Winter 2016 COMP-250: Introduction to Computer Science

Lecture 26, April 14, 2016

REVIEW SESSION

COMP 250	001	Intro to Computer Science	Apr 28	2 pm	Crepeau	AAA	-	ZZZ	GYM	FIELD HOUSE	18-30

COMP 250	001	Intro to Computer Science	Apr 28	2 pm	Crepeau	AAA	-	ZZZ	GYM	FIELD HOUSE	18-30
COMP 250	001	Intro to Computer Science	Apr 28	2 pm	Crepeau	AAA	-	ZZZ	GYM	FIELD HOUSE	18-30
COMP 250	001	Intro to Computer Science	Apr 28	2 pm	Crepeau	AAA	-	ZZZ	GYM	FIELD HOUSE	18-30
COMP 250	001	Intro to Computer Science	Apr 28	2 pm	Crepeau	AAA	-	ZZZ	GYM	FIELD HOUSE	18-30
COMP 250	001	Intro to Computer Science	Apr 28	2 pm	Crepeau	AAA	-	ZZZ	GYM	FIELD HOUSE	18-30

- This is a multiple choices exam. For each question, only one answer can be provided.
- Answer the questions on the multiple choice page, using a LEAD PENCIL.
- You have 180 minutes to write the exam.
- This exam is worth 50% of your total mark.
- ALL DOCUMENTATION IS PERMITTED including books, notes and printed slides.
- No electronic devices are allowed.
- If you believe that none of choices provided for a given question are correct, provide the answer that is the closest to being correct.
- This exam contains 40 questions on 16 pages.
- This examination is printed on both sides of the paper.
- THIS EXAMINATION PAPER MUST BE RETURNED.
- The *Examination Security Monitor Program* detects pairs of students with unusually similar answer patterns on multiple-choice exams. Data generated by this program can be used as admissible evidence, either to initiate or corroborate an investigation or a charge of cheating under Section 16 of the *Code of Student Conduct and Disciplinary Procedures*.

- This is a multiple choices exam. For each question, only one answer can be provided.
- Answer the questions on the multiple choice page, using a LEAD PENCIL.
- You have 180 minutes to write the exam.
- This exam is worth 50% of your total mark.
- ALL DOCUMENTATION IS PERMITTED including books, notes and printed slides.
- No electronic devices are allowed.
- If you believe that none of choices provided for a given question are correct, provide the answer that is the closest to being correct.
- This exam contains 40 questions on 16 pages.
- This examination is printed on both sides of the paper.
- THIS EXAMINATION PAPER MUST BE RETURNED.
- The *Examination Security Monitor Program* detects pairs of students with unusually similar answer patterns on multiple-choice exams. Data generated by this program can be used as admissible evidence, either to initiate or corroborate an investigation or a charge of cheating under Section 16 of the *Code of Student Conduct and Disciplinary Procedures*.

Winter 2016
COMP-250: Introduction to Computer Science

Lectures 1-26, January-April, 2016

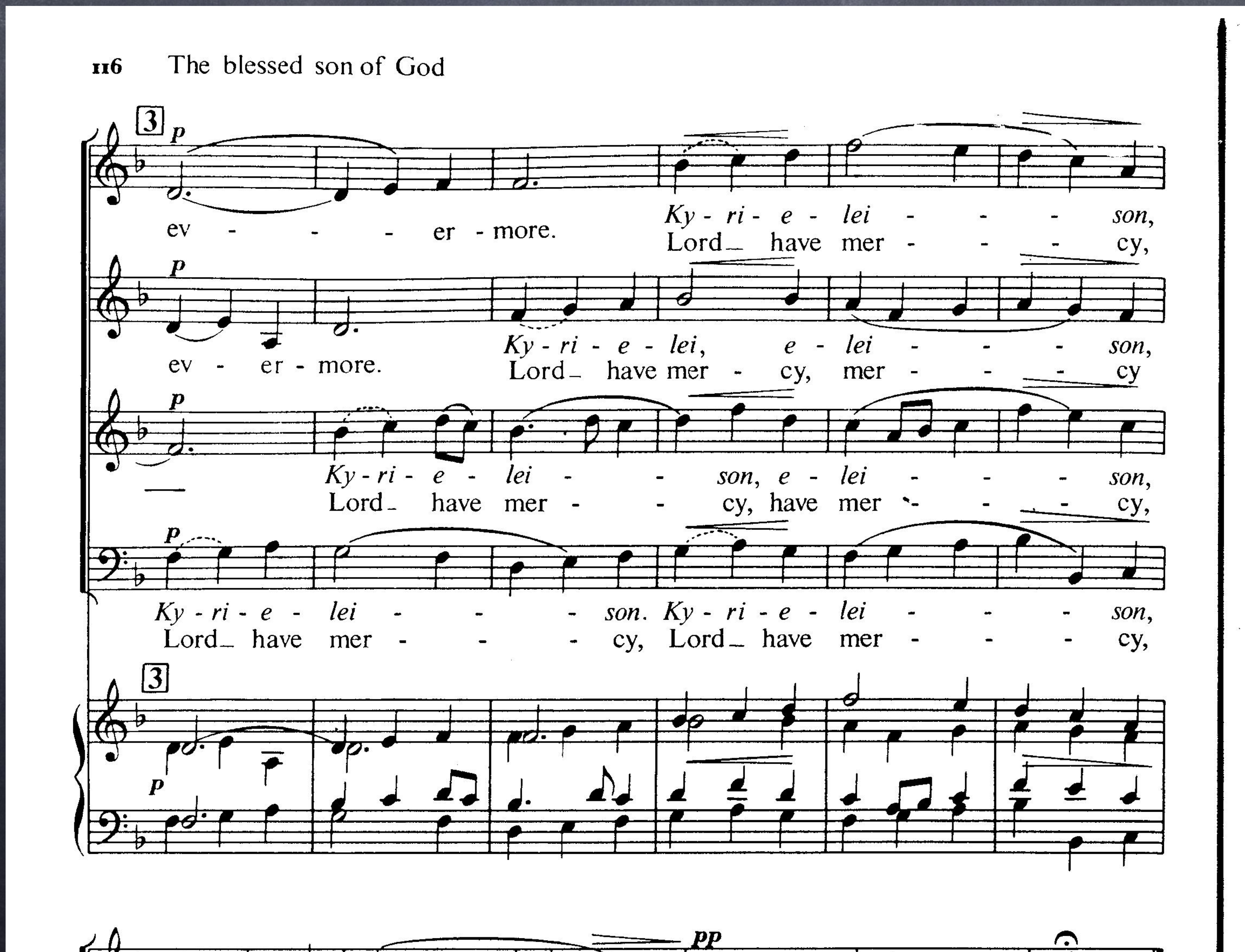
Algorithms

- Informal definition

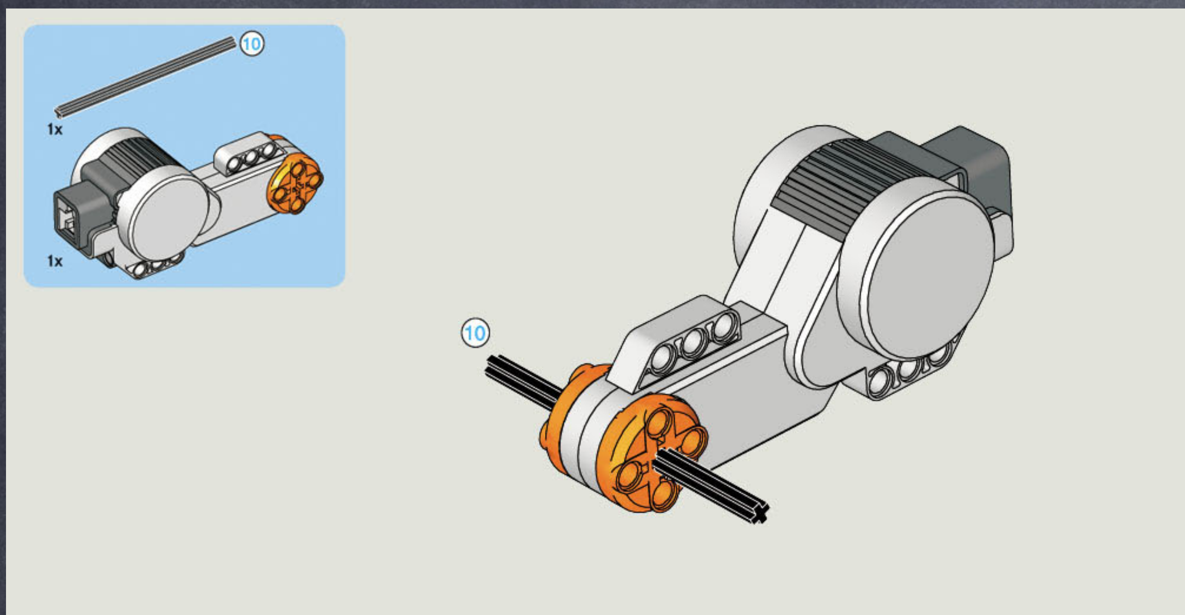
  An algorithm is the specification of a sequence of instructions to be carried out by a processor.

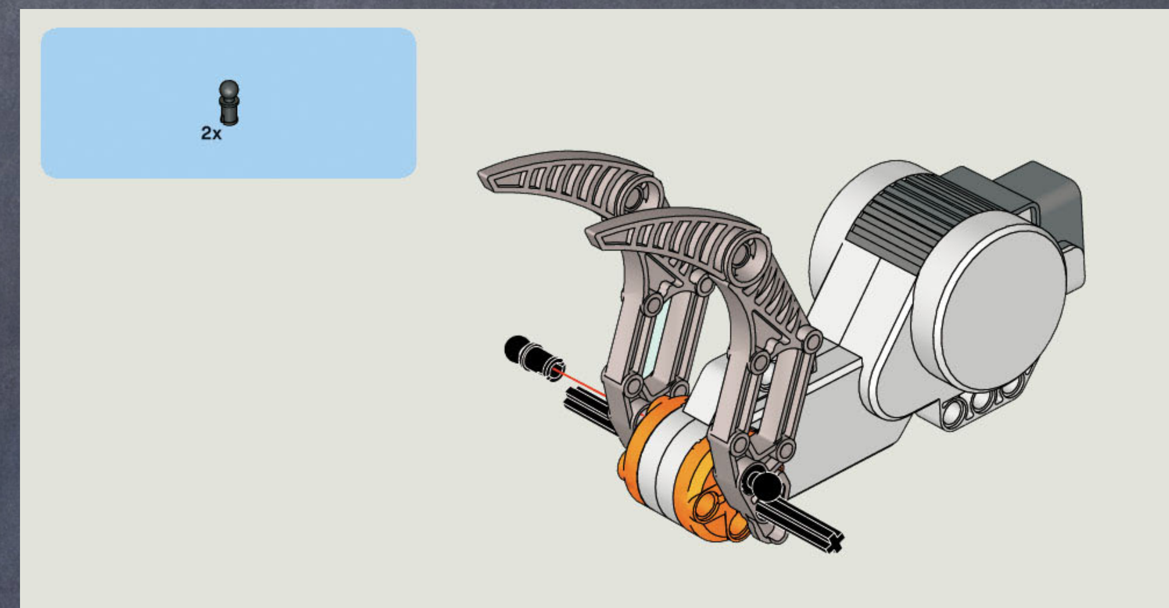
- Algorithms can be run on a computer, but they don't have to:
  - Mayas had algorithms to predict solar eclipses centuries in advance
  - Egyptians had algorithms to build pyramids
  - Indians had algorithms for factorizing polynomials
  - Greeks had algorithms to build all kinds of geometric construction using only a compass and straight lines.

