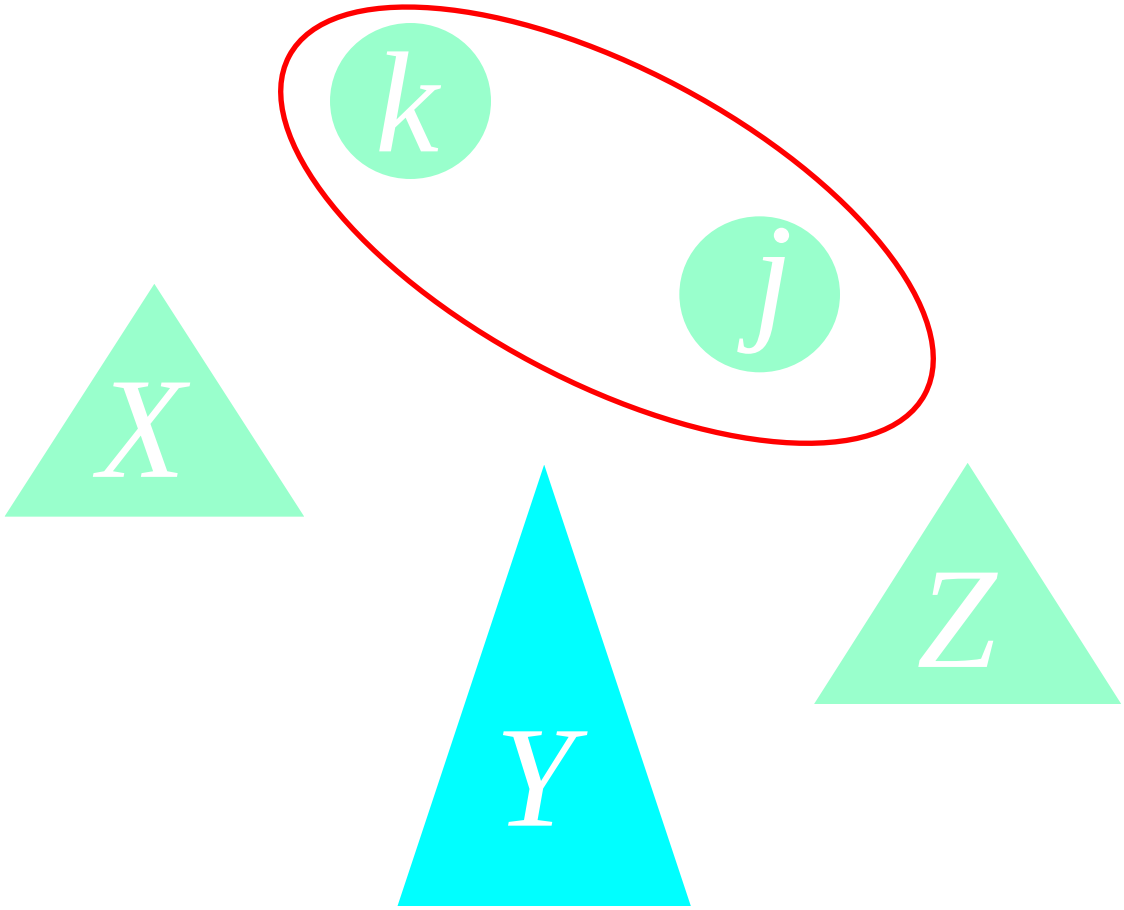


AVL Insertion: Inside Case

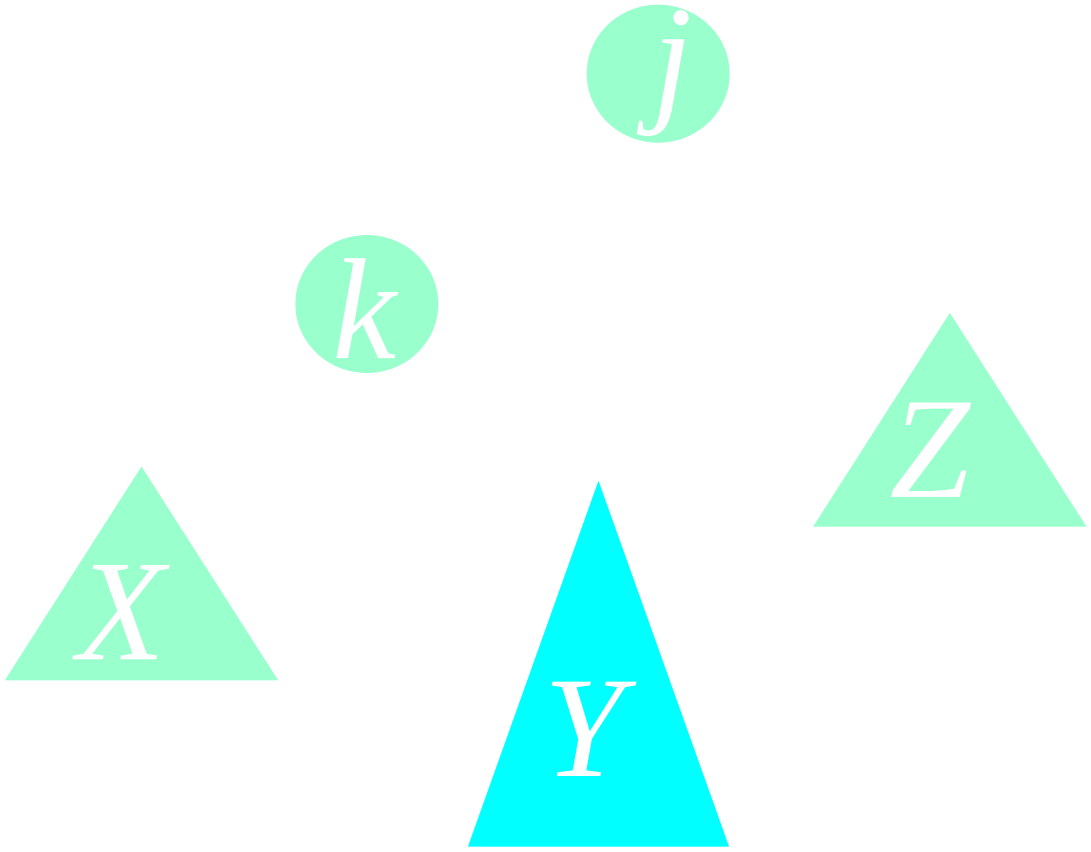
AVL Insertion: Inside Case



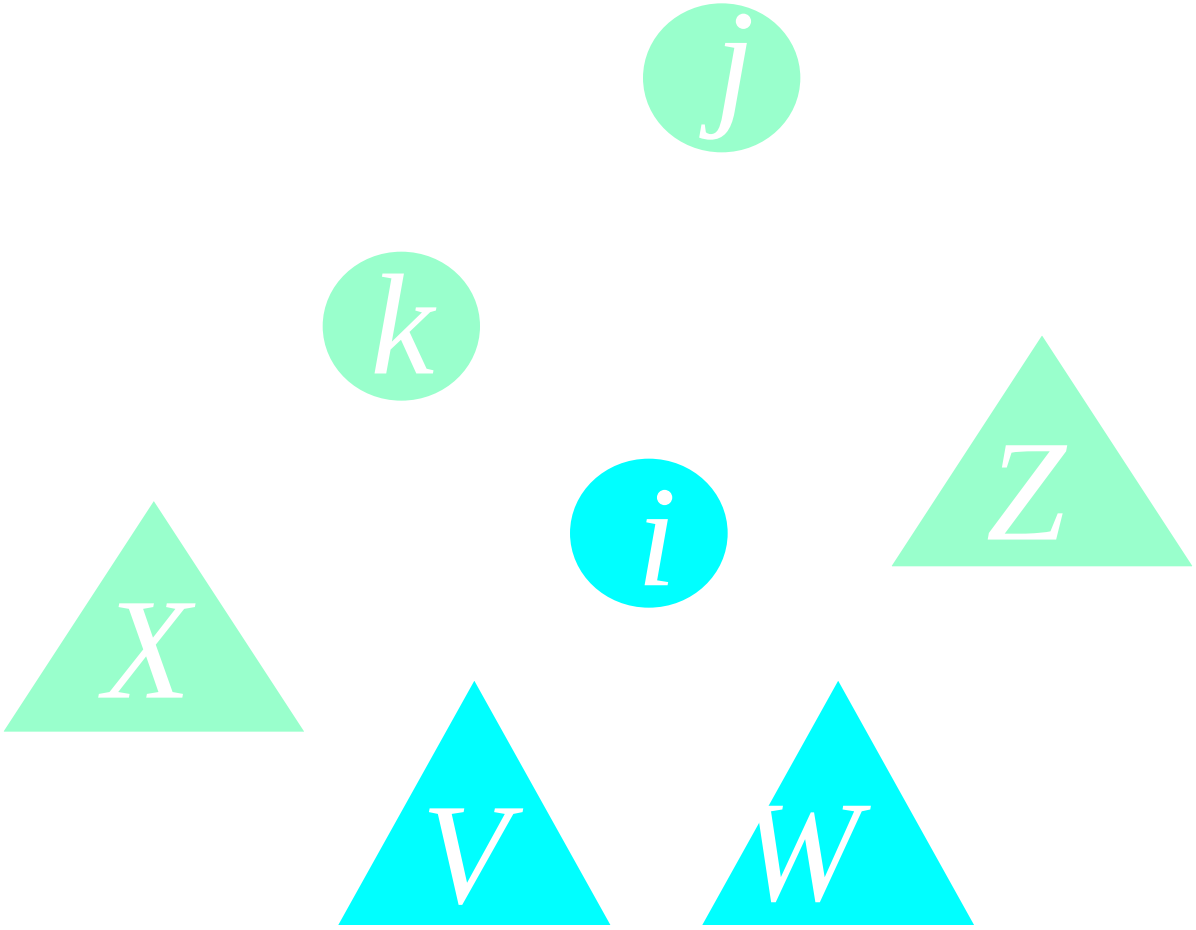
AVL Insertion: Inside Case



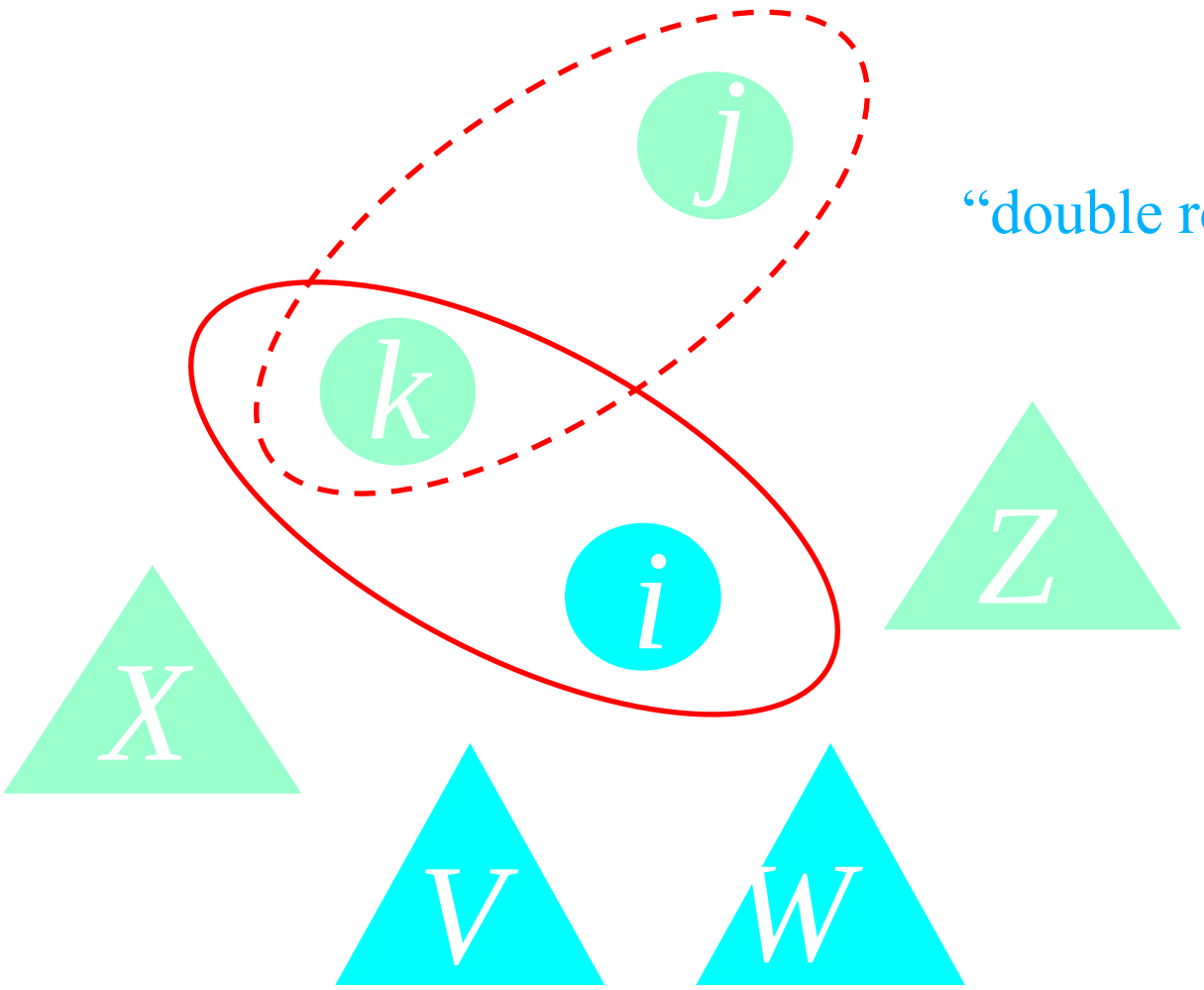
AVL Insertion: Inside Case



AVL Insertion: Inside Case

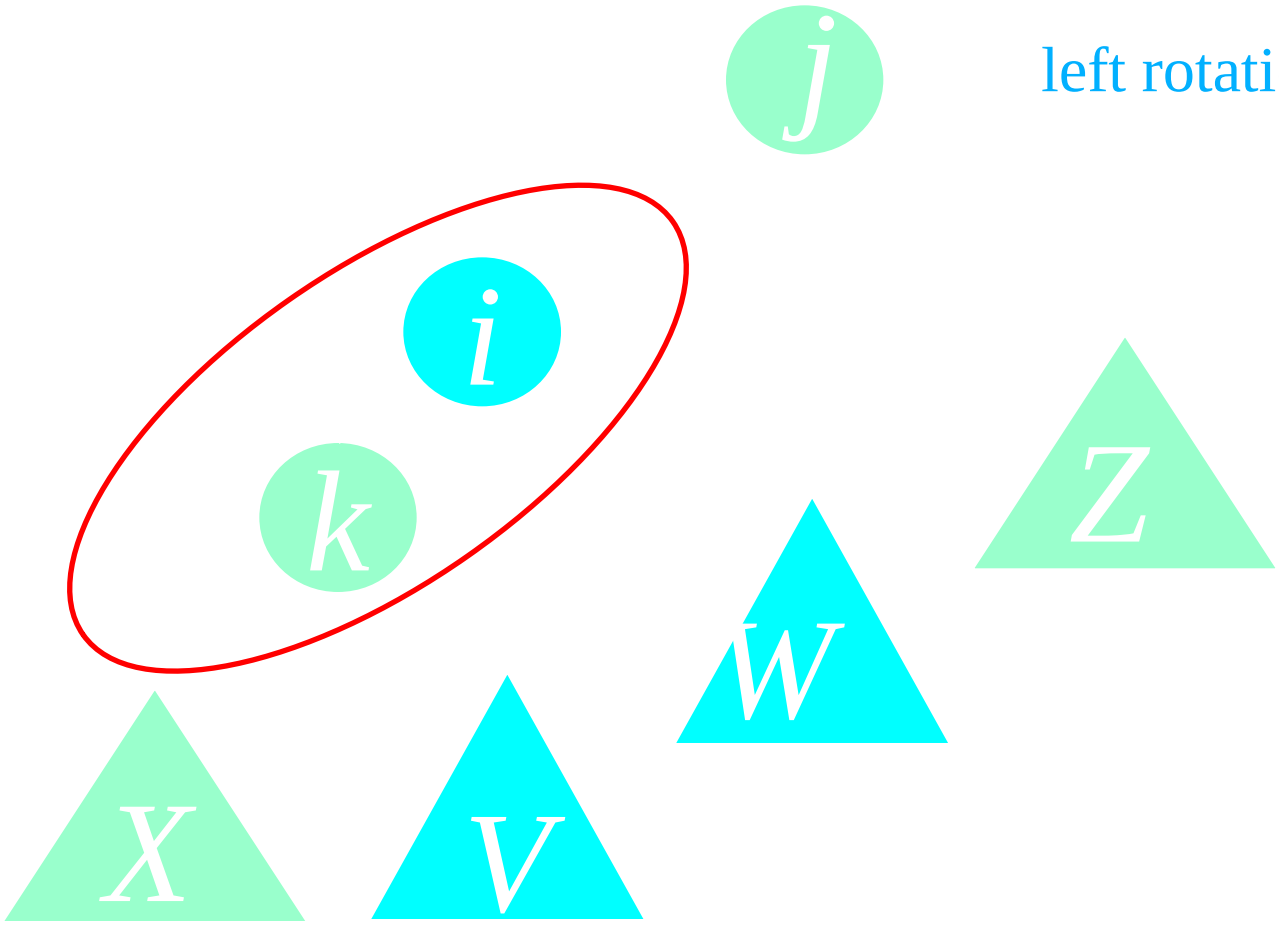


AVL Insertion: Inside Case



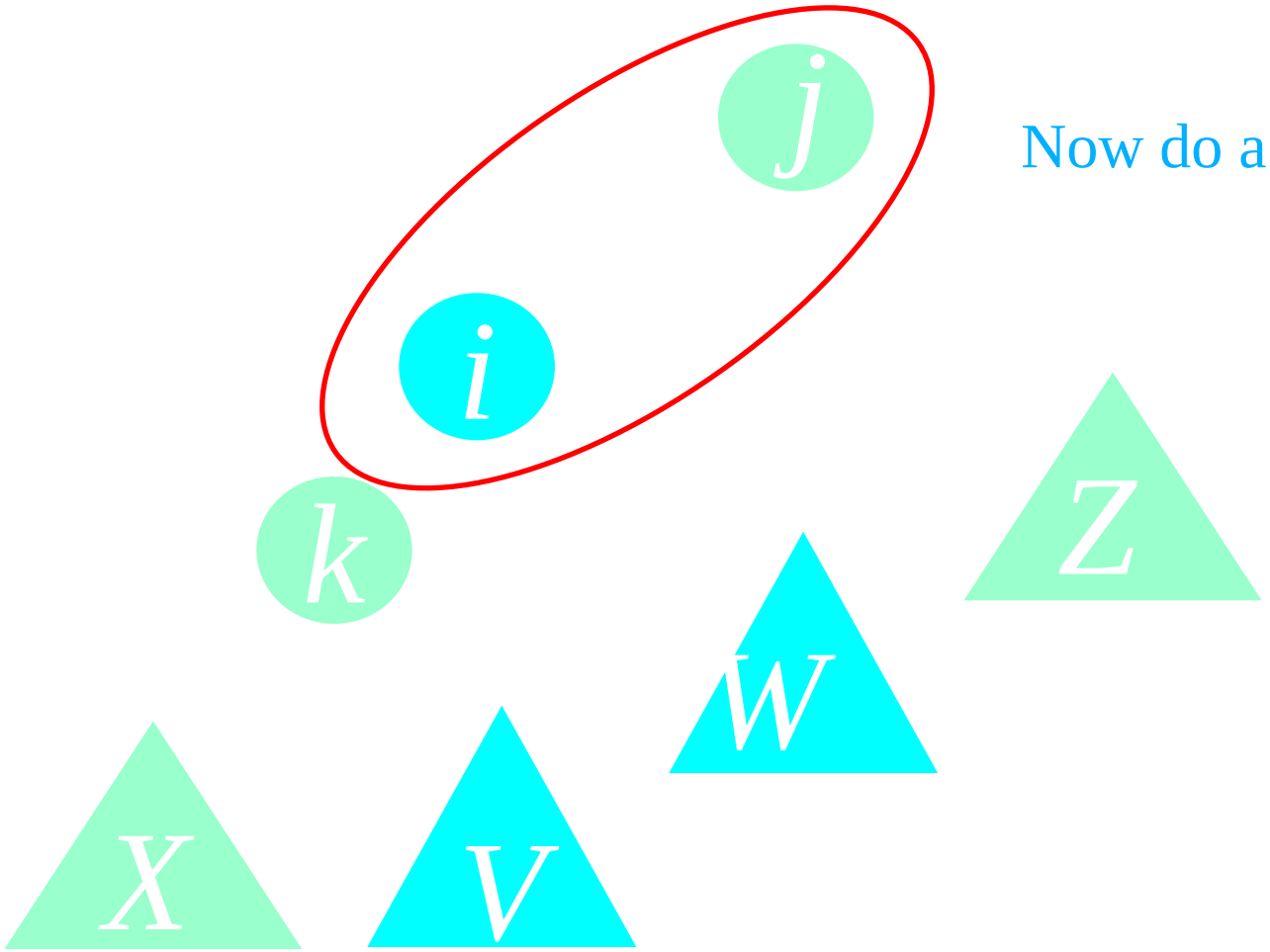
left-right
"double rotation" .

Double rotation : first rotation



left rotation complete

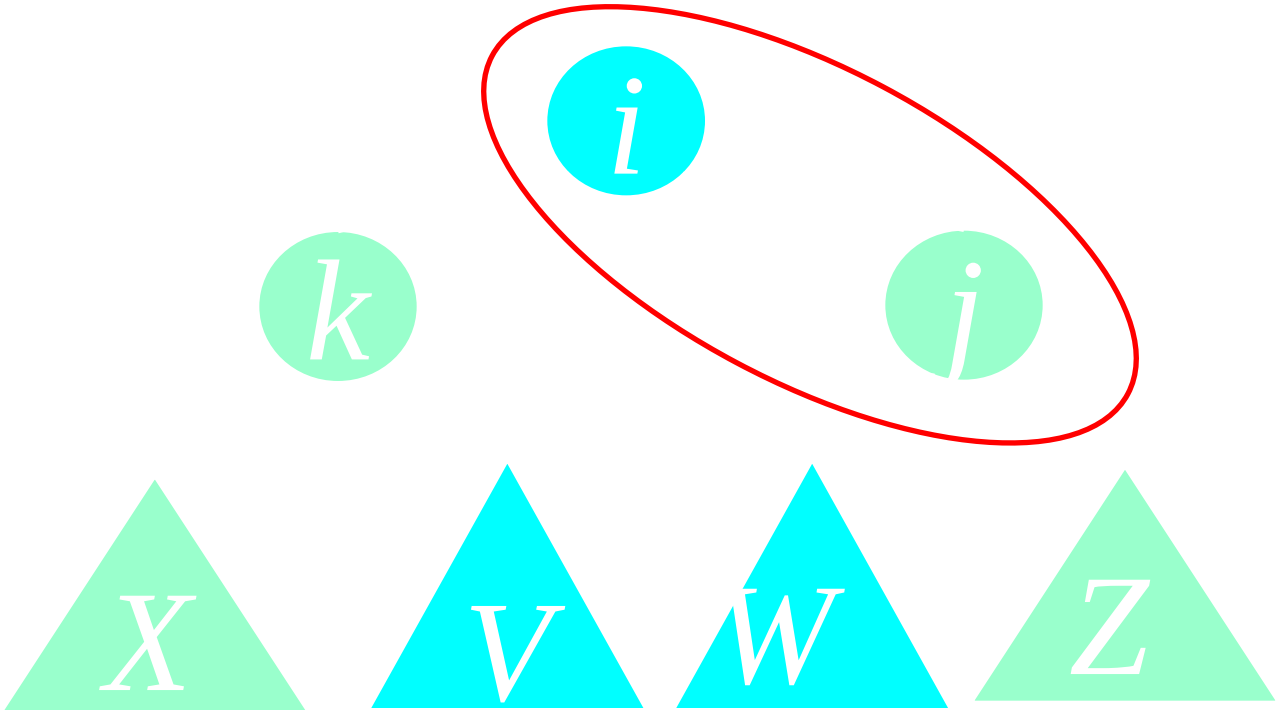
Double rotation : second rotation



Now do a right rotation

Double rotation : second rotation

right rotation complete



Implementation

balance

Single Rotation

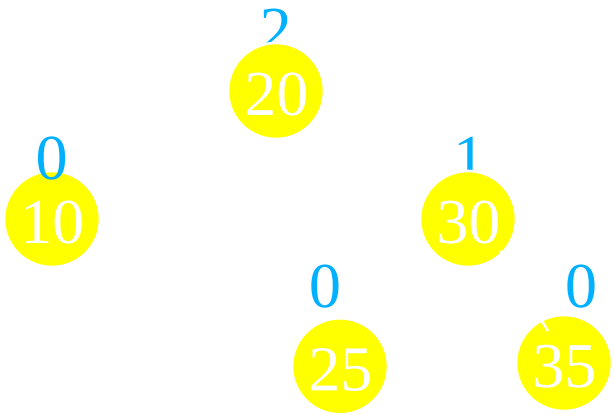
Double Rotation

- Implement Double Rotation in two lines.