Music SCORE

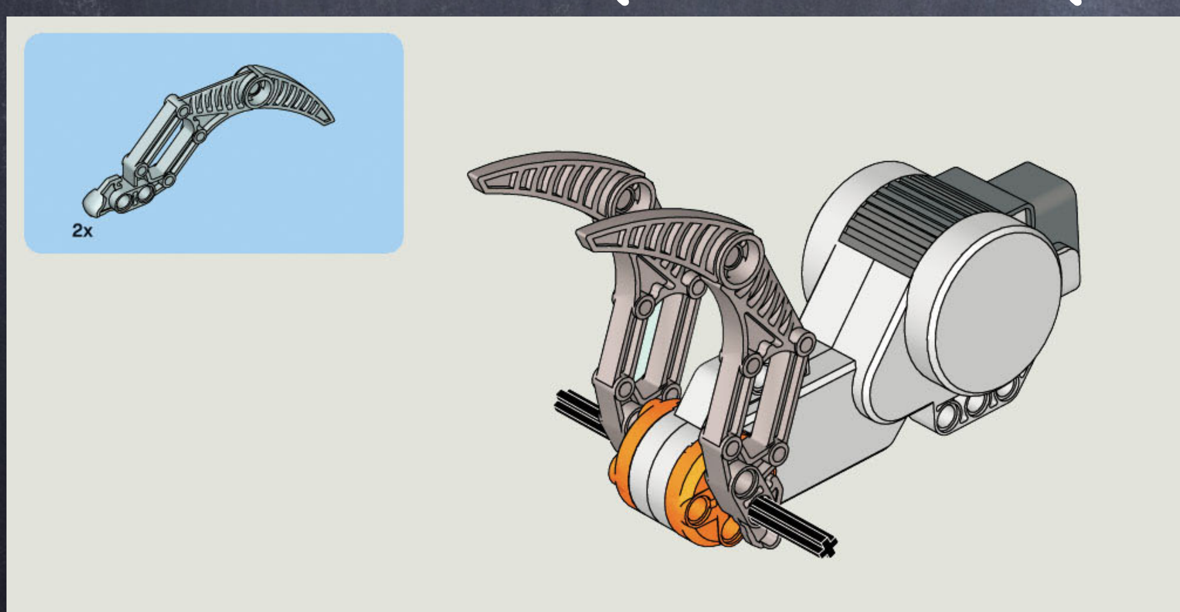

116 The blessed son of God
3
p
ev - - - er - more. Kyrie - e - lei - - - son,
Lord have mer - - - cy,
p
ev - er - more. Ky - ri - e - lei, e - lei - - - son,
Lord have mer - cy, mer - - - cy
p
Ky - ri - e - lei - - son, e - lei - - - son,
Lord have mer - - cy, have mer - - - cy,
p
Ky - ri - e - lei - - - son. Ky - ri - e - lei - - - son,
Lord have mer - - - cy, Lord have mer - - - cy,
3
p
pp

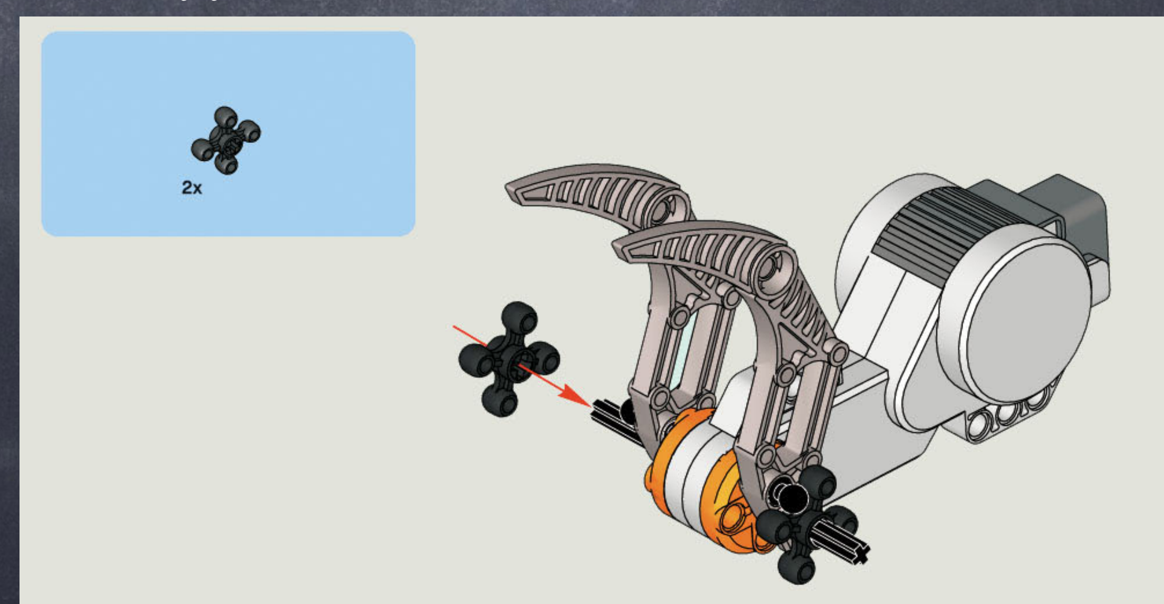
Assembly Instruction





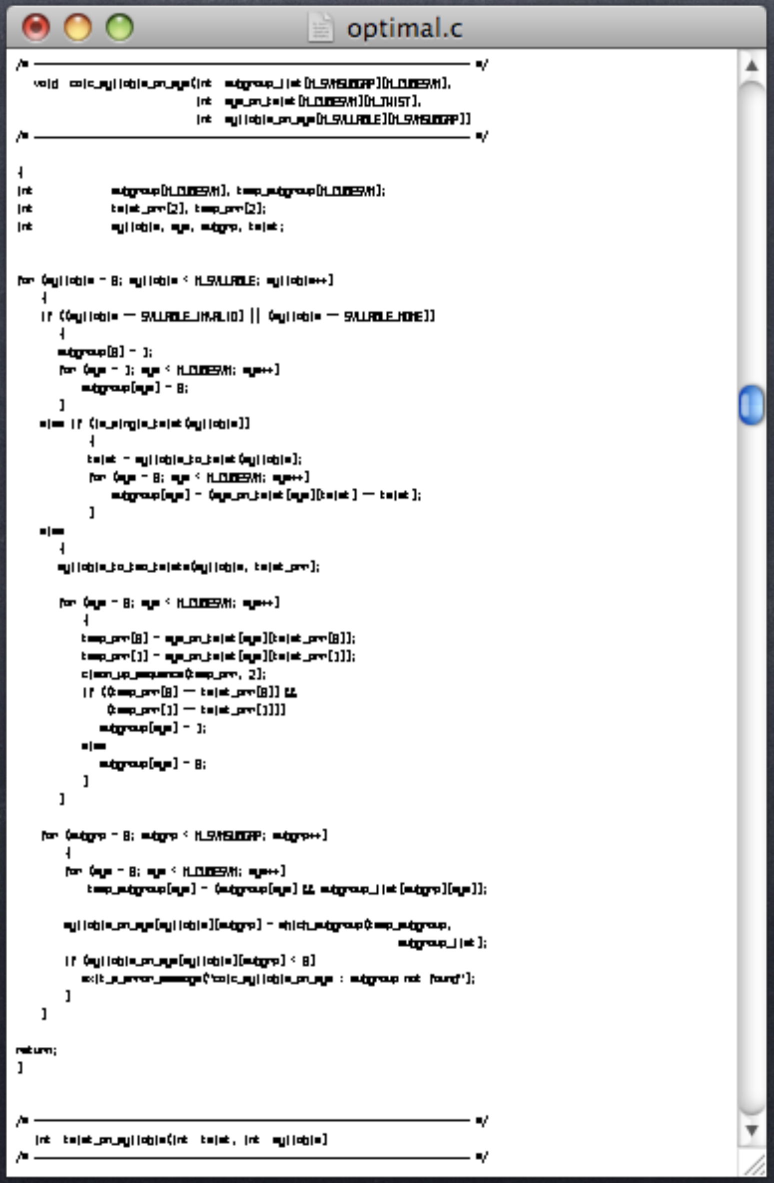
- LEGO (RoboArm (Machine)) instructions

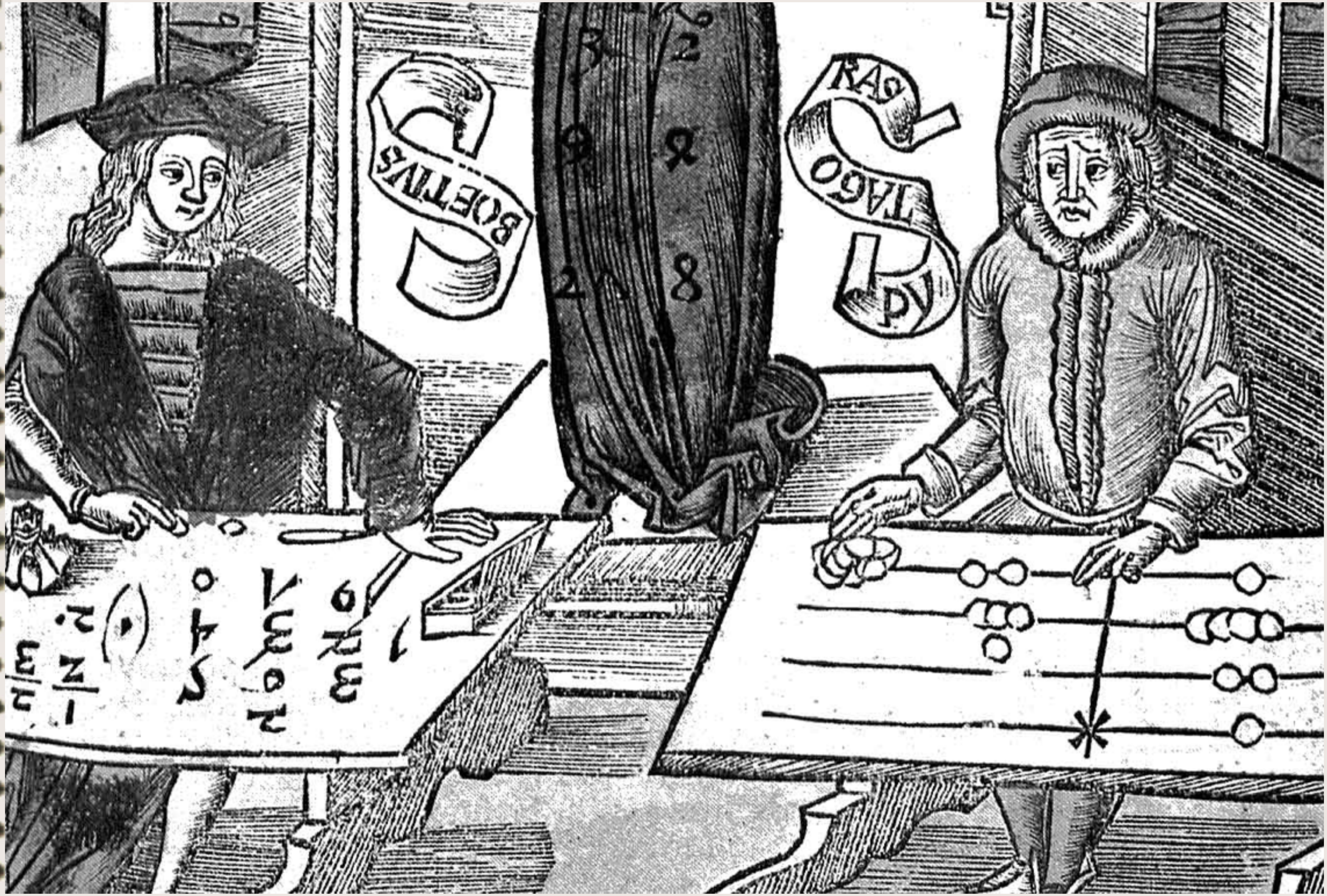




Computer Program

- C program





Fight around 1503 about calculation method

TODAY

CD / DVD
/ Blu-ray

Smart phones

Flat Screens

laptop

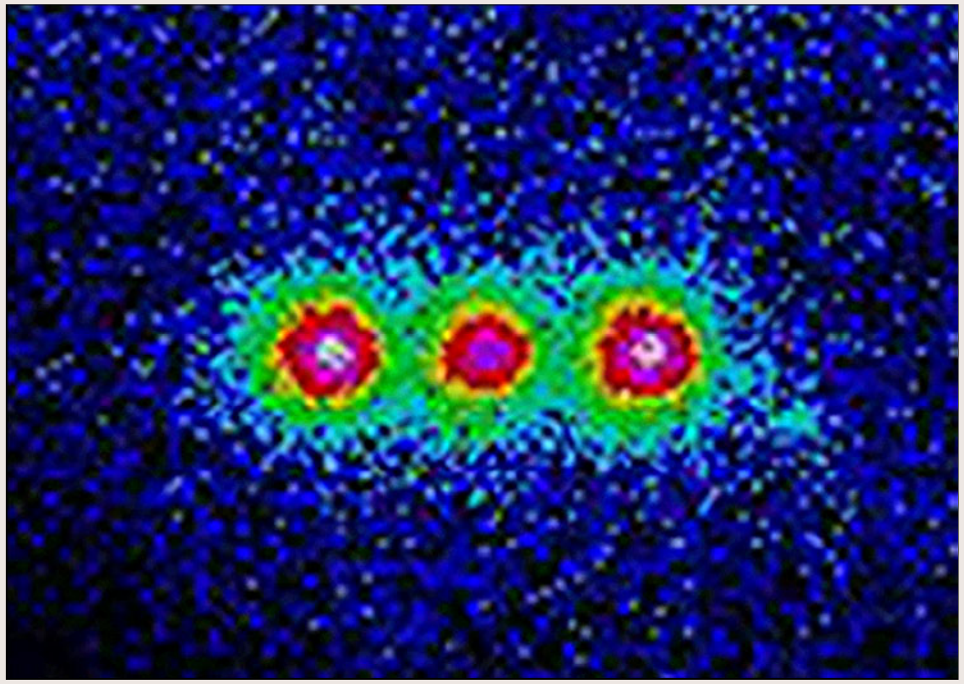
MP3 players

U S B connectivity

Electronic Tablets

scanner

TOMORROW ...???



Computer Science

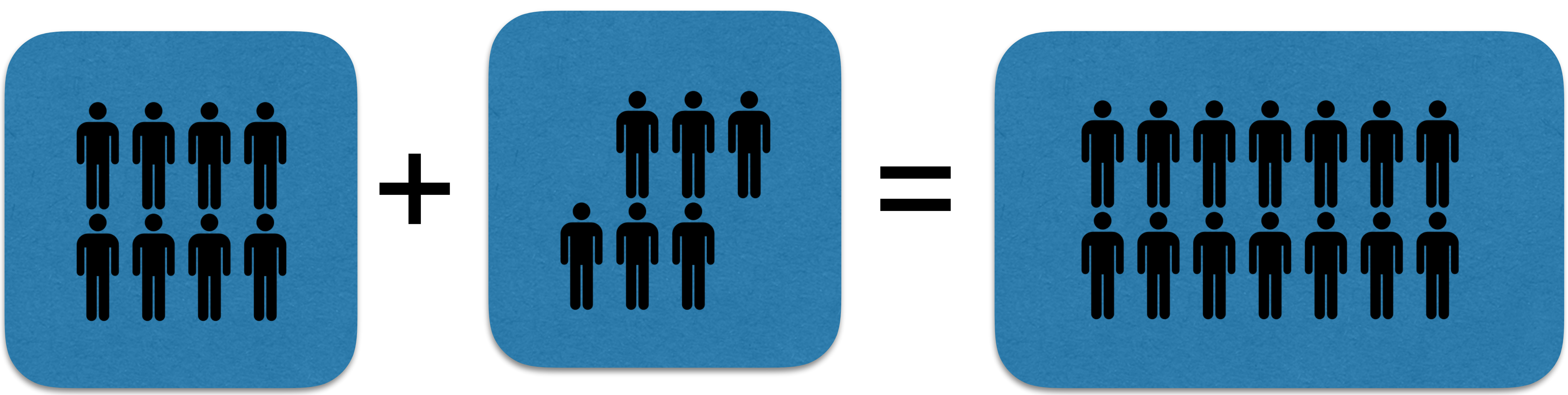
- **Computer Science** is the study of algorithms for computing machines.
- (Formal) Definition of an Algorithm

A **well-ordered** collection of **unambiguous effectively computable** operations that when executed produces a **result** and halts in a **finite** amount of time.

Winter 2016
COMP-250: Introduction to Computer Science

Lecture 2, January 14, 2016

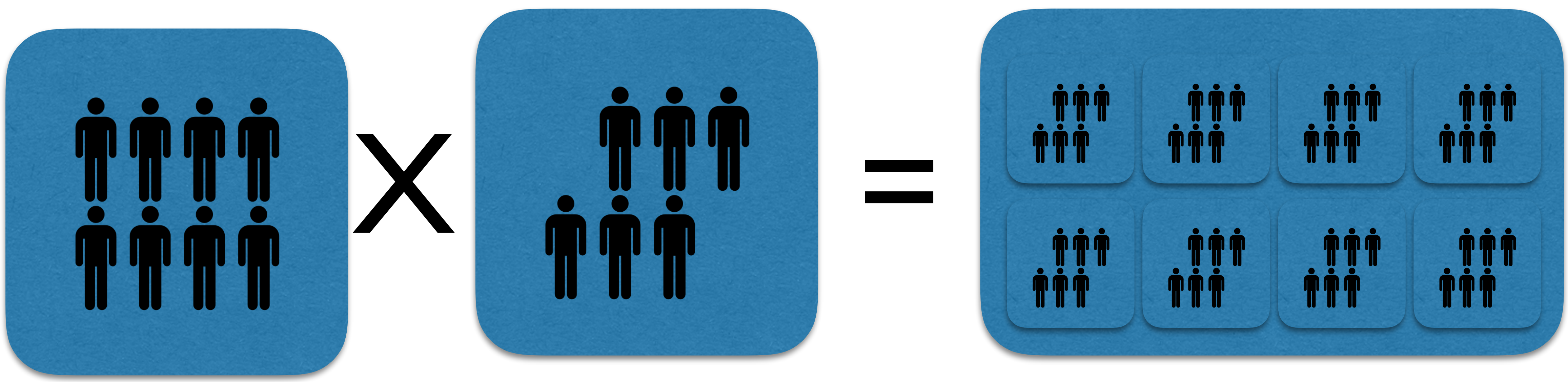
Grade School Algorithms



Representation quite inefficient

"+" easy to describe

Grade School Algorithms



Representation quite inefficient

"X" easy to describe

Grade School Algorithms

Algorithm 1 Addition (base 10): Add two N digit numbers a and b which are represented as arrays of digits

+	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	7	8	9	10	11
3	4	5	6	7	8	9	10	11	12
4	5	6	7	8	9	10	11	12	13
5	6	7	8	9	10	11	12	13	14
6	7	8	9	10	11	12	13	14	15
7	8	9	10	11	12	13	14	15	16
8	9	10	11	12	13	14	15	16	17
9	10	11	12	13	14	15	16	17	18

Grade School Algorithms

Algorithm 1 Addition (base 10): Add two N digit numbers a and b which are represented as arrays of digits

$$\begin{array}{cr} & {\scriptstyle 0\,0\,0\,1} \\ & 2343 \\ + & 4519 \\ \hline & 6862 \end{array}$$

+	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	7	8	9	10	11
3	4	5	6	7	8	9	10	11	12
4	5	6	7	8	9	10	11	12	13
5	6	7	8	9	10	11	12	13	14
6	7	8	9	10	11	12	13	14	15
7	8	9	10	11	12	13	14	15	16
8	9	10	11	12	13	14	15	16	17
9	10	11	12	13	14	15	16	17	18

Grade School Algorithms

Algorithm 1 Addition (base 10): Add two N digit numbers a and b which are represented as arrays of digits

$carry \leftarrow 0$
for $i \leftarrow 0$ **to** $N-1$ **do**
  $r[i] \leftarrow R[a[i], b[i], carry]$
  $carry \leftarrow L[a[i], b[i], carry]$
end for
$r[N] \leftarrow carry$

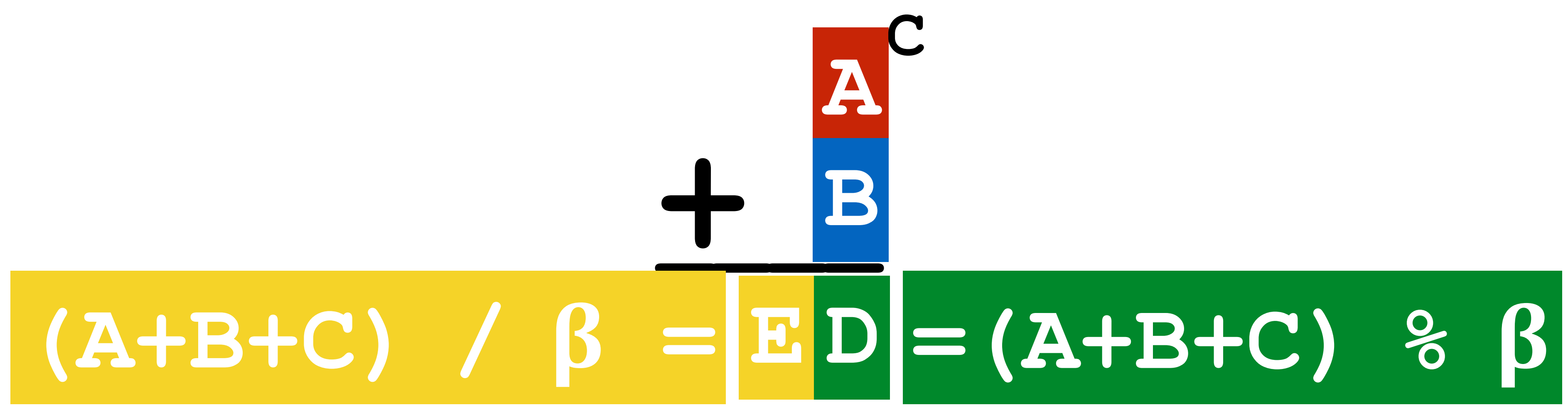
Grade School Algorithms

Algorithm 1 Addition (base 10): Add two N digit numbers a and b which are represented as arrays of digits

$carry = 0$
for $i = 0$ to $N - 1$ **do**
  $r[i] \leftarrow (a[i] + b[i] + carry)\ \%\ 10$
  $carry \leftarrow (a[i] + b[i] + carry)/10$
end for
$r[N] \leftarrow carry$

Grade School Algorithms

Algorithm 1 Addition (base **β**): Add two N **β-git** numbers a and b which are represented as arrays of **β-gits**



Grade School Algorithms

Algorithm 1 Addition (base **β**): Add two **β-git** numbers a and b which are represented as arrays of **β-gits**

$carry = 0$
for $i = 0$ to $N - 1$ **do**
  $r[i] \leftarrow (a[i] + b[i] + carry) \ \% \ \boldsymbol{\beta}$
  $carry \leftarrow (a[i] + b[i] + carry)/\boldsymbol{\beta}$
end for
$r[N] \leftarrow carry$

Subtraction

$$\begin{array}{r} 6343 \\ -\quad \underline{4519} \end{array}$$

+	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	7	8	9	10	11
3	4	5	6	7	8	9	10	11	12
4	5	6	7	8	9	10	11	12	13
5	6	7	8	9	10	11	12	13	14
6	7	8	9	10	11	12	13	14	15
7	8	9	10	11	12	13	14	15	16
8	9	10	11	12	13	14	15	16	17
9	10	11	12	13	14	15	16	17	18

Grade School Algorithms

$$\begin{array}{r} {}^{5}\cancel{6}\,{}^{1}3\;{}^{3}\cancel{4}\,{}^{1}3 \\ -\quad 4519 \\ \hline 1824 \end{array}$$

+	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	7	8	9	10	11
3	4	5	6	7	8	9	10	11	12
4	5	6	7	8	9	10	11	12	13
5	6	7	8	9	10	11	12	13	14
6	7	8	9	10	11	12	13	14	15
7	8	9	10	11	12	13	14	15	16
8	9	10	11	12	13	14	15	16	17
9	10	11	12	13	14	15	16	17	18

Grade School Algorithms

Algorithm 2 Multiplication (base 10) of two numbers a and b

Name: ____________________

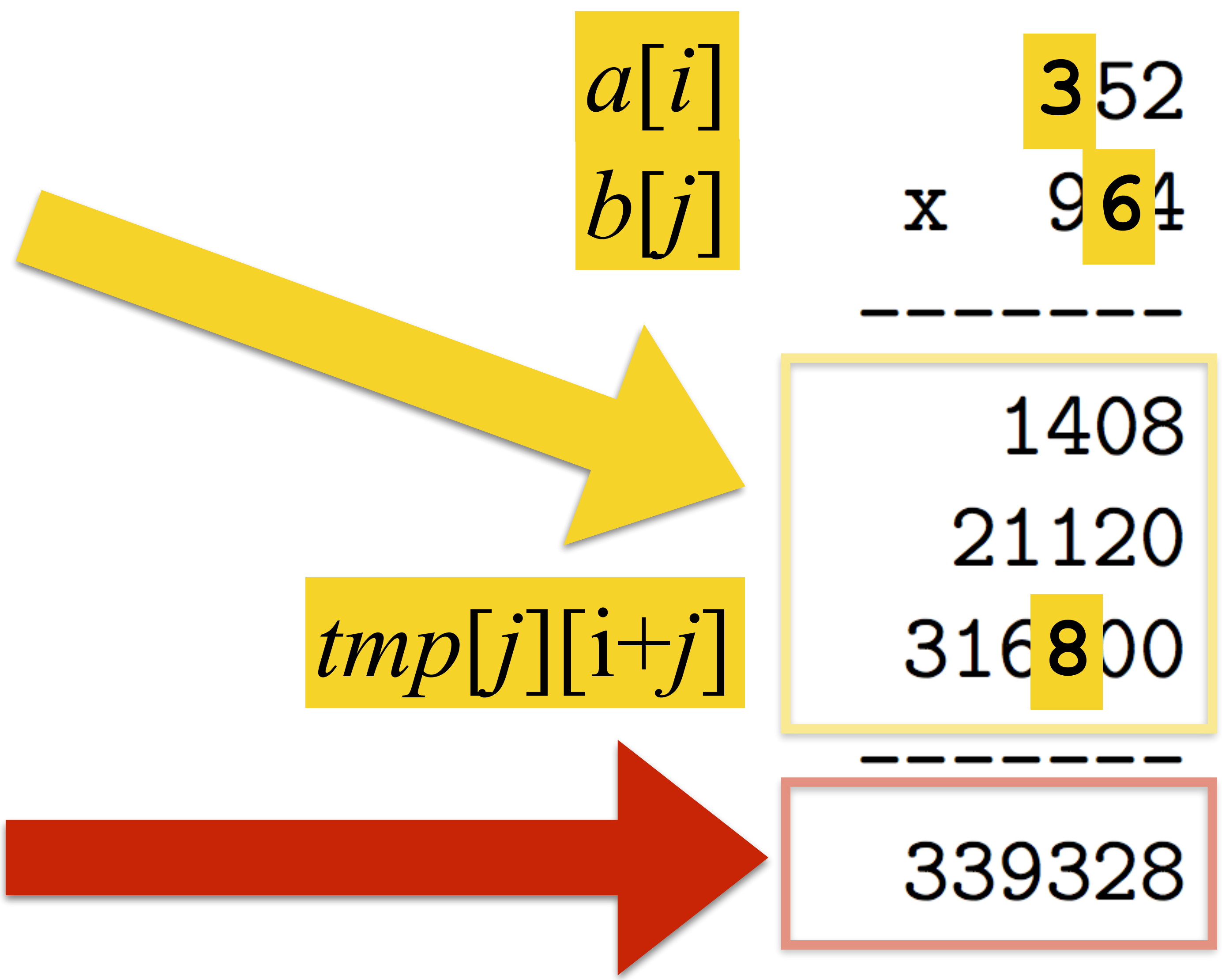
Multiplication Table

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Grade School Algorithms

Algorithm 2 Multiplication (base 10) of two numbers a and b

for $j = 0$ to $N-1$ **do**
  $carry \leftarrow 0$
  for $i = 0$ to $N-1$ **do**
    $prod \leftarrow (a[i] * b[j] + carry)$
    $tmp[j][i+j] \leftarrow prod\%10$
    $carry \leftarrow prod/10$
  end for
  $tmp[j][N+j] \leftarrow carry$
end for
$carry \leftarrow 0$
for $i = 0$ to $2*N-1$ **do**
  $sum \leftarrow carry$
  for $j = 0$ to $N-1$ **do**
    $sum \leftarrow sum + tmp[j][i]$
  end for
  $r[i] \leftarrow sum\%10$
  $carry \leftarrow sum/10$
end for
$r[2*N] \leftarrow carry$