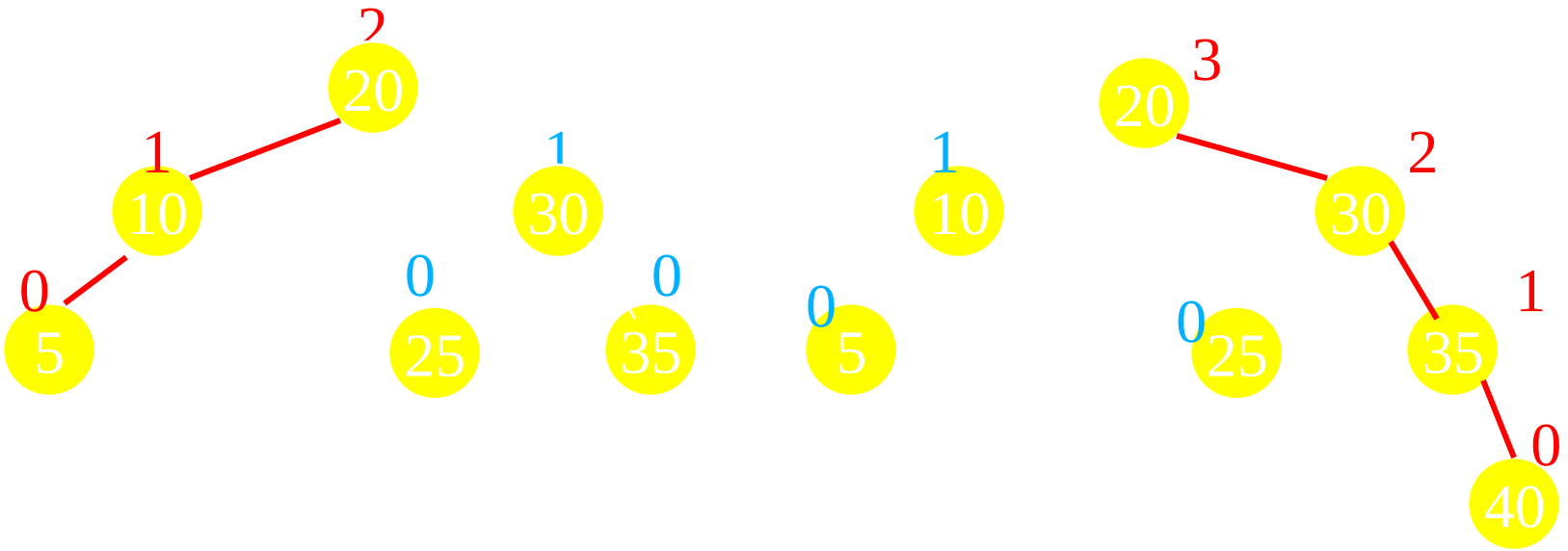
Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

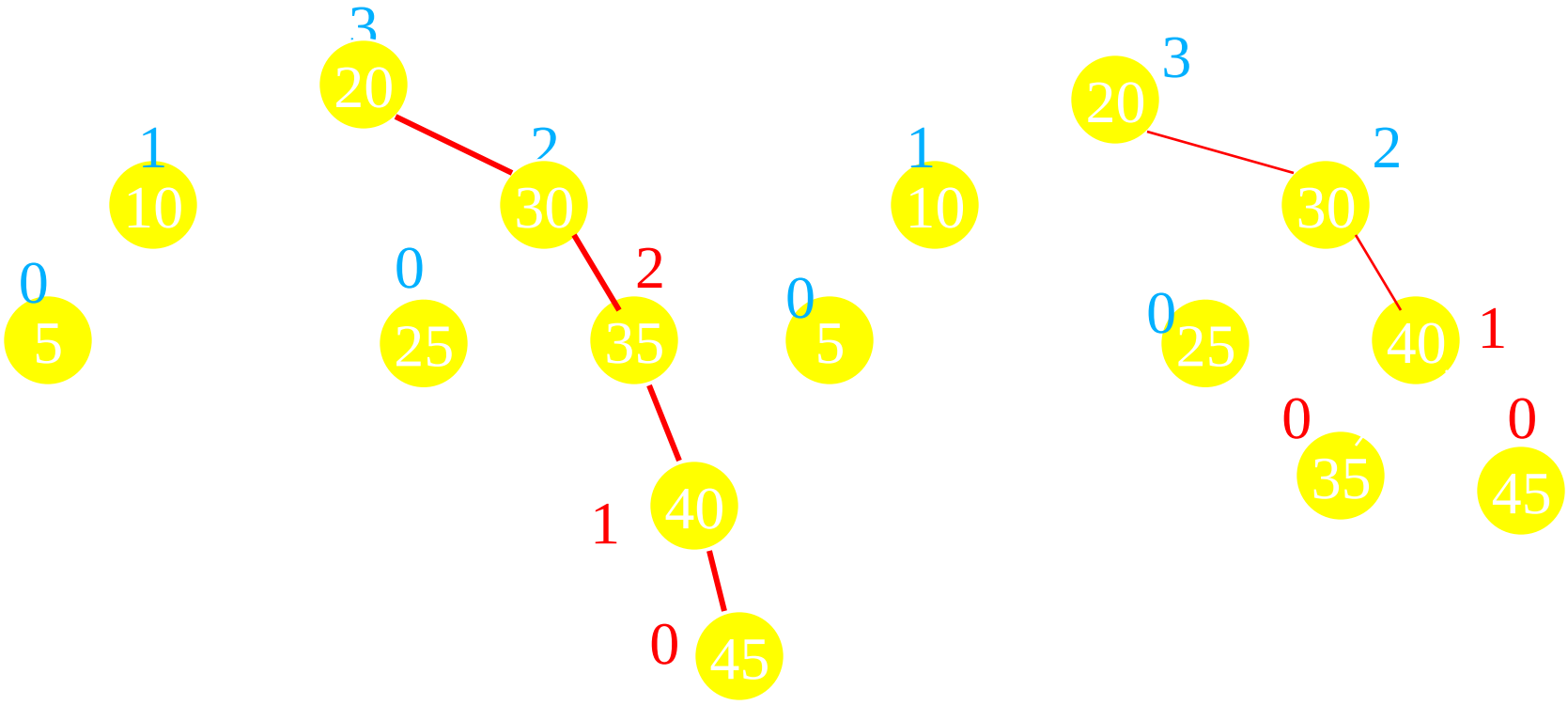
Example of Insertions in an AVL Tree



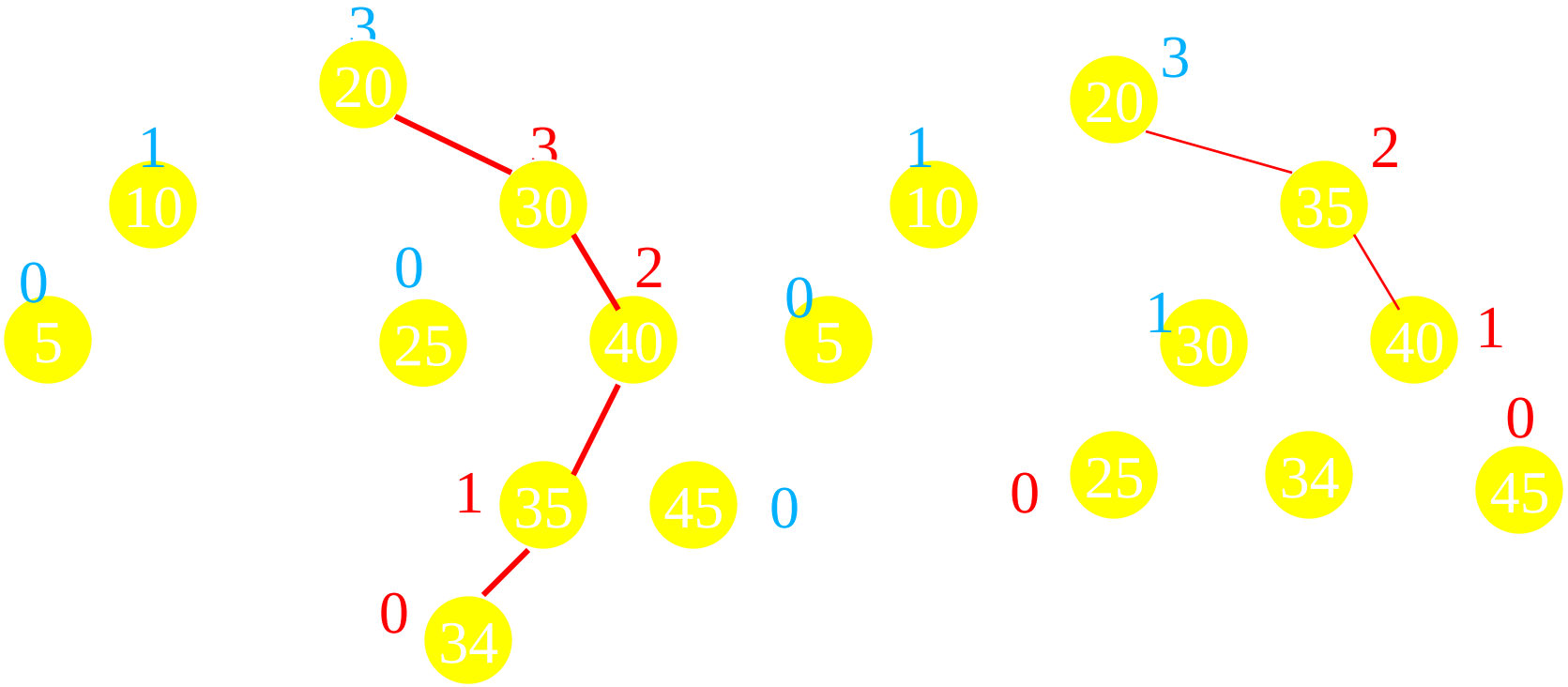
Example of Insertions in an AVL Tree



Single rotation (outside case)



Double rotation (inside case)



AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

Pros and Cons of AVL Trees

Arguments for AVL trees:

always balanced

Arguments against using AVL trees

Thank You !!!

L
P
U

CSE408
Bubble sort
Maximum & Minimum

Why Study Sorting Algorithms?

- There are a variety of situations that we can encounter
  - Do we have randomly ordered keys?
  - Are all keys distinct?
  - How large is the set of keys to be ordered?
  - Need guaranteed performance?

- Various algorithms are better suited to some of these situations

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Bubble Sort

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order

8 4 6 9 2 3 1

- Easier to implement, but slower than Insertion sort

Example

8 4 6 9 2 3 1

8 4 6 9 2 1 3

8 4 6 9 1 2 3

8 4 6 1 9 2 3

8 4 1 6 9 2 3

8 1 4 6 9 2 3

1 8 4 6 9 2 3

1 8 4 6 9 2 3

1 2 8 4 6 9 3

1 2 3 8 4 6 9

1 2 3 4 8 6 9

1 2 3 4 6 8 9

1 2 3 4 6 8 9

Bubble Sort

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

do for j ← length[A] **downto** i + 1

do if A[j] < A[j -1]

then exchange A[j] ↔ A[j-1]

8 4 6 9 2 3 1

Bubble-Sort Running Time

Alg.:

Comparisons: $\approx n^2/2$ **do if** $A[j] <$

Exchanges: $\approx n^2/2$ **then** exchange $A[j] \leftrightarrow A[j-1]$

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

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

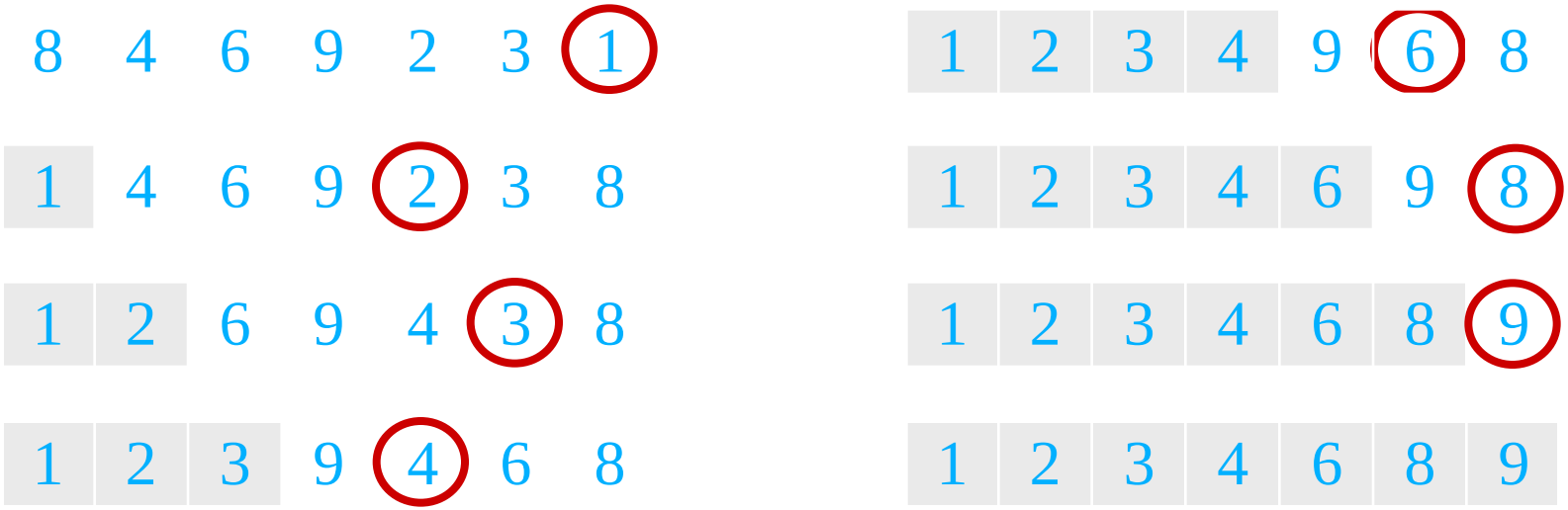
$$\text{where } \sum_{i=1}^{n}(n-i)=\sum_{i=1}^{n}n-\sum_{i=1}^{n}i=n^2-\frac{n(n+1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