Multiplication

$$
\begin{array}{l}
\textbf{for } j = 0 \text{ to } N-1 \textbf{ do} \\
\quad carry \leftarrow 0 \\
\quad \textbf{for } i = 0 \text{ to } N-1 \textbf{ do} \\
\quad\quad prod \leftarrow (a[i] * b[j] + carry) \\
\quad\quad tmp[j][i+j] \leftarrow prod\%10 \\
\quad\quad carry \leftarrow prod/10 \\
\quad \textbf{end for} \\
\quad tmp[j][N+j] \leftarrow carry \\
\textbf{end for}
\end{array}
$$

Multiplication

$carry \leftarrow 0$

for $i = 0$ to $2 * N - 1$ **do**

$\quad sum \leftarrow carry$

$\quad$ **for** $j = 0$ to $N - 1$ **do**

$\quad\quad sum \leftarrow sum + tmp[j][i]$

$\quad$ **end for**

$\quad r[i] \leftarrow sum\%10$

$\quad carry \leftarrow sum/10$

end for

$r[2 * N] \leftarrow carry$

Multiplication

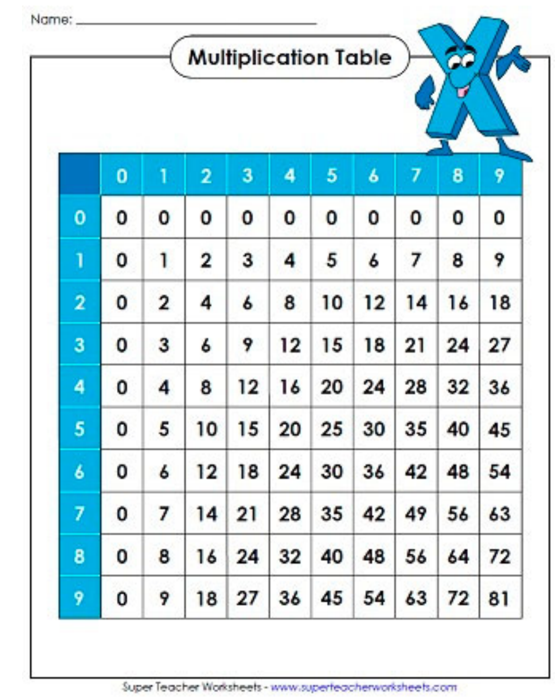
Algorithm 2 Multiplication (base β) of two numbers a and b

for $j = 0$ to $N-1$ **do**
  $carry \leftarrow 0$
  for $i = 0$ to $N-1$ **do**
    $prod \leftarrow (a[i] * b[j] + carry)$
    $tmp[j][i+j] \leftarrow prod\%\beta$
    $carry \leftarrow prod/\beta$
  end for
  $tmp[j][N+j] \leftarrow carry$
end for

$carry \leftarrow 0$
for $i = 0$ to $2*N-1$ **do**
  $sum \leftarrow carry$
  for $j = 0$ to $N-1$ **do**
    $sum \leftarrow sum + tmp[j][i]$
  end for
  $r[i] \leftarrow sum\%\beta$
  $carry \leftarrow sum/\beta$
end for
$r[2*N] \leftarrow carry$

Long Division

```
        ______________
723 |   41672542996
```

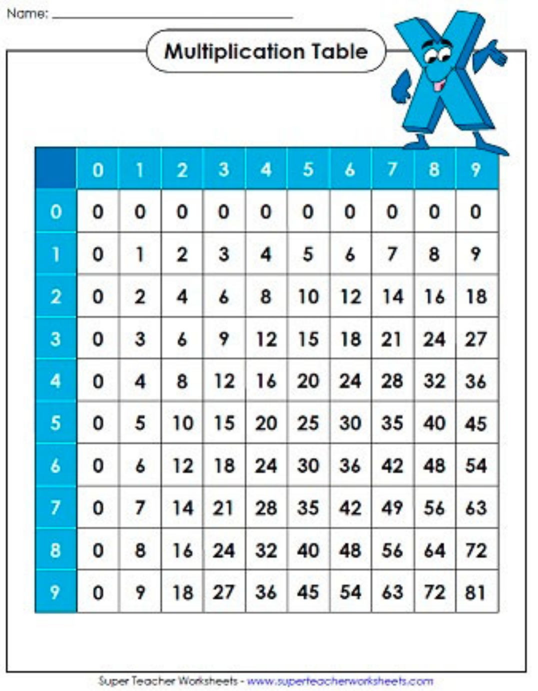

Name: ______

Multiplication Table

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Grade School Algorithms

```
               5 . . .
      ______________
723 |   41672542996
```


Name:

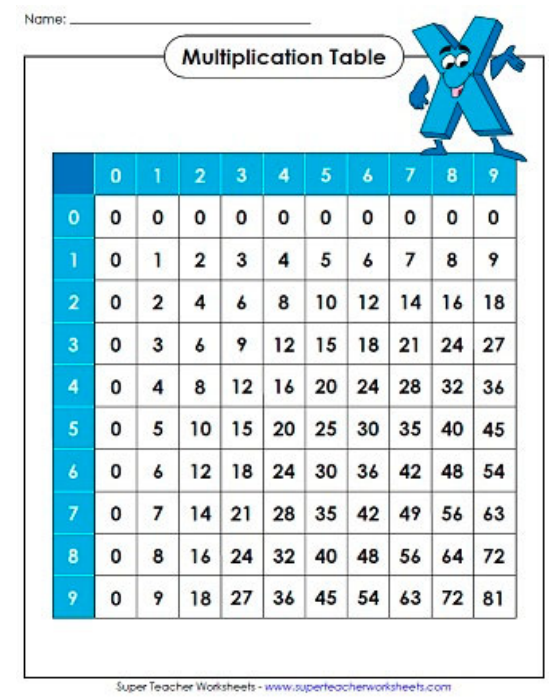
Multiplication Table

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Super Teacher Worksheets - www.superteacherworksheets.com

Grade School Algorithms

```
              57638372
      ______________
723 |               50
```


Name:

Multiplication Table

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Super Teacher Worksheets - www.superteacherworksheets.com

```
41672542996 / 723 = 57638372
41672542996 % 723 = 50
```

Analysis of Addition

$$
\begin{array}{l}
\texttt{cst}\ \{\ carry = 0 \\
\texttt{linear}\ \left\{ \begin{array}{l}
\textbf{for } i = 0 \text{ to } N-1 \ \ \textbf{do} \\
\texttt{cst}\ \left\{ \begin{array}{l}
r[i] \leftarrow (a[i] + b[i] + carry)\ \%\ 10 \\
carry \leftarrow (a[i] + b[i] + carry)/10
\end{array} \right. \\
\textbf{end for}
\end{array} \right. \\
\texttt{cst}\ \{\ r[N] \leftarrow carry
\end{array}
$$

$\texttt{Time(N)} = c_1 + c_2 \times N$

Analysis of Multiplication

quadratic

for $j = 0$ to $N-1$ **do**

cst $\{ carry \leftarrow 0$

linear

for $i = 0$ to $N-1$ **do**

cst

$prod \leftarrow (a[i] * b[j] + carry)$

$tmp[j][i+j] \leftarrow prod\%10$

$carry \leftarrow prod/10$

end for

cst $\{ tmp[j][N+j] \leftarrow carry$

end for

Analysis of Multiplication

quadratic

linear

cst $\{$ $carry \leftarrow 0$

for $i = 0$ to $2 * N - 1$ **do**

cst $\{$ $sum \leftarrow carry$

for $j = 0$ to $N - 1$ **do**

cst $\{$ $sum \leftarrow sum + tmp[j][i]$

end for

cst $\{$ $r[i] \leftarrow sum\%10$

$carry \leftarrow sum/10$

end for

cst $\{$ $r[2 * N] \leftarrow carry$

Analysis of Algorithms

Algorithm 2 Multiplication (base 10) of two numbers a and b

for $j = 0$ to $N - 1$ **do**
  $carry \leftarrow 0$
  for $i = 0$ to $N - 1$ **do**
    $prod \leftarrow (a[i] * b[j] + carry)$
    $tmp[j][i + j] \leftarrow prod\%10$
    $carry \leftarrow prod/10$
  end for
  $tmp[j][N + j] \leftarrow carry$
end for

$carry \leftarrow 0$
for $i = 0$ to $2 * N - 1$ **do**
  $sum \leftarrow carry$
  for $j = 0$ to $N - 1$ **do**
    $sum \leftarrow sum + tmp[j][i]$
  end for
  $r[i] \leftarrow sum\%10$
  $carry \leftarrow sum/10$
end for
$r[2 * N] \leftarrow carry$