Thus, T(n) = $\Theta(n^2)$

Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Example


8 4 6 9 2 3 1
1 4 6 9 2 3 8
1 2 6 9 4 3 8
1 2 3 9 4 6 8
1 2 3 4 9 6 8
1 2 3 4 6 9 8
1 2 3 4 6 8 9
1 2 3 4 6 8 9

Selection Sort

Alg.: SELECTION-SORT*(A)*

```
n ← length[A]
for j ← 1 to n - 1
      do smallest ← j
          for i ← j + 1 to n
                do if A[i] < A[smallest]
                      then smallest ← i
          exchange A[j] ↔ A[smallest]
```

8 4 6 9 2 3 1

Analysis of Selection Sort

Alg.: SELECTION-SORT*(A)*

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
    exchange A[j] ↔ A[smallest]
```

≈n²/2 comparisons

≈n exchanges

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

$c_5\sum_{j=1}^{n-1}(n-j)$

$\sum_{j=1}^{n-1}(n-j)$

c_7 n-1

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

MINIMUM AND MAXIMUM

```
MINIMUM(A)
1   min = A[1]
2   for i = 2 to A.length
3       if min > A[i]
4           min = A[i]
5   return min
```

CSE408
Count, Radix & Bucket Sort

How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort:
- Heap Sort, Merge sort:
- Quicksort:
- What is common to all these algorithms?
  - Make comparisons between input elements

$a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$

Lower-Bound for Sorting

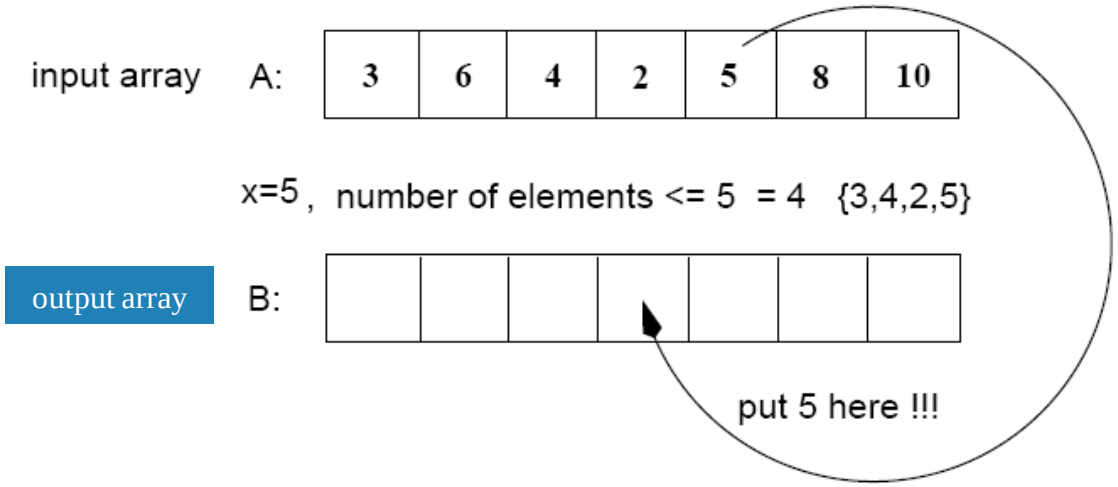
- **Theorem:** To sort n elements, comparison sorts must make $\Omega(n \lg n)$ comparisons in the worst case.

Can we do better?

- Linear sorting algorithms
  - Counting Sort
  - Radix Sort
  - Bucket sort
- Make certain assumptions about the data
- Linear sorts are NOT “comparison sorts”

Counting Sort

- Assumptions:
  - n integers which are in the range $[0 \ldots r]$
  - r is in the order of n, that is, $r=O(n)$
- Idea:
  - For each element x, find the number of elements $\leq x$
  - Place x into its correct position in the output array



Step 1

Find the number of times $A[i]$ appears in A

(i.e., frequencies)

input array A:

3	6	4	1	3	4	1	4

Allocate $C[1..r]$

allocate C

1	2	3	4	5	6
0	0	0	0	0	0

For $1 \le i \le n, ++C[A[i]];$

i=1, A[1]=3

1	2	3	4	5	6
0	0	1	0	0	0

C[A[1]]=C[3]=1

i=2, A[2]=6

1	2	3	4	5	6
0	0	1	0	0	1

C[A[2]]=C[6]=1

i=3, A[3]=4

1	2	3	4	5	6
0	0	1	1	0	1

C[A[3]]=C[4]=1

.
.
.

i=8, A[8]=4

1	2	3	4	5	6
2	0	2	3	0	1

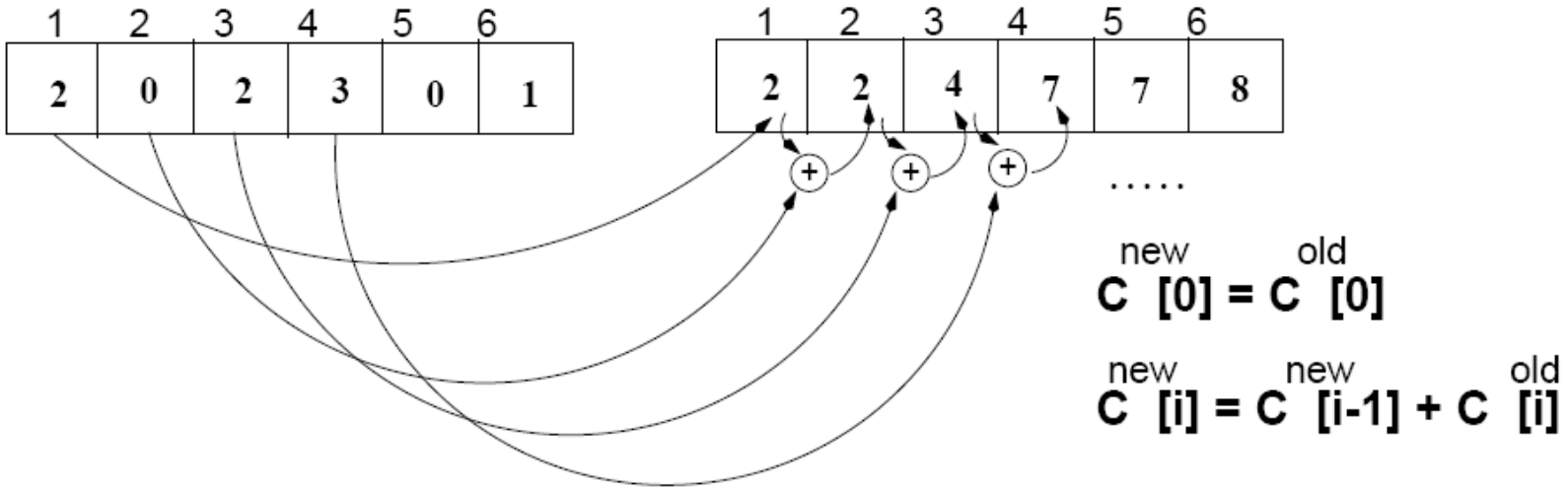
C[A[8]]=C[4]=3

C[i] = number of times element i appears in A

Step 2

Find the number of elements $\leq A[i]$,

(frequencies) (cumulative sums)



$C^{new}[0] = C^{old}[0]$

$C^{new}[i] = C^{new}[i-1] + C^{old}[i]$

C[i] = # elements <= i

Algorithm

- Start from the last element of A
- Place A[i] at its correct place in the output array
- Decrease C[A[i]] by one

2 5 3 0 2 3 0 3

2 2 4 7 7 8

Example

2 5 3 0 2 3 0 3

2 2 4 7 7 8

Example (cont.)

2 5 3 0 2 3 0 3

0 0 2 3 3

0 2 3 5 7 8

0 0 2 3 3 3 5

0 2 3 4 7 7

0 0 2 3 3 3

0 2 3 4 7 8

0 0 2 2 3 3 3 5

COUNTING-SORT

Alg.: **COUNTING-SORT(A, B, n, k)**

```
1.      for i ← 0 to r
2.          do C[ i ] ← 0
3.      for j ← 1 to n
4.          do C[A[ j ]] ← C[A[ j ]] + 1
5.      C[i] contains the number of elements equal to i
6.      for i ← 1 to r
7.          do C[ i ] ← C[ i ] + C[i -1]
8.       C[i] contains the number of elements ≤ i
9.      for j ← n downto 1
10.         do B[C[A[ j ]]] ← A[ j ]
11.             C[A[ j ]] ← C[A[ j ]] - 1
```

Analysis of Counting Sort

Alg.: **COUNTING-SORT(A, B, n, k)**