$$\mathrm{Time}(N) = c_1 + c_2 \times N + c_3 \times N^2$$

Analysis of Algorithms

Addition

$$\texttt{Time(N)} = c_1 + c_2 \times N$$

Multiplication

$$\texttt{Time(N)} = c_1 + c_2 \times N + c_3 \times N^2$$

Analysis of Algorithms

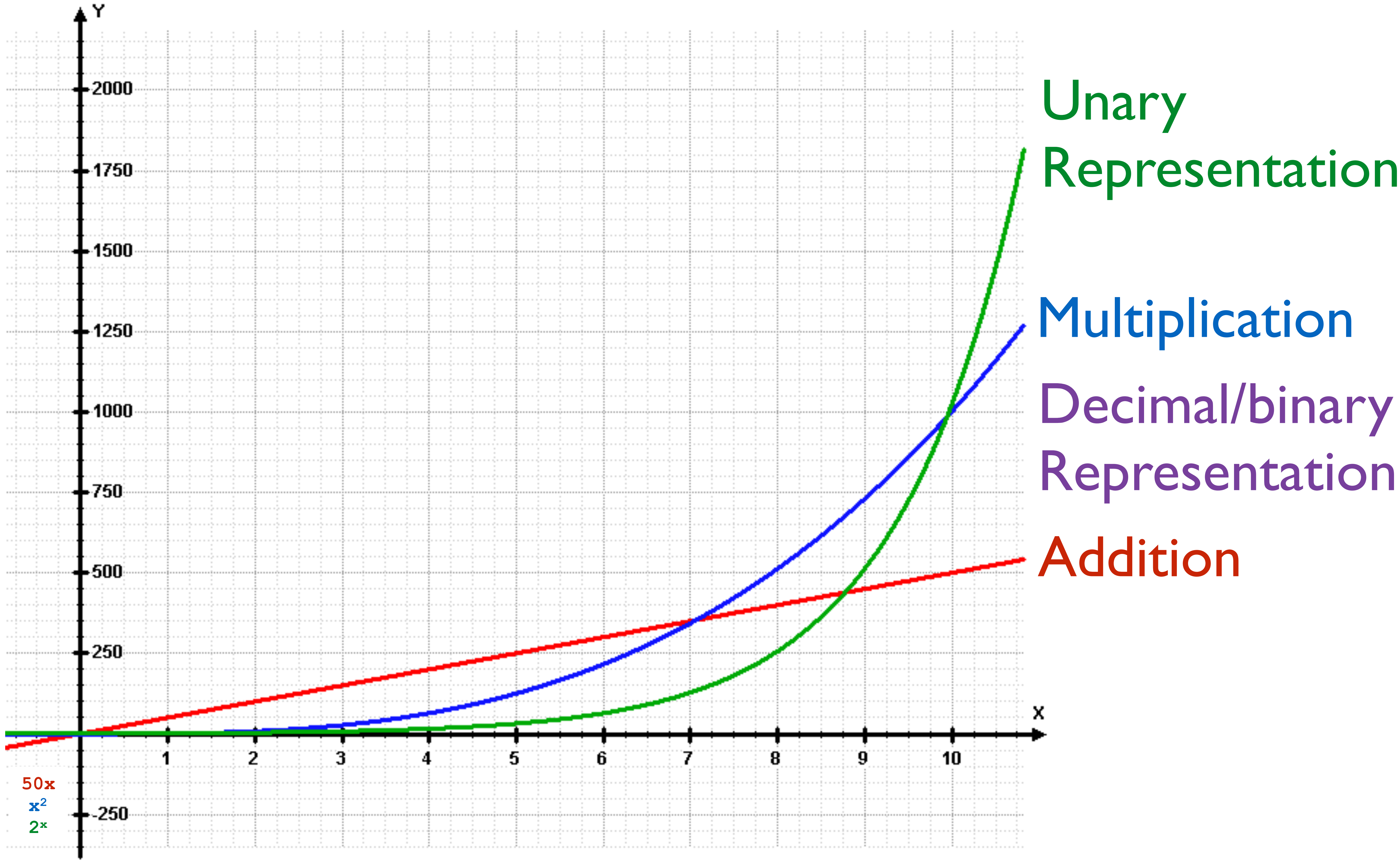
Addition

`Time(N)` **`is`** `O(N)`

Multiplication

`Time(N)` **`is`** $O(N^2)$

Analysis of Algorithms



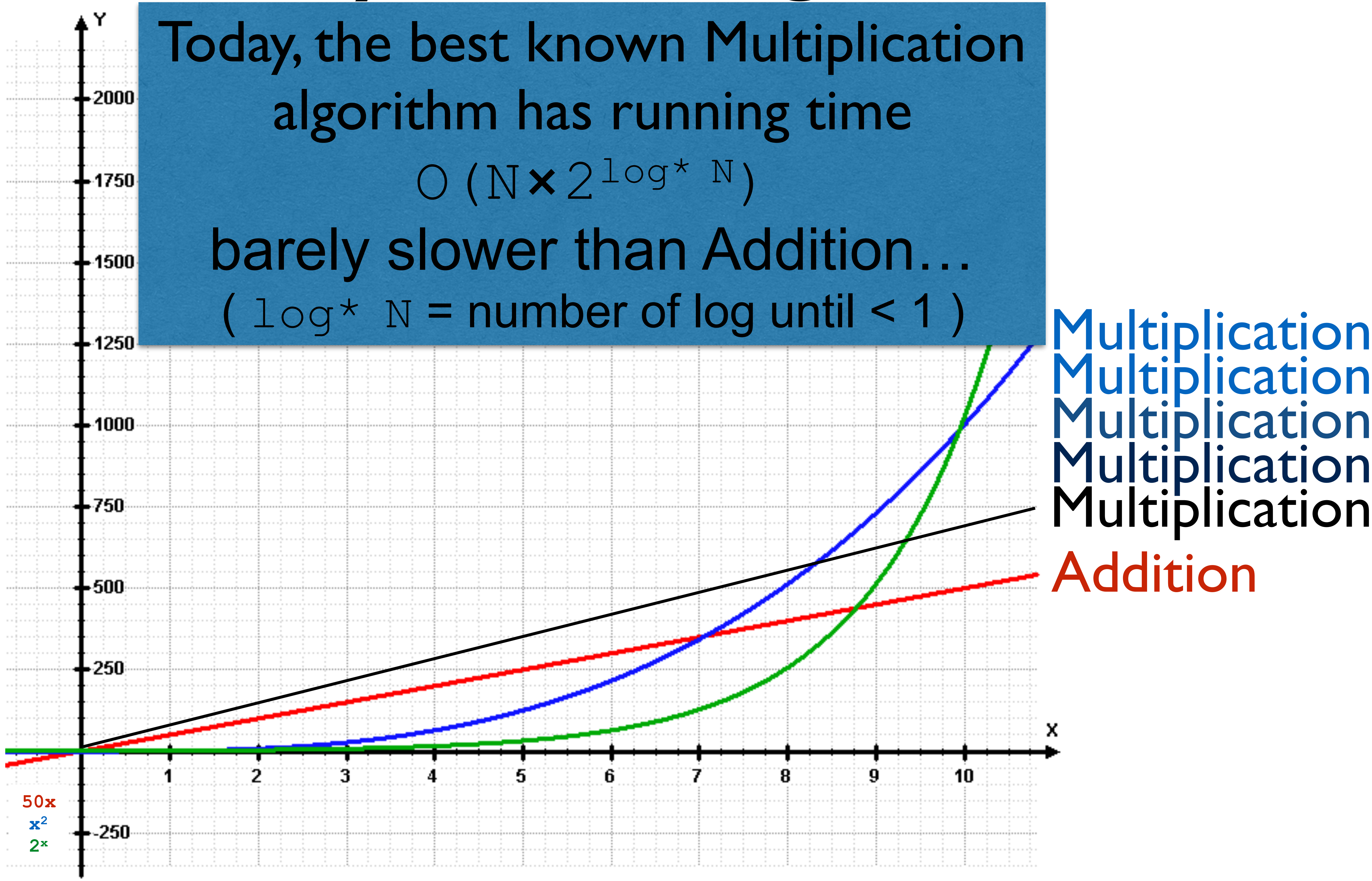
Analysis of Algorithms

Today, the best known Multiplication algorithm has running time

$$O(N \times 2^{\log^* N})$$

barely slower than Addition…

($\log^* N$ = number of log until < 1)



Base 8 vs Base 2

$$(2143)_8$$

$$=(????)_2$$

Base 8 vs Base 2

$$(2143)_8$$

$$=(\ 010\ \ 001\ \ 100\ \ 101\ )_2$$

$$=(10001100101)_2$$

Powers of 2 in Base 10

$2^0=1$ $2^1=2$ $2^2=4$

$2^3=8$ $2^4=16$ $2^5=32$

$2^6=64$ $2^7=128$ $2^8=256$

$2^9=512$ $2^{10}=1024$ $2^{11}=2048$

$2^{12}=4096$ $2^{13}=8192$ $2^{14}=16384$

$2^{15}=32768$ $2^{16}=65536$

$2^{32} = 4\,294\,967\,296$

Powers of 10 in Base 2

10^0=1

10^1=1010

10^2=1100110

10^3=1111101000 $\approx 2^{10}$

10^4=10011100010000

to Base 2

Algorithm 3 Convert integer to binary

INPUT: a number m

OUTPUT: the number m **expressed in base 2 using a bit array** $b[\,]$

$i \leftarrow 0$
while $m > 0$ **do**
  $b[i] \leftarrow m\%2$
  $m \leftarrow m/2$
  $i \leftarrow i + 1$
end while

to Base β

Algorithm 3 Convert integer to binary

INPUT: a number m

OUTPUT: the number m **expressed in base β using a bit array** $b[\,]$

$i \leftarrow 0$

while $m > 0$ **do**

$b[i] \leftarrow m\% \beta$

$m \leftarrow m/\beta$

$i \leftarrow i + 1$

end while

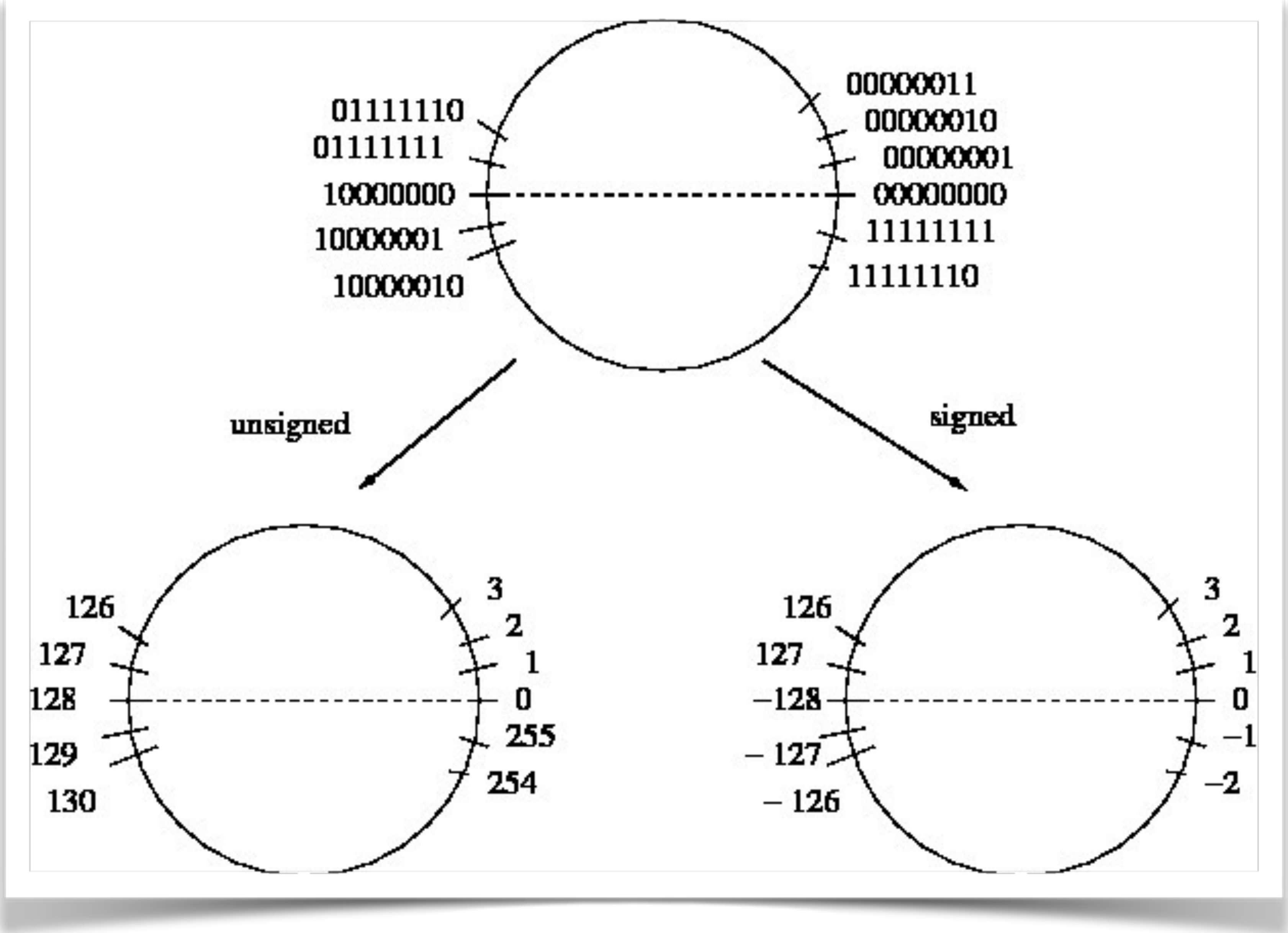
Fractional Numbers

$$26.375 = (11010.\_\_\_\_\_)_2$$

$$0.375 = 1/4+1/8 = (0.011)_2$$

$$26.375 = (11010.011)_2$$

More Binary Representation



Representation

binary	signed	unsigned
00000000	0	0
00000001	1	1
:	:	:
01111111	127	127
10000000	-128	128
10000001	-127	129
:	:	:
11111111	-1	255

Representation

binary	signed	unsigned
0000000000000000	0	0
0000000000000001	1	1
⋮	⋮	⋮
0000000001111111	127	127
0000000010000000	128	128
0000000010000001	129	129
⋮	⋮	
0111111111111111	$2^{15} - 1$	$2^{15} - 1$
1000000000000000	-2^{15}	2^{15}
1000000000000001	$-2^{15} + 1$	$2^{15} + 1$
⋮	⋮	
1111111101111111	-129	$2^{16} - 129$
1111111110000000	-128	$2^{16} - 128$
1111111110000001	-127	$2^{16} - 127$
⋮	⋮	
1111111111111111	-1	$2^{16} - 1$

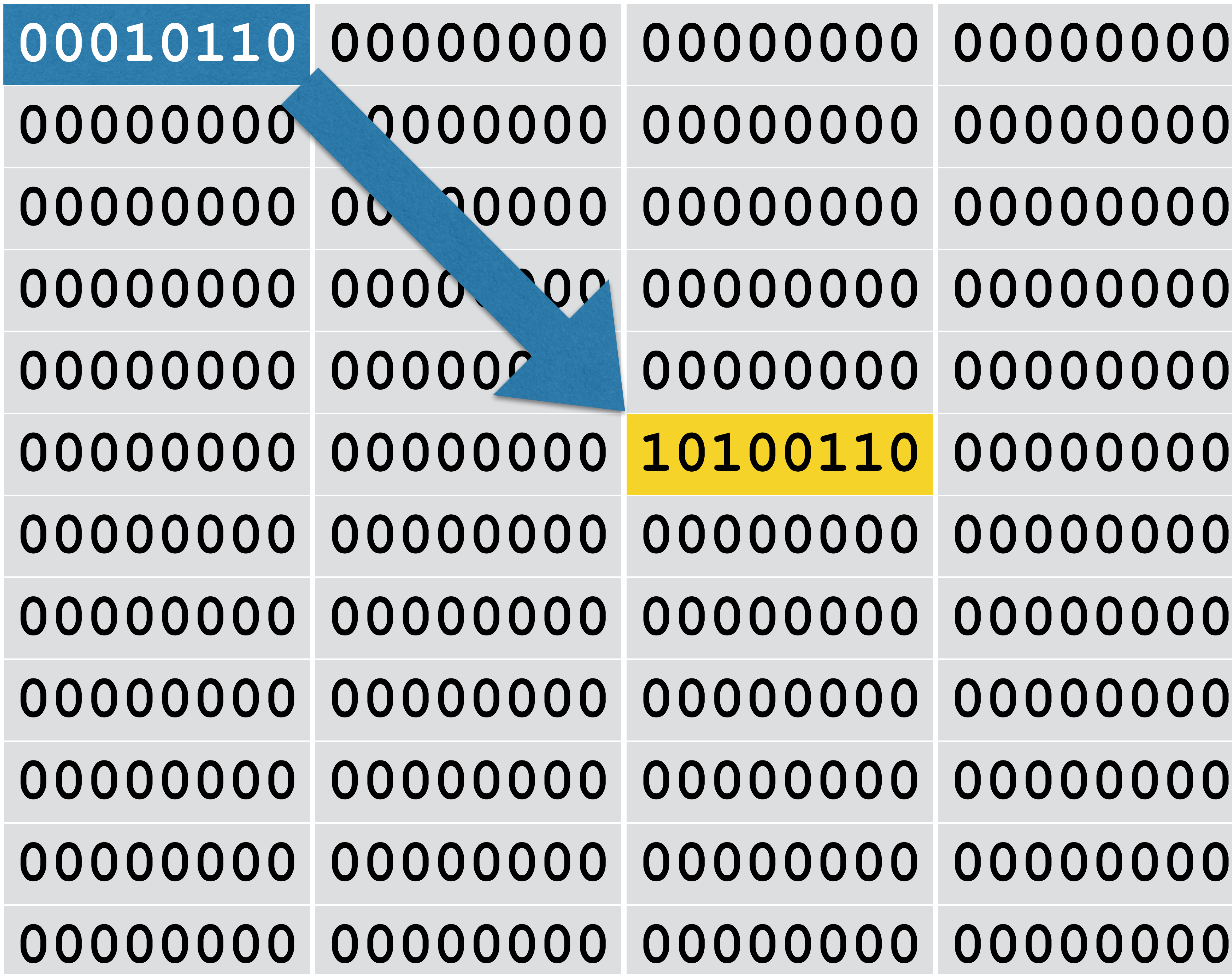
A Byte

10100110

A Byte

00010110	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	10100110	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