```
1.      for i ← 0 to r
2.          do C[ i ] ← 0
3.      for j ← 1 to n
4.          do C[A[ j ]] ← C[A[ j ]] + 1
5.      C[i] contains the number of elements equal to i
6.      for i ← 1 to r
7.          do C[ i ] ← C[ i ] + C[i -1]
8.      C[i] contains the number of elements ≤ i
9.      for j ← n downto 1
10.         do B[C[A[ j ]]] ← A[ j ]
11.             C[A[ j ]] ← C[A[ j ]] - 1
```

Analysis of Counting Sort

- Overall time: $O(n + r)$

- In practice we use COUNTING sort when $r = O(n)$

  $\Rightarrow$ running time is $O(n)$

Radix Sort

- Represents keys as d-digit numbers in some base-k

  $key = x_1 x_2 ... x_d$ where $0 \le x_i \le k-1$

- Example: key=15

  $key_{10} = 15$, $d=2$, $k=10$ where $0 \le x_i \le 9$

  $key_2 = 1111$, $d=4$, $k=2$ where $0 \le x_i \le 1$

- Assumptions

  d=O(1) and k =O(n)

- Sorting looks at one column at a time
  - For a d digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit, until all digits have been sorted
  - Requires only d passes through the list

326
453
608
835
751
435
704
690

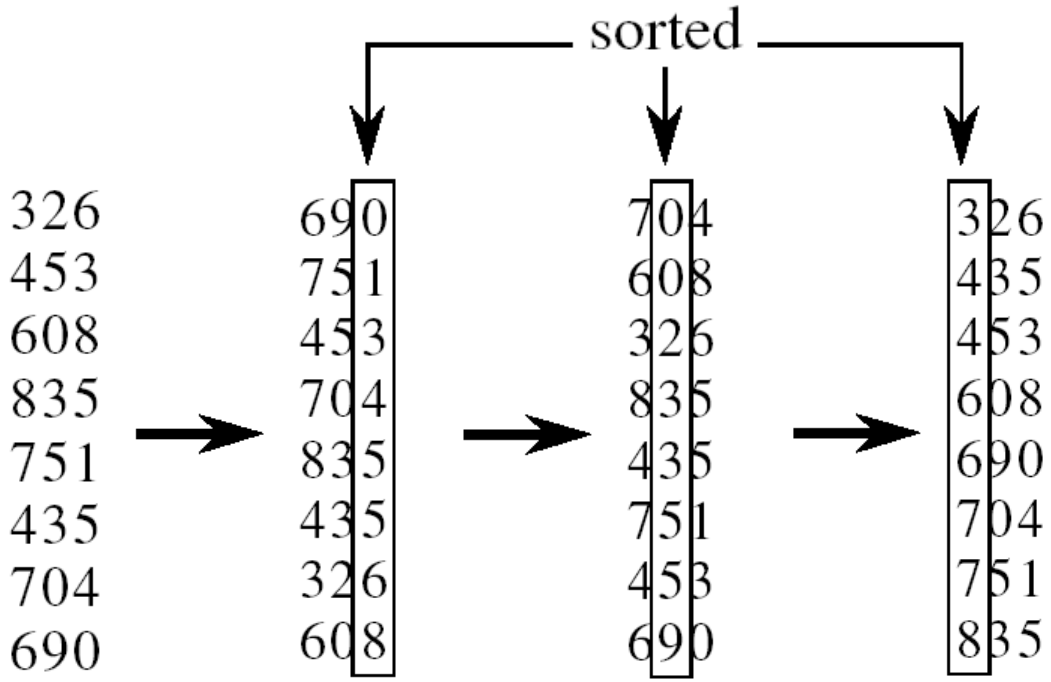
RADIX-SORT

Alg.: RADIX-SORT(*A*, d)

for i ← 1 to d

do use a stable sort to sort array *A* on digit i

(stable sort: preserves order of identical elements)



Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $O(d(n+k))$
  - One pass of sorting per digit takes $O(n+k)$ assuming that we use **counting sort**
  - There are d passes (for each digit)

Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $O(d(n+k))$
  - Assuming d=O(1) and k=O(n), running time is O(n)

Bucket Sort

- Assumption:
  - the input is generated by a random process that distributes elements uniformly over [0, 1)
- Idea:
  - Divide [0, 1) into k equal-sized buckets ($k=\Theta(n)$)
  - Distribute the n input values into the buckets
  - Sort each bucket (e.g., using quicksort)
  - Go through the buckets in order, listing elements in each one

- **Input:** $A[1 \ldots n]$, where $0 \le A[i] < 1$ for all i

- **Output:** elements $A[i]$ sorted

Example - Bucket Sort

.78

.17

.39

.26

.72

.94

.21

.12

.23

.68

Example - Bucket Sort

Example - Bucket Sort

Analysis of Bucket Sort

Alg.: BUCKET-SORT(A, n)

for i ← 1 **to** n

do insert A[i] into list B[⌊nA[i]⌋]

for i ← 0 **to** k - 1

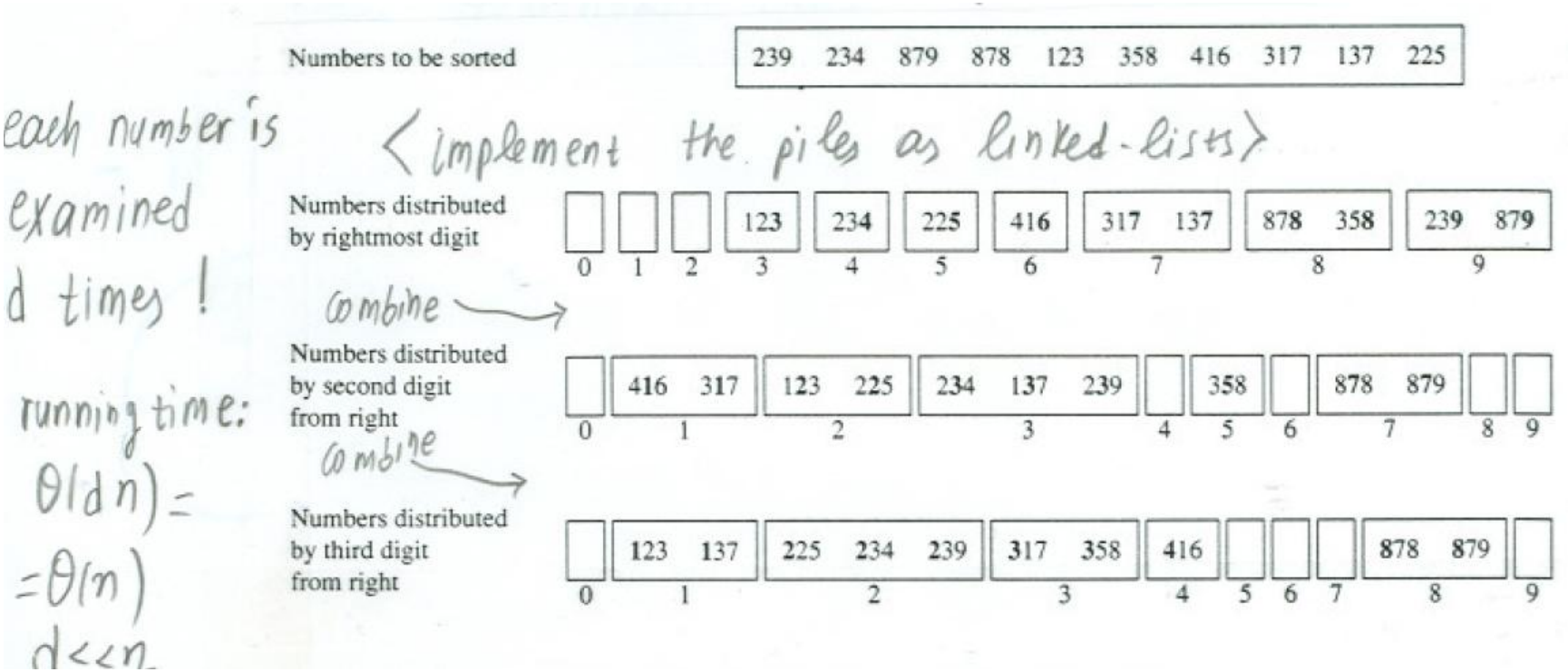
do sort list B[i] with quicksort sort

concatenate lists B[0], B[1], . . . , B[n -1]

together in order

return the concatenated lists

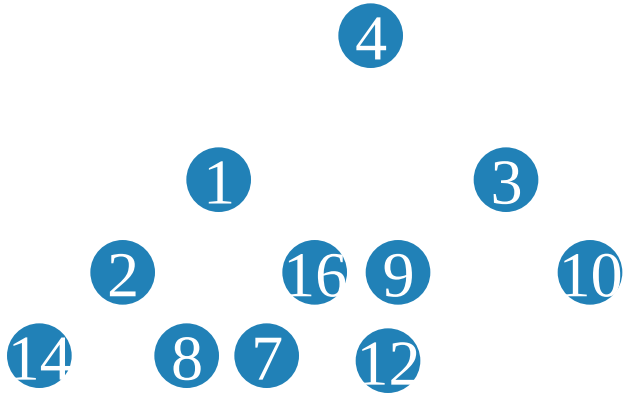
Radix Sort as a Bucket Sort