An address


00010110 00000000 00000000 00000000
00000000 0000000 00000000 00000000
00000000 00 00000 00000000 00000000
00000000 0000 00 00000000 00000000
00000000 00000 00000000 00000000
00000000 00000000 10100110 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000

A (32-bit) address

00000000	00000000	00000000	00010110
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	10100110	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

A (64-bit) address

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00010110
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	10100110	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Java Primitive Types

Boolean	00000000	00000000	00000000	00000000
Byte	00000000	00000000	00000000	00000000
Char	00000000	00000000	00000000	00000000
Short	00000000	00000000	00000000	00000000
Int	00000000	00000000	00000000	00000000
Long	00000000	10100110	00000000	00000000
	00000000	00000000	00000000	00000000
Float	00000000	00000000	00000000	00000000
Double	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

(32-bit) addresses

00000000	00000000	00000000	00010010
00000000	00000000	00000000	00101010
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	10100110	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Java Reference Types

32-Bit

Address	00000000	00000000	00000000	00000000
Address	00000000	10100110	00000000	00000000
	00000000	00000000	00000000	00000000

64-Bit

byte[] a;

a:	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

a=new byte[3];

a:	00000000	00000000	00000000	00010010
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

a[0]=166;

a:	00000000	00000000	00000000	00010010
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	10100110	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

int[ ] b;

a:	00000000	00000000	00000000	00010010
b:	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	10100110	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

b=new int[2];

a:	00000000	00000000	00000000	00010010
b:	00000000	00000000	00000000	00101010
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	10100110	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

b[1]=-1;

a:	00000000	00000000	00000000	00010010
b:	00000000	00000000	00000000	00101010
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	10100110	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	11111111	11111111
	11111111	11111111	00000000	00000000

Sorting

ALGORITHM: INSERTION SORT

INPUT: an array $a[\,]$ with N elements that can be compared $(<, =, >)$

OUTPUT: the array $a[\,]$ containing the same elements, in increasing order

for $k = 1$ to $N - 1$ **do**
  $tmp \leftarrow a[k]$
  $i \leftarrow k$
  while $(i > 0)$ & $(tmp < a[i-1])$ **do**
    $a[i] \leftarrow a[i-1]$
    $i \leftarrow i - 1$
  end while
  $a[i] = tmp$
end for

http://tech-algorithm.com/articles/insertion-sort

Analysis of Insertion Sort

ALGORITHM: INSERTION SORT

INPUT: an array $a[\,]$ with N elements that can be compared $(<, =, >)$

OUTPUT: the array $a[\,]$ containing the same elements, in increasing order

$$
\left.\begin{array}{l}
\textbf{for } k = 1 \text{ to } N-1 \textbf{ do} \\
\quad \left.\begin{array}{l} tmp \leftarrow a[k] \\ i \leftarrow k \end{array}\right\} \texttt{cst} \\
\quad \left.\begin{array}{l} \textbf{while } (i > 0)\ \&\ (tmp < a[i-1]) \textbf{ do} \\ \quad \left.\begin{array}{l} a[i] \leftarrow a[i-1] \\ i \leftarrow i-1 \end{array}\right\} \texttt{cst} \\ \textbf{end while} \end{array}\right\} \texttt{cst\textasciitilde linear} \\
\quad \left.a[i] = tmp\right\} \texttt{cst} \\
\textbf{end for}
\end{array}\right\} \texttt{linear\textasciitilde quadratic}
$$

Analysis of Insertion Sort

ALGORITHM: INSERTION SORT

INPUT: an array $a[\,]$ with N elements that can be compared $(<, =, >)$

OUTPUT: the array $a[\,]$ containing the same elements, in increasing order

for $k = 1$ to $N - 1$ **do**
    $tmp \leftarrow a[k]$
    $i \leftarrow k$ } cst
    while $(i > 0)$ & $(tmp < a[i-1])$ **do**
        $a[i] \leftarrow a[i-1]$
        $i \leftarrow i - 1$ } cst   } cst
    end while
    $a[i] = tmp$ } cst
end for   } linear

$$\text{Time}(N) \geq c_1 + c_2 \times N$$

Analysis of Insertion Sort

ALGORITHM: INSERTION SORT

INPUT: an array $a[\,]$ with N elements that can be compared $(<, =, >)$

OUTPUT: the array $a[\,]$ containing the same elements, in increasing order

```
for k = 1 to N − 1 do                                        ⎫
    tmp ← a[k]                   } cst                       ⎪
    i ← k                                                    ⎪
    while (i > 0) & (tmp < a[i − 1]) do  ⎫                   ⎪
        a[i] ← a[i − 1]          } cst   ⎬ linear            ⎬ quadratic
        i ← i − 1                        ⎪                   ⎪
    end while                            ⎭                   ⎪
    a[i] = tmp                   } cst                       ⎪
end for                                                      ⎭
```

$$\text{Time}(N) \leq c_1 + c_2 \times N + c_3 \times N^2$$

Analysis of Algorithms

Best Case

`Time(N)` **`is`** Ω`(N)`

Worst Case

`Time(N)` **`is`** `O(N`^{2}`)`

Linked Lists

`List = ordered set of elements.`

$$(a_0, a_1, \ldots, a_{\mathtt{size-1}})$$

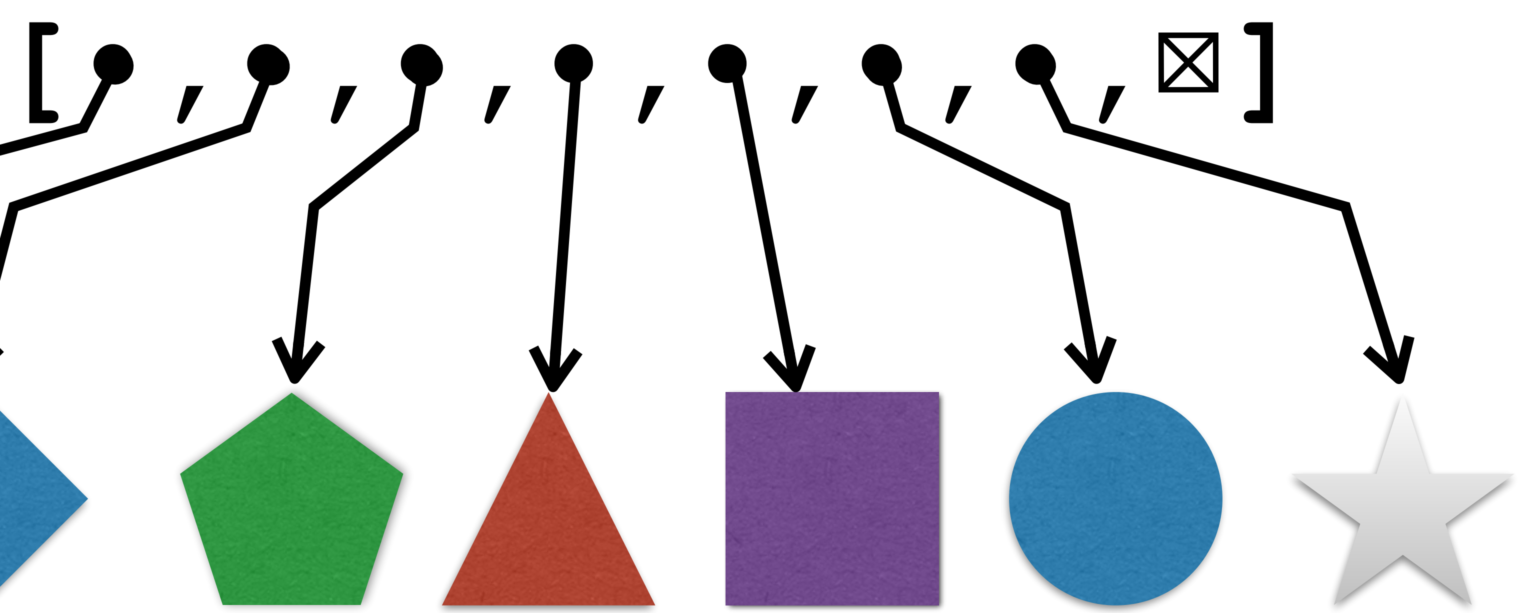
`Size = number of elements.`

Array of integers:

```
 0   1   2   3   4   5   6   7
[5 , 2 , 9 , 3 , 3 , 1 , 7 , 0]
```

Array of shapes:

```
 0   1   2   3   4   5   6   7
```



Adding element to Front

```
//   add new element to front of the list
//   assuming that there is room left in the array
//
     for (i = size; i > 0; i--)
         a[i] = a[i-1]
     a[0] = new element
     size = size + 1
```

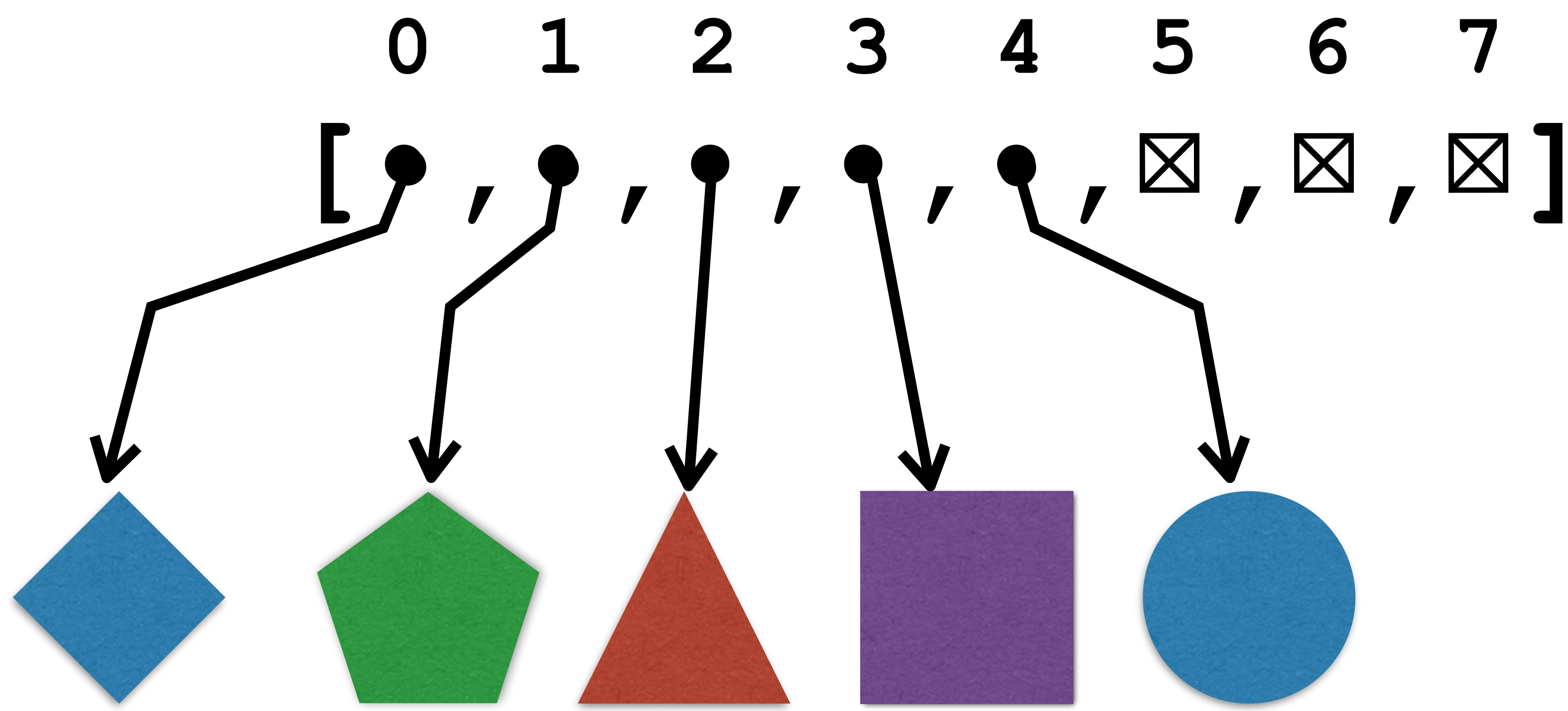
Removing element at Front

```
//  remove the element at front of the list
//
    for (i = 1; i < size-1; i++)
        a[i-1] = a[i]
    a[size-1] = null
    size = size - 1
```

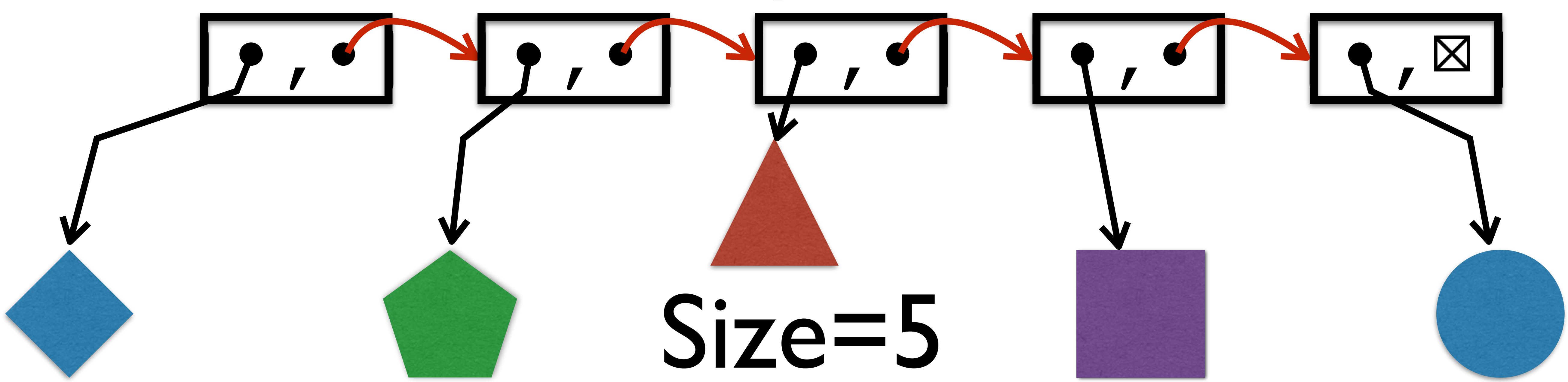
Adding/Removing at End

```
//   add new last element to the list
//   assuming that there is room left in the array
//
     a[size] = new element
     size = size + 1



//   remove the last element from the list
//
     a[size-1] = null
     size = size - 1
```

Array of shapes:

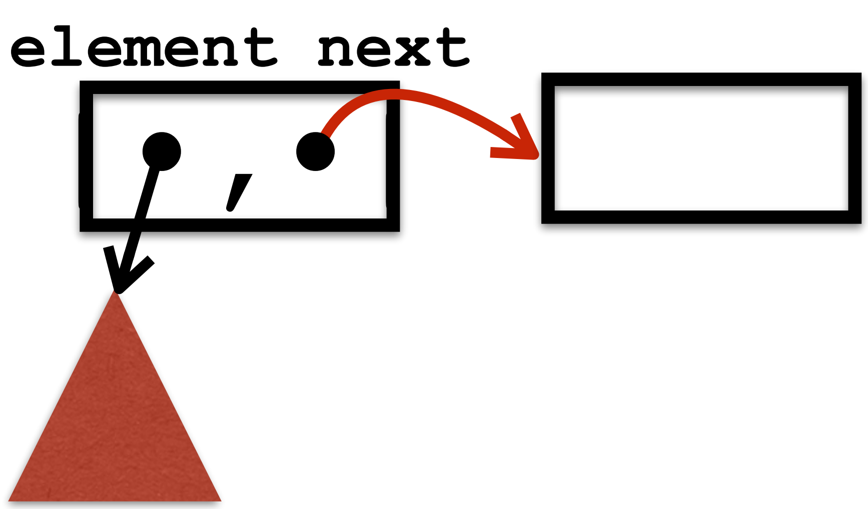


Linked list of shapes:



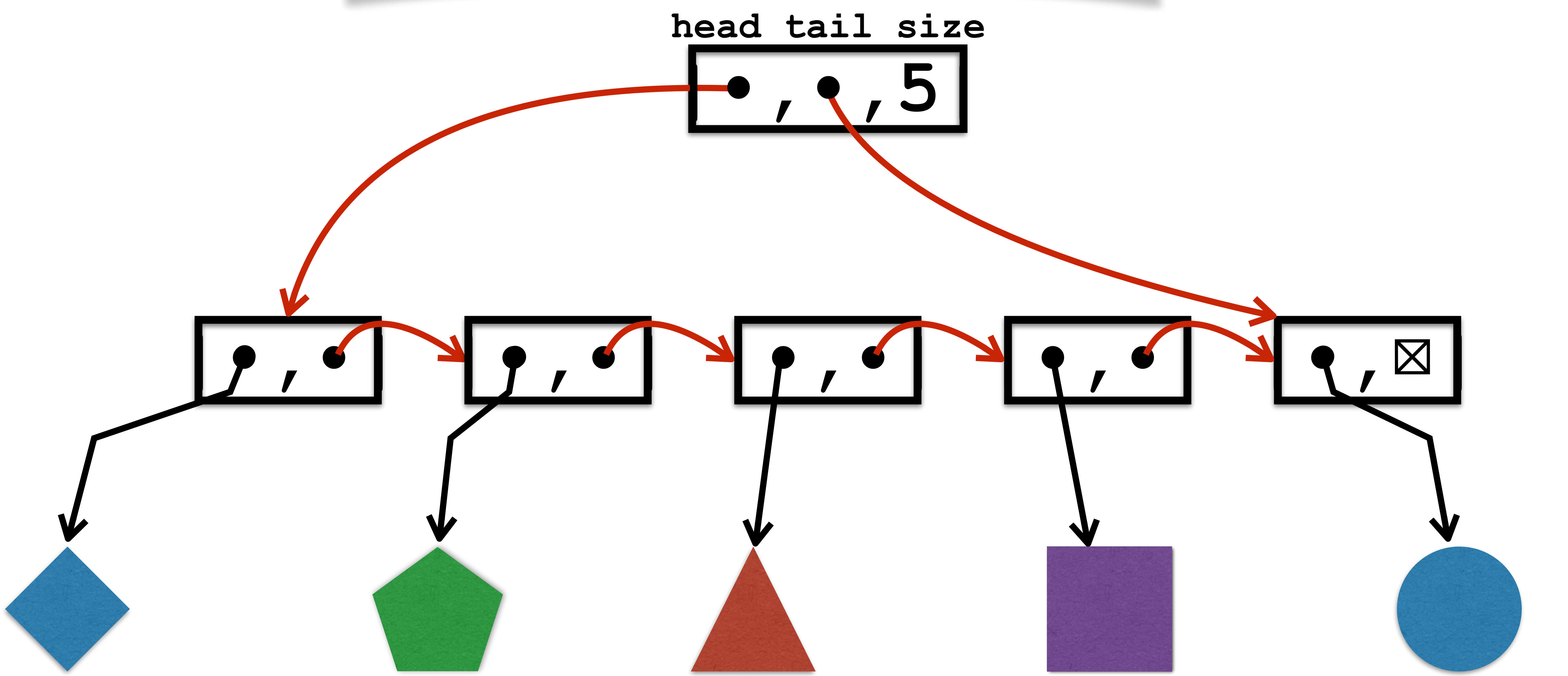
(Singly) Linked List Node

```
class  SNode{
    Type            element;
    SNode           next;
}
```

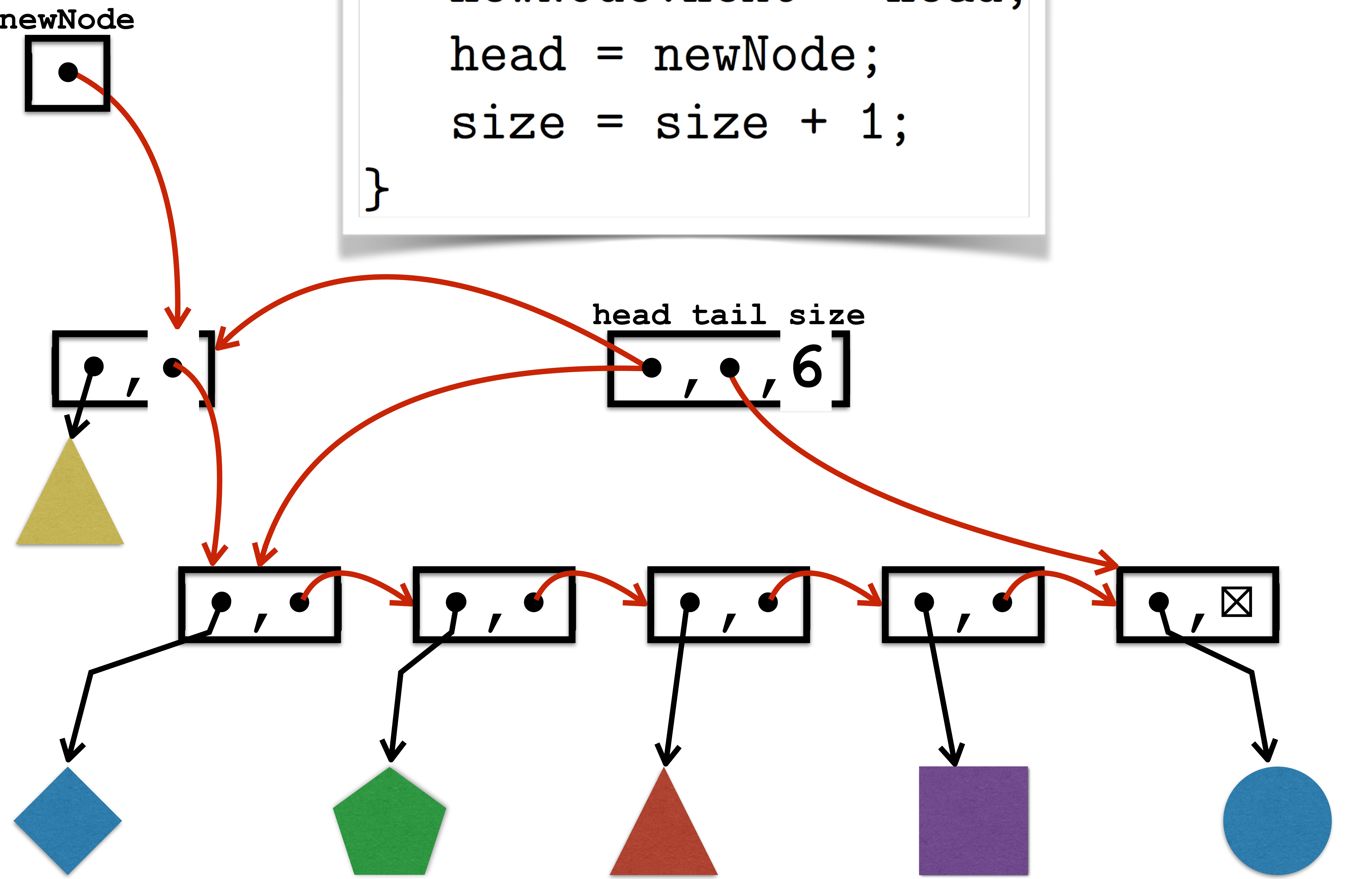


```
class SLinkedList{
    SNode       head;
    SNode       tail;
    integer     size;
}
```

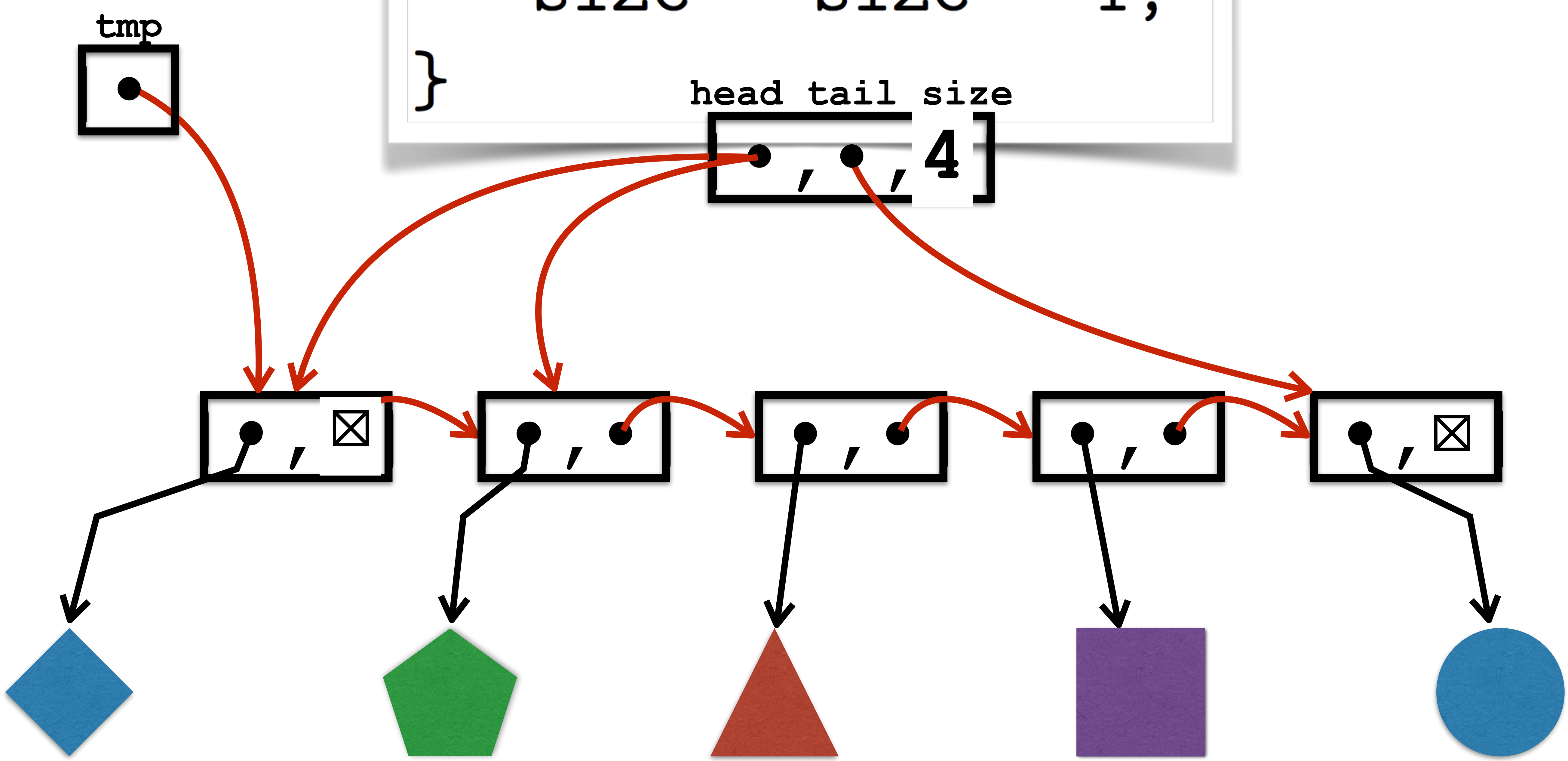
```
class  SLinkedList{
    SNode          head;
    SNode          tail;
    integer        size;
}
```