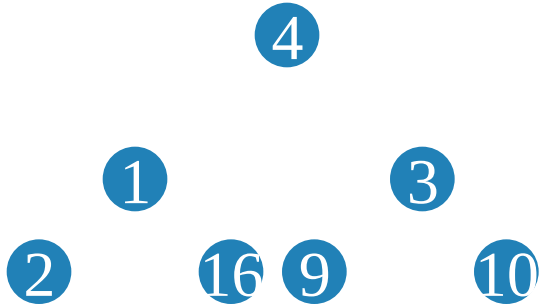
CSE408
Heap & Heap sort,Hashing

Special Types of Trees

- *Def:* Full binary tree =



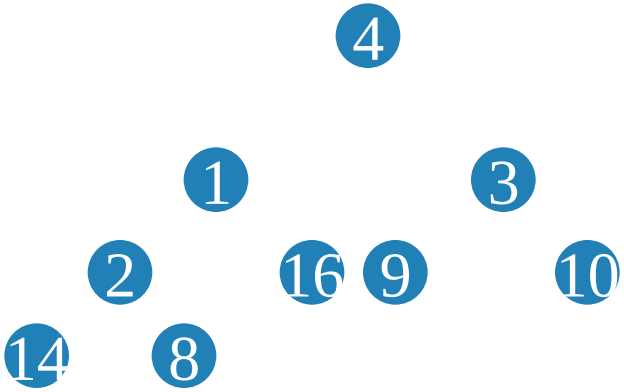
- *Def:* Complete binary tree =



Definitions

- **Height** of a node =

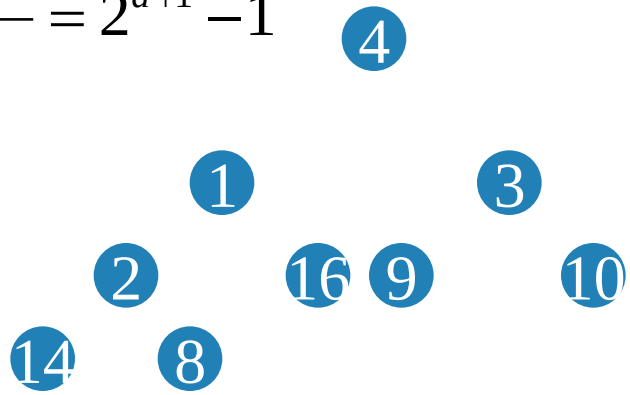
- **Level** of a node =
- **Height** of tree =



Useful Properties

- There are **at most** 2^l nodes at level (or depth) l of a binary tree
- A binary tree with height d has **at most** $2^{d+1} - 1$ nodes
- A binary tree with n nodes has height **at least** $\lfloor lgn \rfloor$

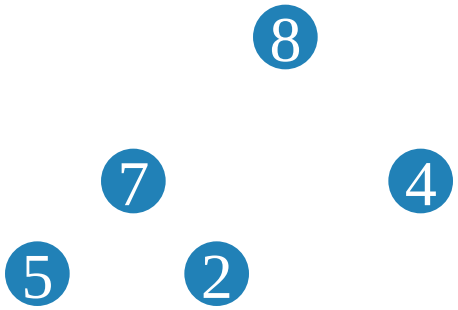
$$n \leq \sum_{l=0}^{d} 2^l = \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1$$



The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

$$Parent(x) \geq x$$

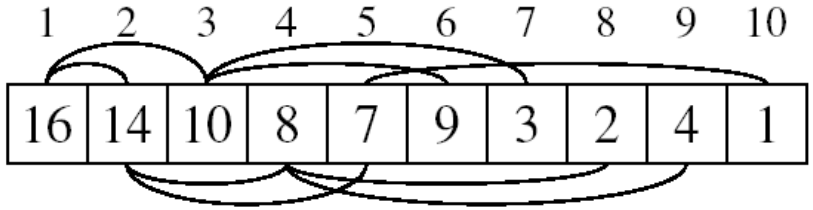


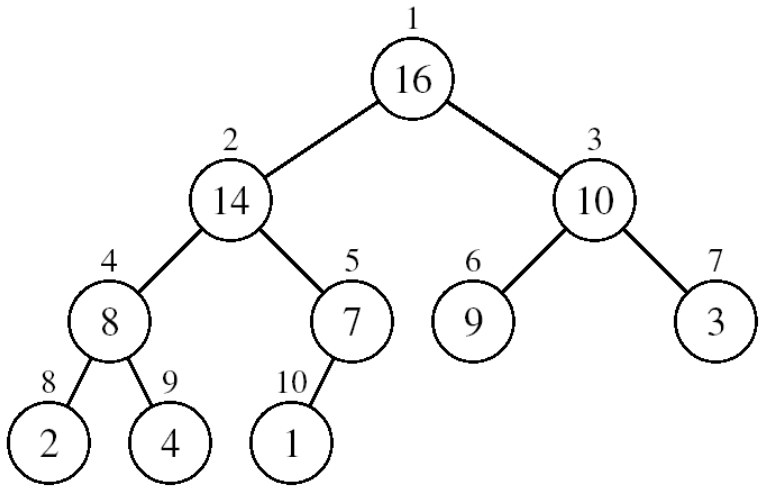
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is *A*[1]
  - Left child of *A*[i] = *A*[2i]
  - Right child of *A*[i] = *A*[2i + 1]
  - Parent of *A*[i] = *A*[$\lfloor i/2 \rfloor$]
  - Heapsize[A] ≤ length[A]
- The elements in the subarray *A*[($\lfloor n/2 \rfloor$+1) .. n] are leaves





Heap Types

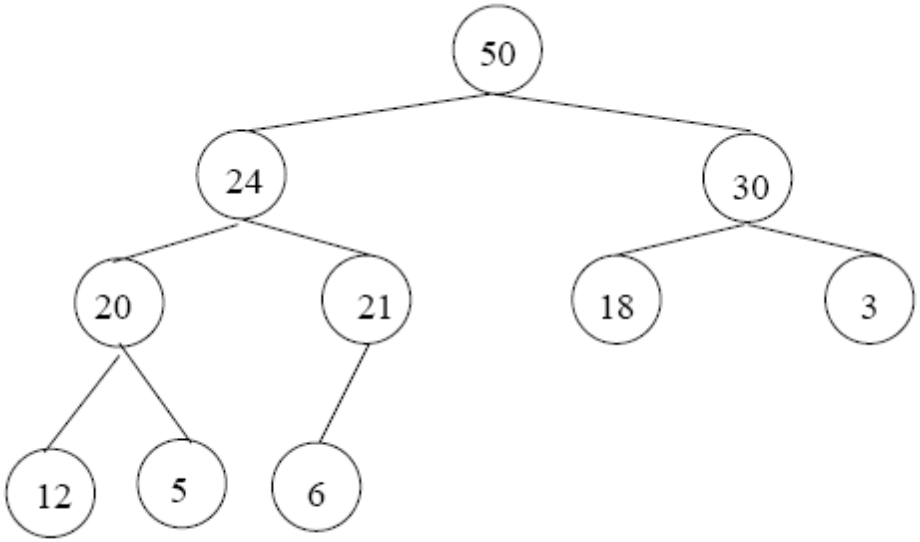
- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

$$A[PARENT(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

Adding/Deleting Nodes

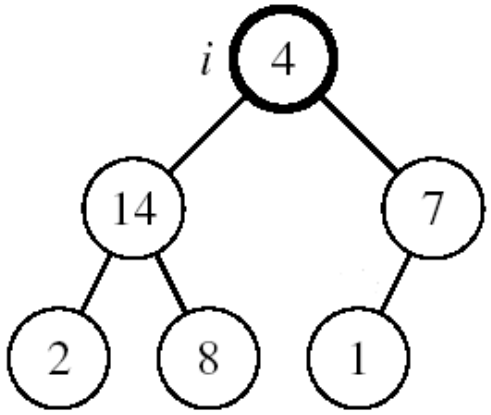


Operations on Heaps

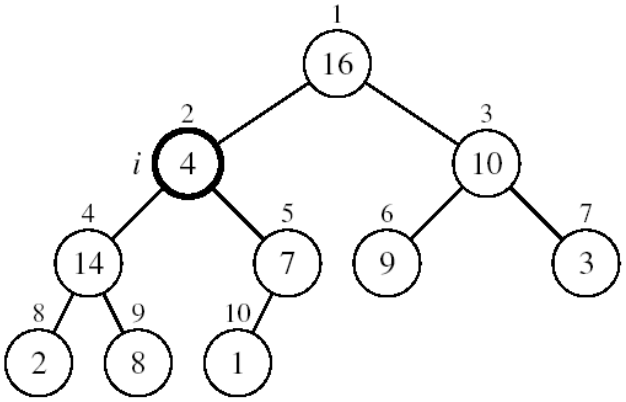
-
  - MAX-HEAPIFY
-
  - BUILD-MAX-HEAP
-
  - HEAPSORT
-

Maintaining the Heap Property

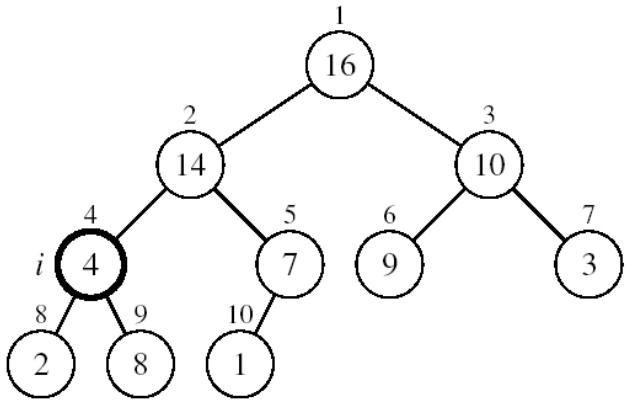
- **Suppose a node is smaller than a child**
  - **Left and Right subtrees of i are max-heaps**
- **To eliminate the violation:**
  - **Exchange with larger child**
  - **Move down the tree**
  - **Continue until node is not smaller than children**



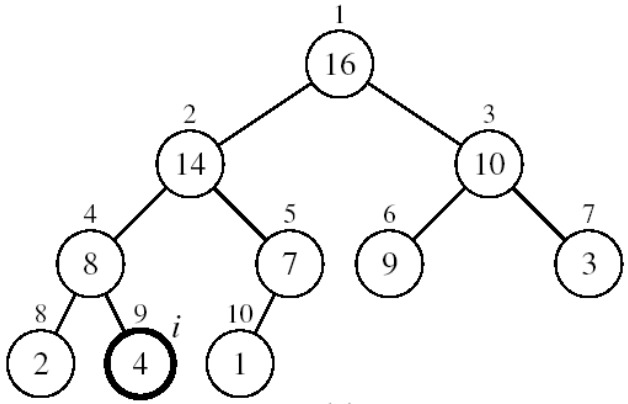
Example



A[2] ↔ A[4]

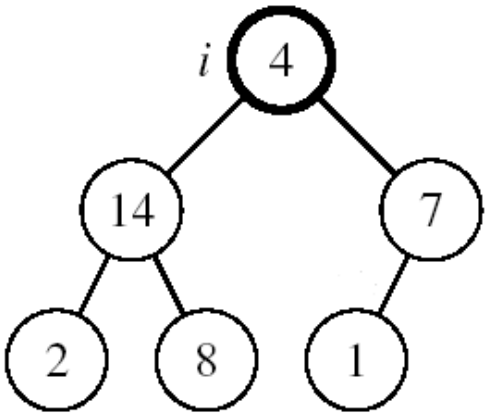


A[4] ↔ A[9]



Maintaining the Heap Property

- **Assumptions:**
  - **Left and Right subtrees of i are max-heaps**
  - **A[i] may be smaller than its children**



Alg: MAX-HEAPIFY(A, i, n)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and A[l] > A[i]
4. **then** largest ←l
5. **else** largest ←i
6. **if** r ≤ n and A[r] > A[largest]
7. **then** largest ←r
8. **if** largest ≠ i
9. **then** exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest, n)

MAX-HEAPIFY Running Time

- Intuitively:

  It traces a path from the root to a leaf (longest path length: h)
  At each level, it makes exactly 2 comparisons
  Total number of comparisons is 2h
  Running time is O(h) or $O(lgn)$

- Running time of MAX-HEAPIFY is $O(lgn)$
- Can be written in terms of the height of the heap, as being

  $O(h)$

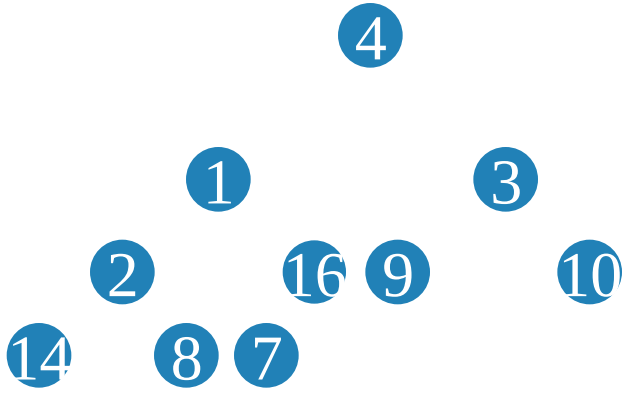
  - Since the height of the heap is $\lfloor lgn \rfloor$

Building a Heap

- Convert an array A[1 … n] into a max-heap (n = length[A])
- The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves
- Apply MAX-HEAPIFY on elements between 1 and ⌊n/2⌋

Alg: **BUILD-MAX-HEAP(A)**

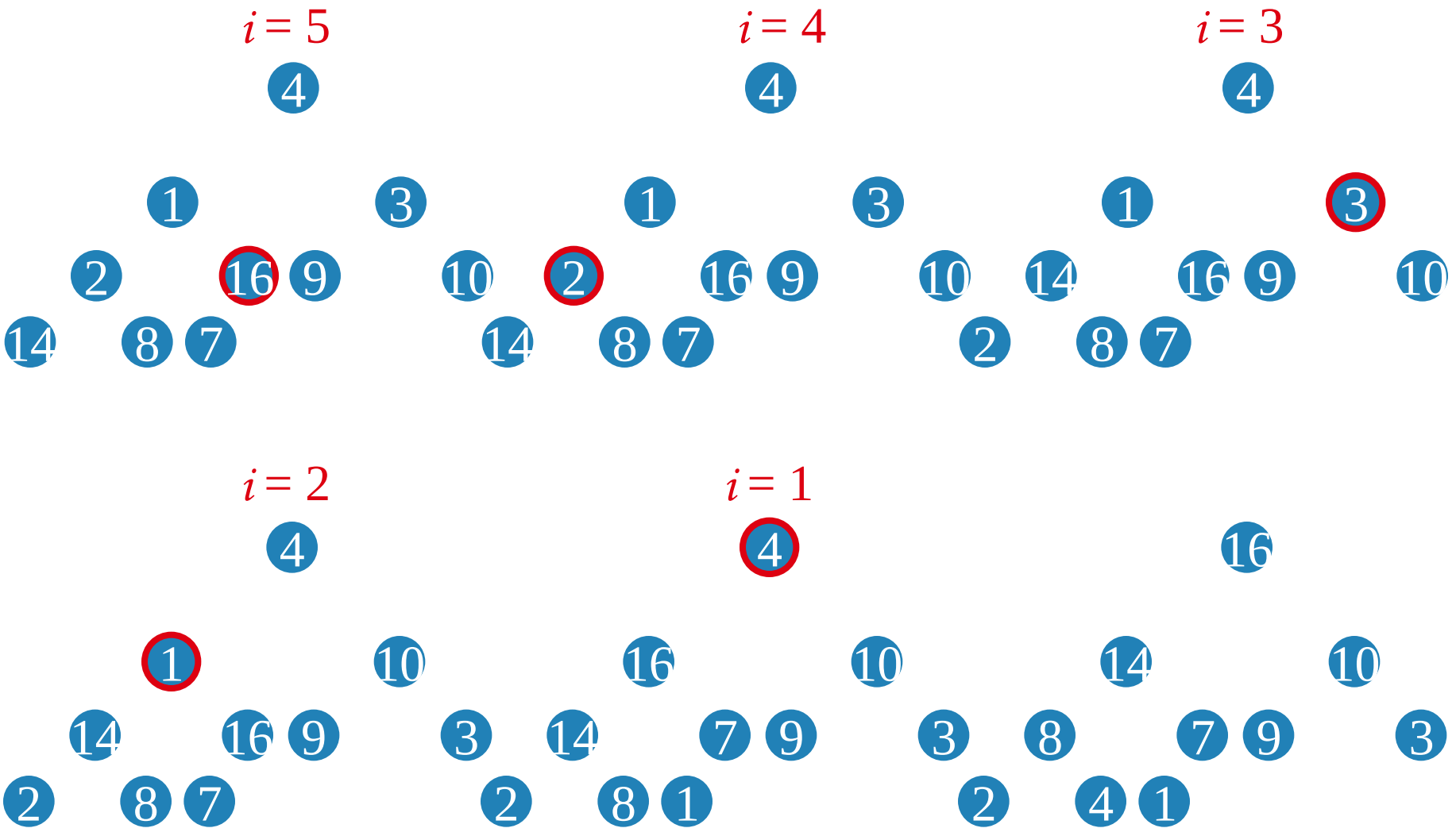
1. n = length[A]
2. for i ← ⌊n/2⌋ downto 1
3. do MAX-HEAPIFY(A, i, n)



4 1 3 2 16 9 10 14 8 7

Example: A

4 1 3 2 16 9 10 14 8 7



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

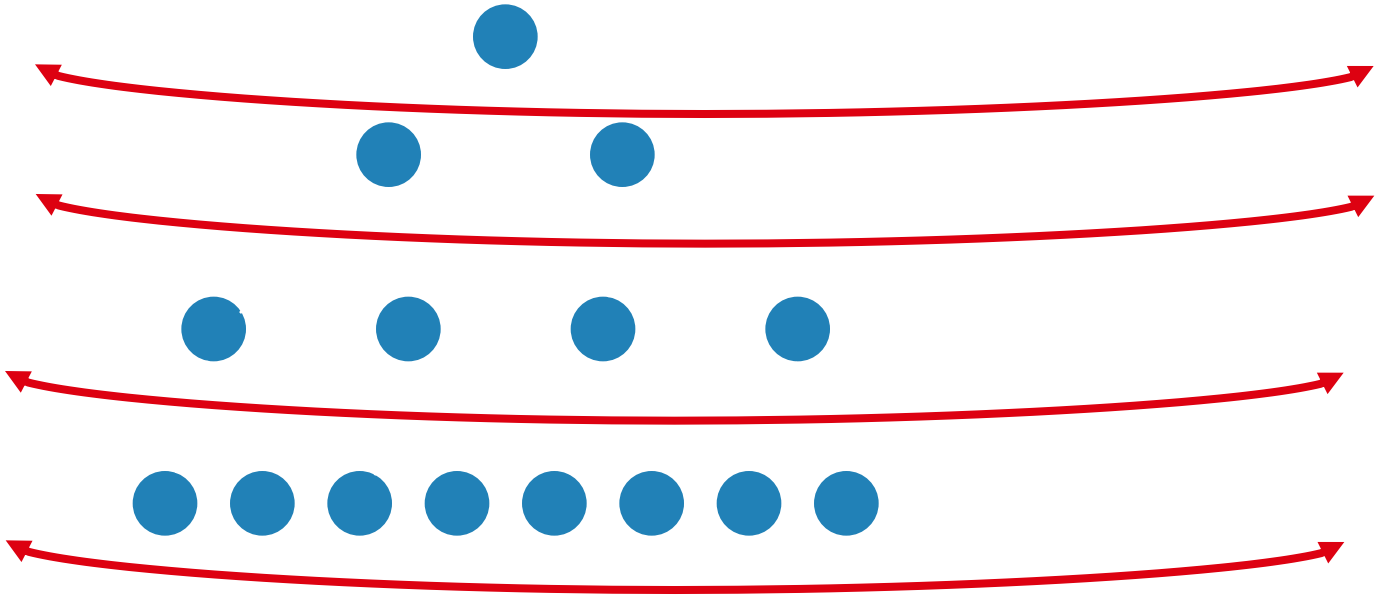
$\Rightarrow$ Running time: $O(n \lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- **HEAPIFY takes $O(h) \Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree**

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$



Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$

$$= \sum_{i=0}^{h} 2^i (h-i)$$

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} \cdot 2^h \qquad \frac{1}{2^{-i}}$$

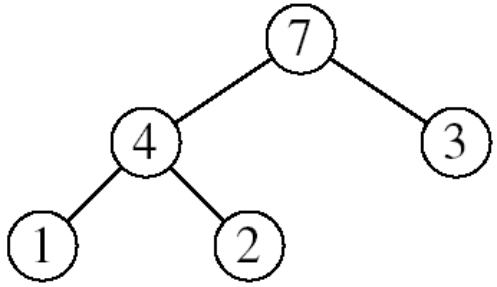
$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

$$= O(n)$$

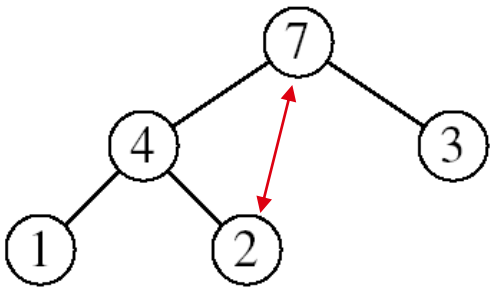
Heapsort

- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array

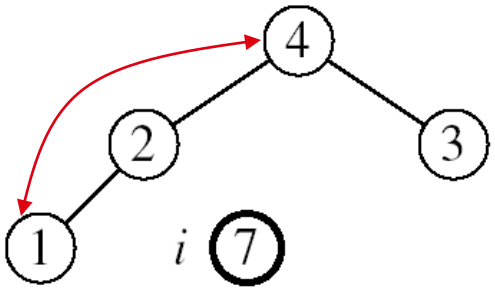


  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

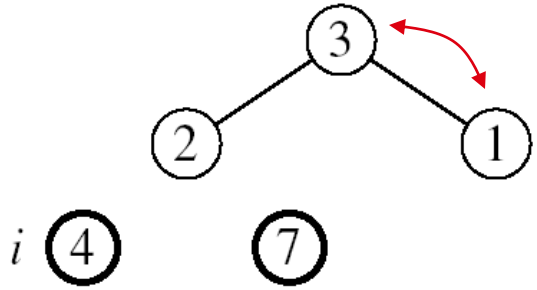
Example: A=[7, 4, 3, 1, 2]



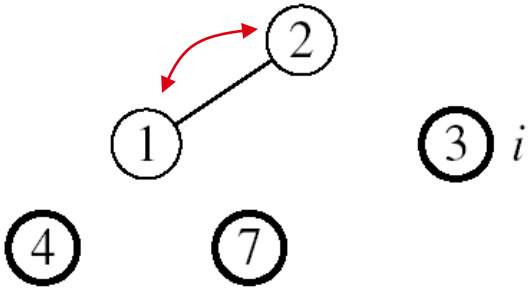
MAX-HEAPIFY(A, 1, 4)

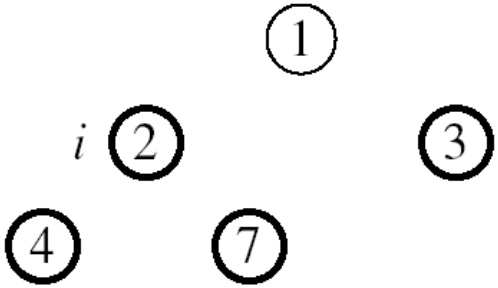


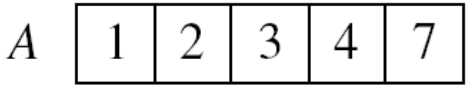
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)







MAX-HEAPIFY(A, 1, 1)

Alg: HEAPSORT*(A)*

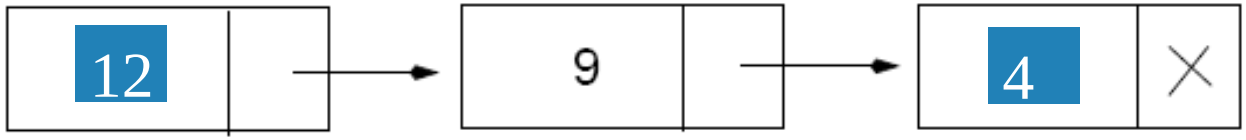
1. BUILD-MAX-HEAP(*A*)
2. **for** i ← length[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[i]
4. MAX-HEAPIFY(*A*, 1, i - 1)

- Running time: O(nlgn) --- Can be shown to be Θ(nlgn)

Priority Queues

Properties

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first



Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): inserts element x into set S
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - MAXIMUM(S): returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of element x's key to k (Assume $k \geq x$'s current key value)

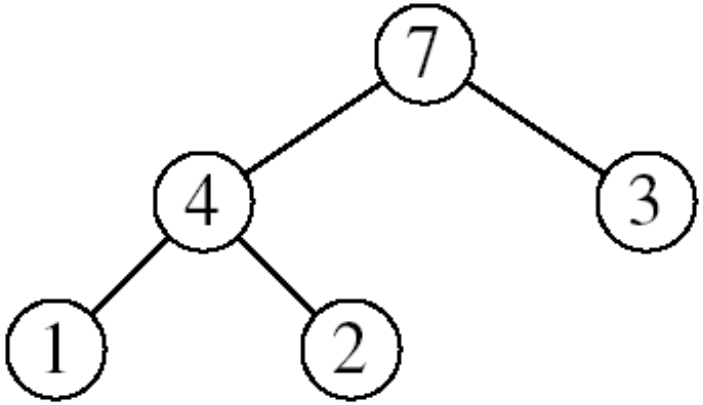
HEAP-MAXIMUM

Goal:

- **Return the largest element of the heap**

Alg: **HEAP-MAXIMUM(*A*)**

1. **return *A*[1]**



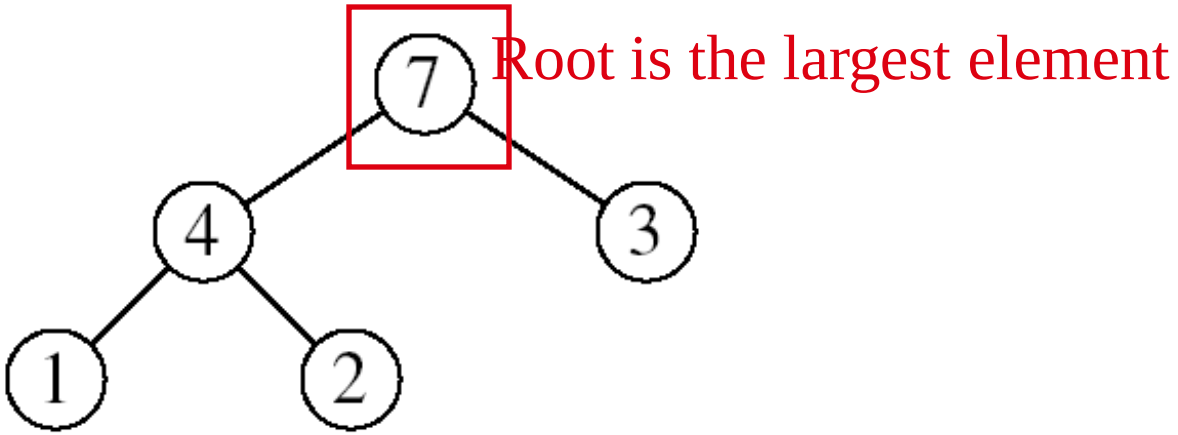
HEAP-EXTRACT-MAX

Goal:

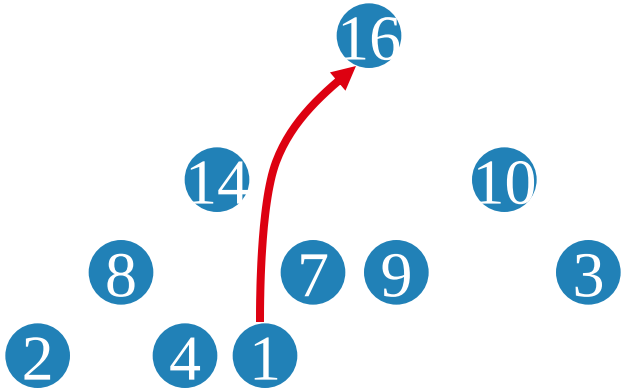
- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Idea:

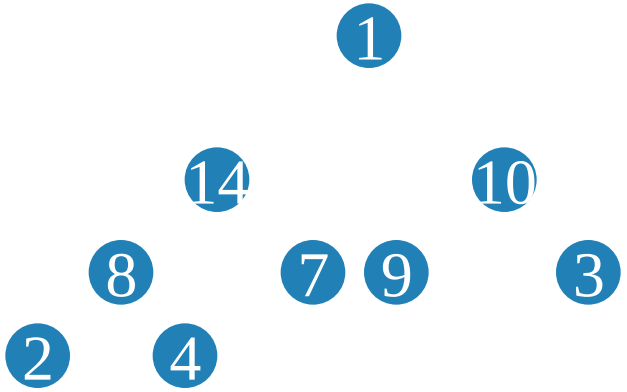
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1



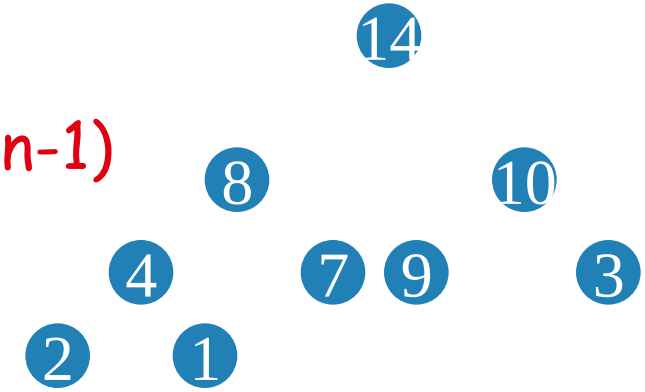
Example: HEAP-EXTRACT-MAX



max = 16



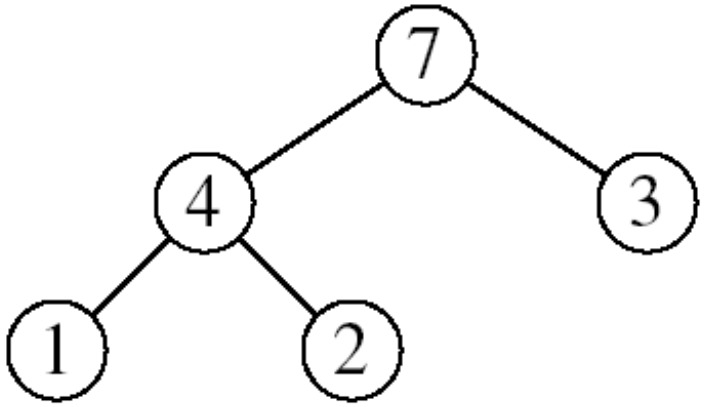
Call MAX-HEAPIFY(*A*, 1, n-1)



HEAP-EXTRACT-MAX

Alg: **HEAP-EXTRACT-MAX(A, n)**

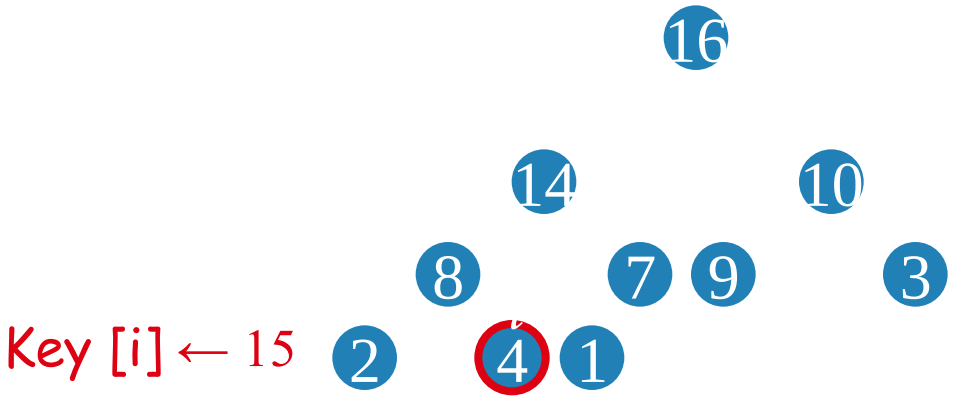
1. **if** n < 1
2. **then error** “heap underflow”
3. max ← A[1]
4. A[1] ← A[n]
5. **MAX-HEAPIFY**(A, 1, n-1)
6. **return** max



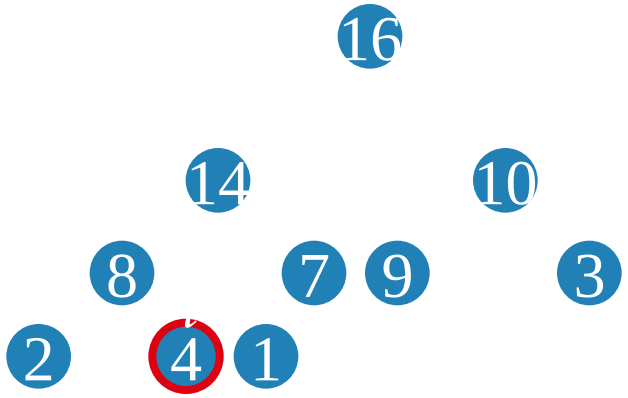
remakes heap

HEAP-INCREASE-KEY

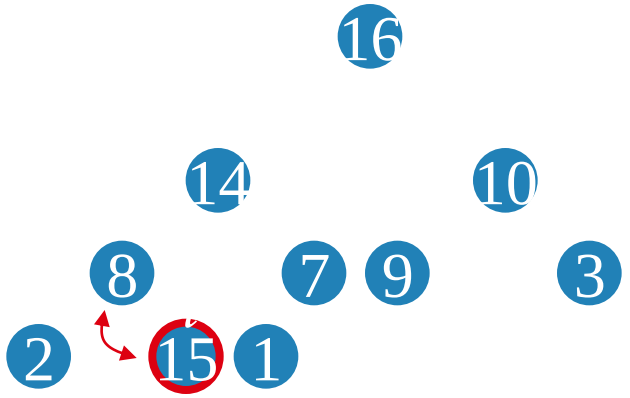
- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

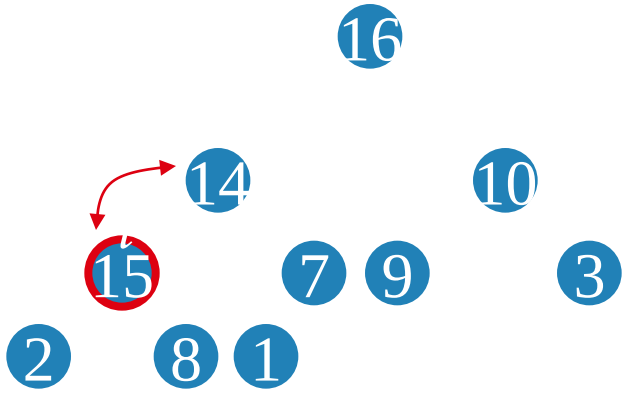


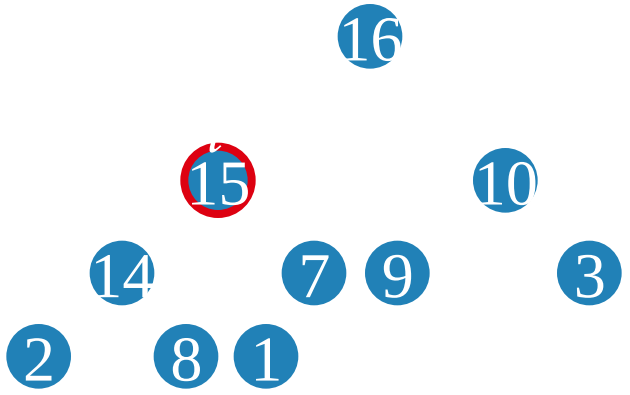
Example: HEAP-INCREASE-KEY



$Key[i] \leftarrow 15$





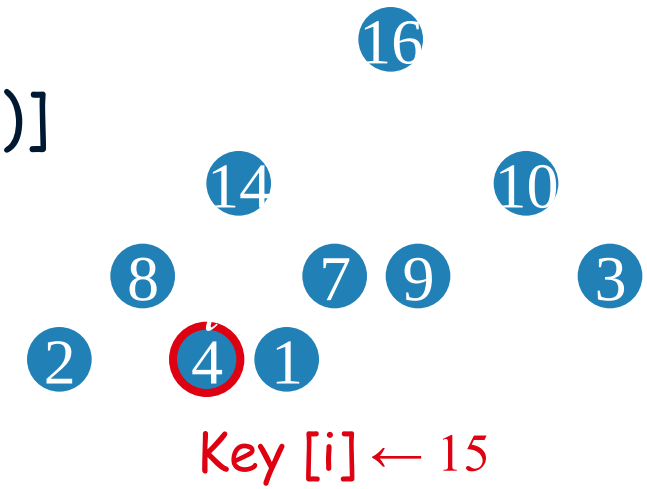


HEAP-INCREASE-KEY

Alg: HEAP-INCREASE-KEY(A, i, key)

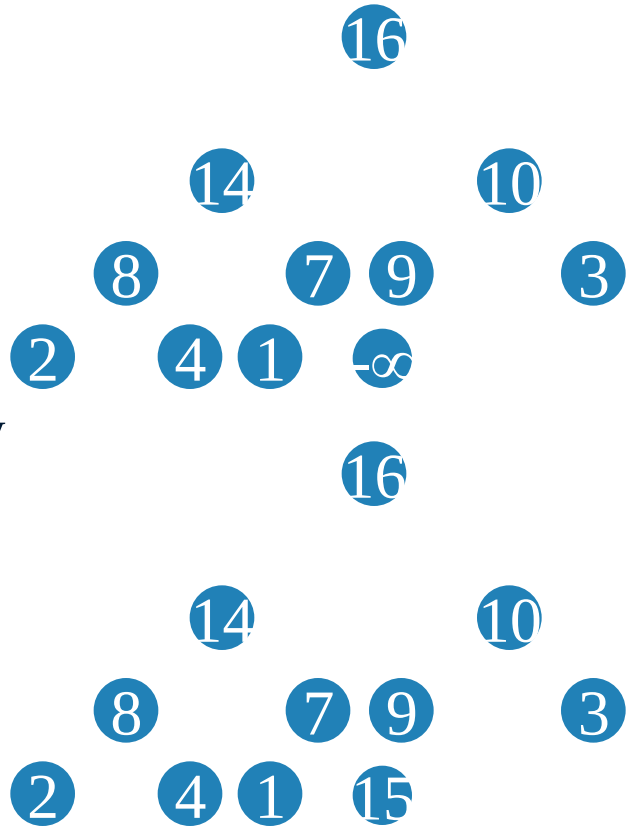
1. **if** key < A[i]
2. **then error** "new key is smaller than current key"
3. A[i] ← key
4. **while** i > 1 and A[PARENT(i)] < A[i]
5. **do** exchange A[i] ↔ A[PARENT(i)]
6. i ← PARENT(i)

- Running time: O(lgn)

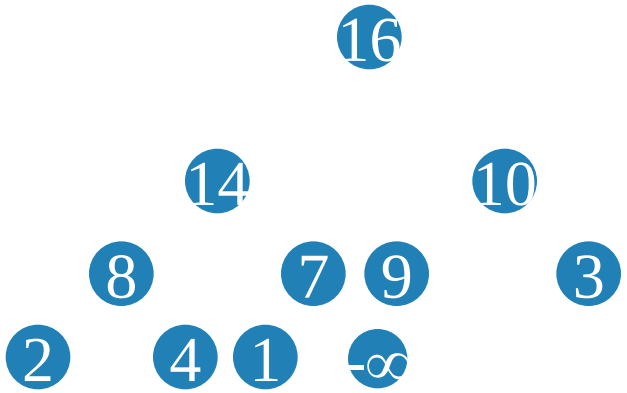


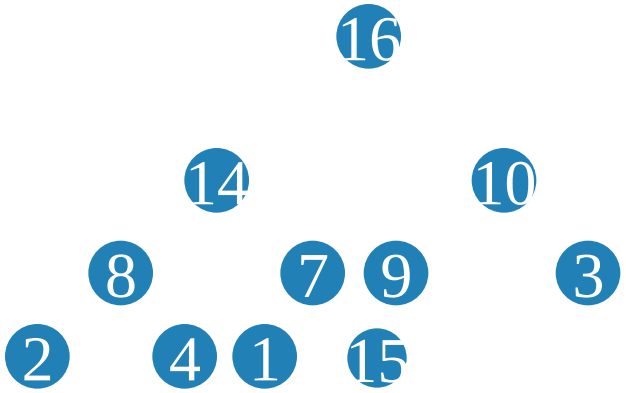
MAX-HEAP-INSERT

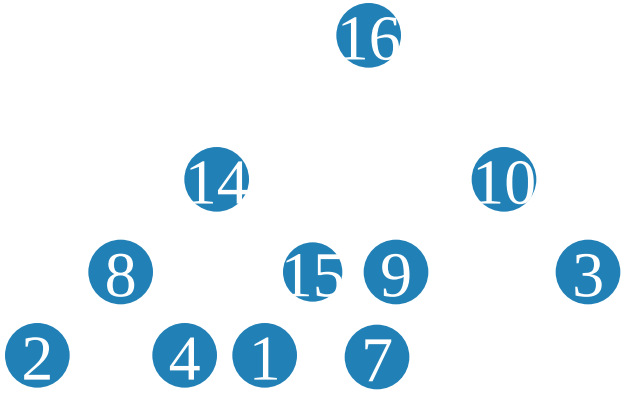
- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property

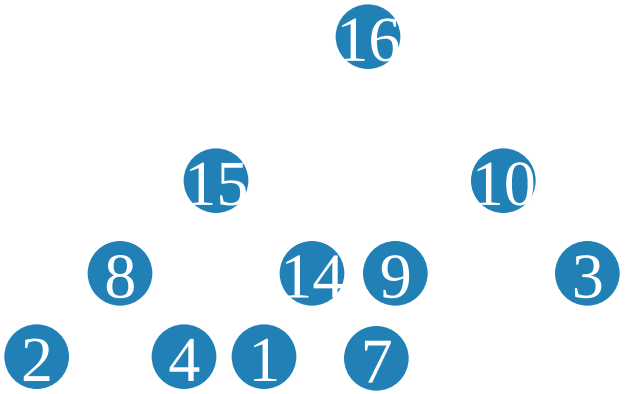


Example: MAX-HEAP-INSERT





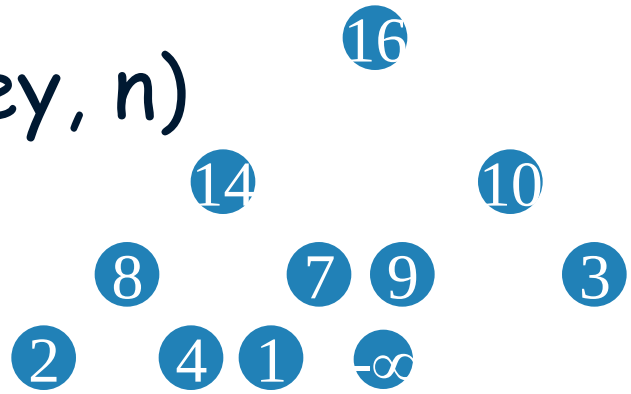




MAX-HEAP-INSERT

Alg: MAX-HEAP-INSERT(A, key, n)

1. heap-size[A] ← n + 1
2. A[n + 1] ← -∞
3. HEAP-INCREASE-KEY(A, n + 1, key)



Summary

- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n \lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Hash Functions

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Some methods

- **Truncation:**
  - e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Folding:**
  - e.g. 123|456|789: add them and take mod.
- **Key mod N:**
  - N is the size of the table, better if it is prime.
- **Squaring:**
  - Square the key and then truncate
- **Radix conversion:**
  - e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.

Hash Function 1

- Add up the ASCII values of all characters of the key.

Hash Function 2

- Examine only the first 3 characters of the key.

```
int hash (const string &key, int tableSize)
                                {
    return (key[0]+27 * key[1] + 729*key[2]) % tableSize;

                                }
```

Hash Function 3

$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^{i}$$

```
int hash (const string &key, int tableSize)
                        {
                    int hashVal = 0;

          for (int i = 0; i < key.length(); i++)
                hashVal = 37 * hashVal + key[i];

                    hashVal %=tableSize;
  if (hashVal < 0)    /* in case overflows occurs */
                    hashVal += tableSize;

                  return hashVal;
                        };
```

Hash function for strings:

Double Hashing

- A second hash function is used to drive the collision resolution.
  - $f(i) = i * hash_2(x)$
- We apply a second hash function to x and probe at a distance $hash_2(x)$, $2*hash_2(x)$, … and so on.
- The function $hash_2(x)$ must never evaluate to zero.
  - e.g. Let $hash_2(x) = x \bmod 9$ and try to insert 99 in the previous example.
- A function such as $hash_2(x) = R - (x \bmod R)$ with R a prime smaller than TableSize will work well.
  - e.g. try R = 7 for the previous example.(7 - x mode 7)

Hashing Applications

- Compilers use hash tables to implement the *symbol table* (a data structure to keep track of declared variables).
- Game programs use hash tables to keep track of positions it has encountered (*transposition table*)
- Online spelling checkers.

Summary

- Hash tables can be used to implement the insert and find operations in constant average time.
  - it depends on the load factor not on the number of items in the table.
- It is important to have a prime TableSize and a correct choice of load factor and hash function.
- For separate chaining the load factor should be close to 1.
- For open addressing load factor should not exceed 0.5 unless this is completely unavoidable.
  - Rehashing can be implemented to grow (or shrink) the table.