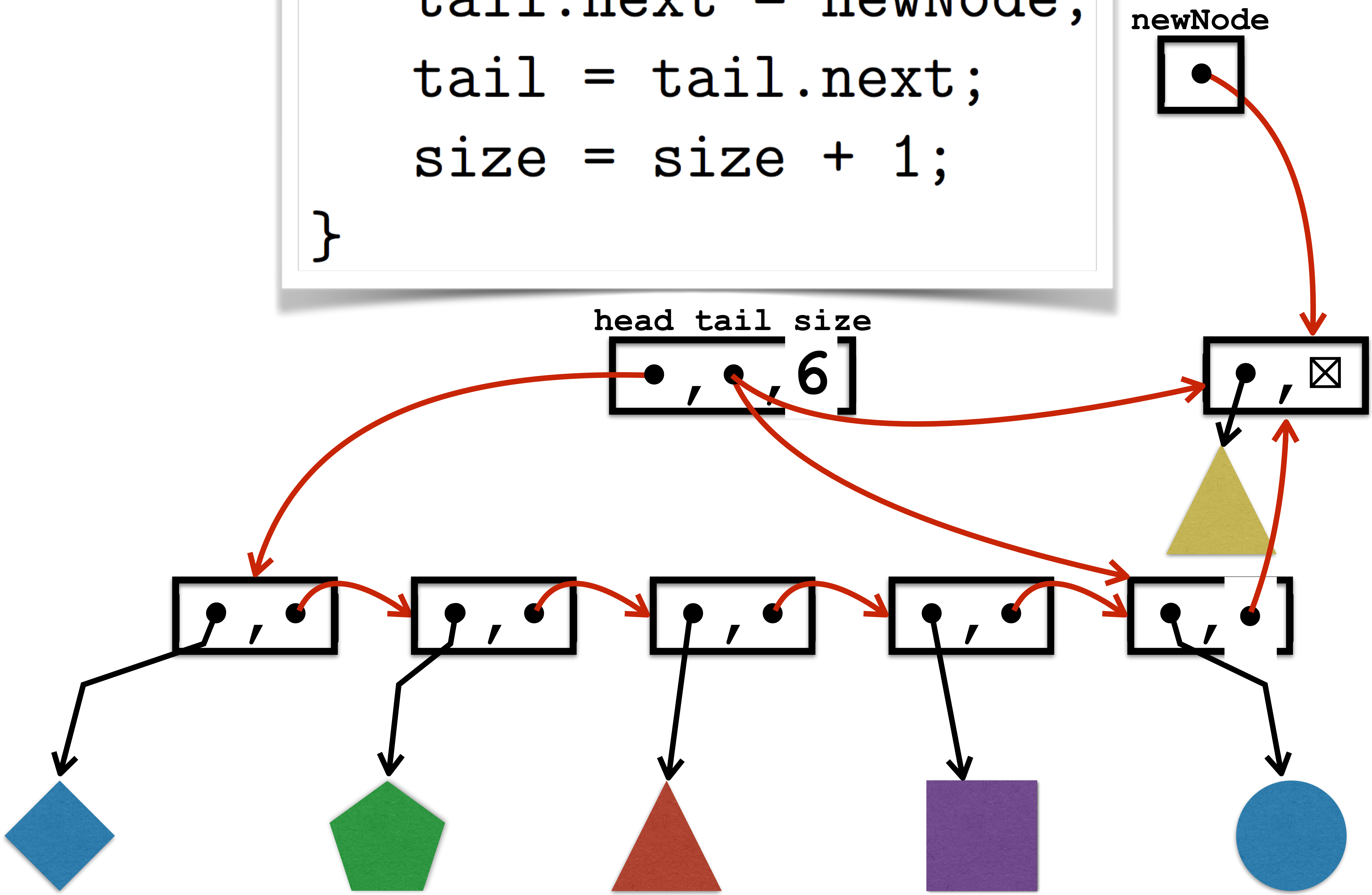
```
addFirst( newNode ){
    newNode.next = head;
    head = newNode;
    size = size + 1;
}
```



```
removeFirst(){
    tmp = head;
    head = head.next;
    tmp.next = null;
    size = size - 1;
}
```

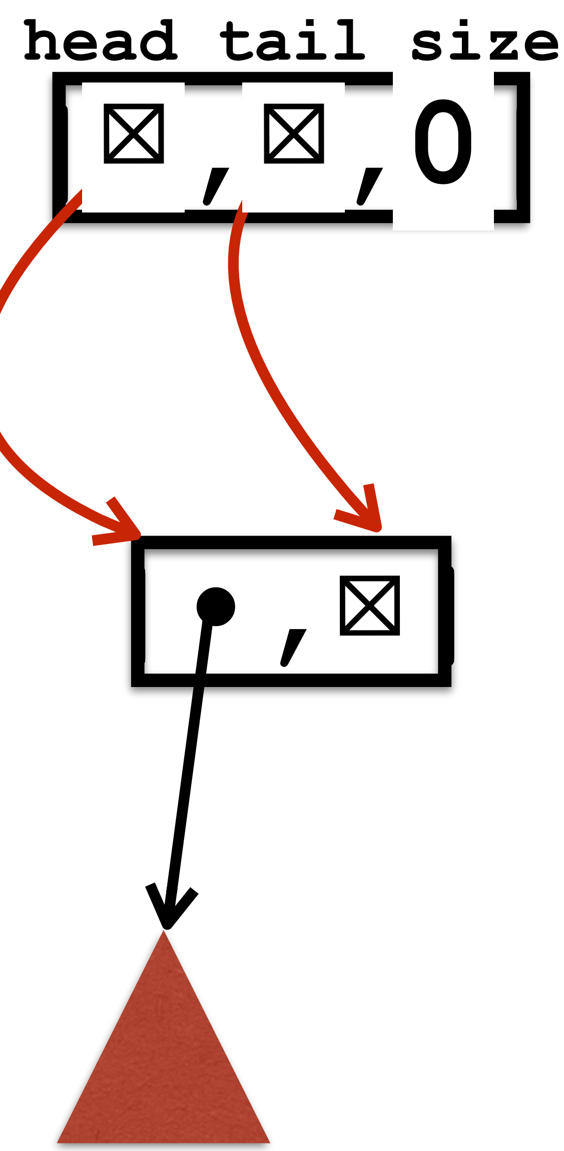


```
addLast( newNode ){
    tail.next = newNode;
    tail = tail.next;
    size = size + 1;
}
```

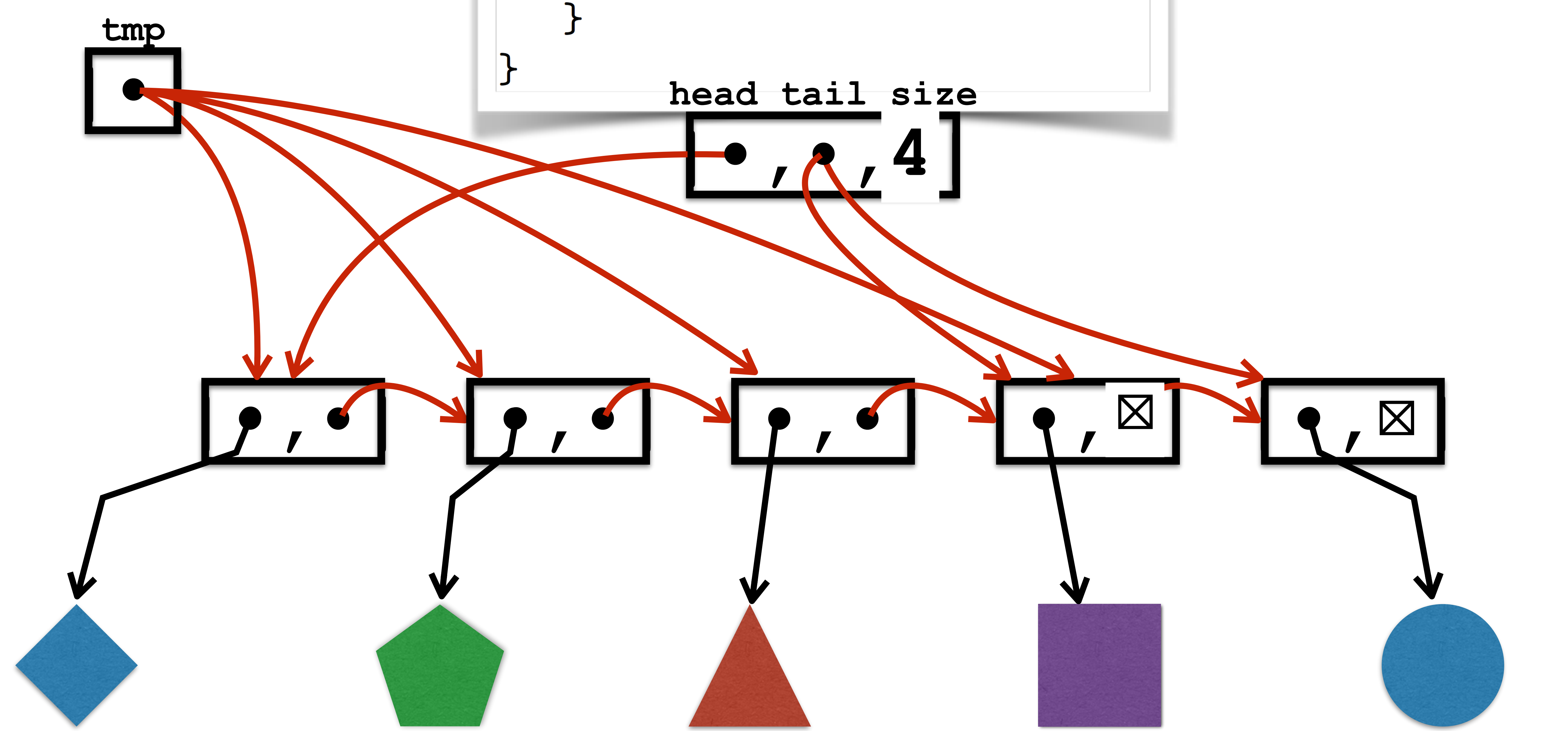


```
removeLast(){
   if (head == tail){
      head = null;
      tail = null;
      size = 0;
   }
   else{
     tmp = head;
     while (tmp.next != tail){
        tmp = tmp.next;
     }
     tmp.next = null;
     tail = tmp;
     size = size - 1;
   }
}
```

```
removeLast(){
    if (head == tail){
        head = null;
        tail = null;
        size = 0;
    }
```

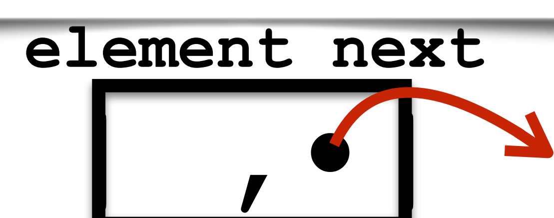


```
else{
    tmp = head;
    while (tmp.next != tail){
        tmp = tmp.next;
    }
    tmp.next = null;
    tail = tmp;
    size = size - 1;
  }
}
```



Java Generics

```
class  SNode<E>{
    E              element
    SNode<E>          next
      :
}

class  SLinkedList<E>{
    SNode<E>         head;
    SNode<E>         tail;
    int              size;
      :
}
```



Java Generics

```
class  SNode<E>{
    E               element
    SNode<E>           next
       :
}

class  SLinkedList<E>{
    SNode<E>           head;
    SNode<E>           tail;
    int                size;
       :
}
```

```
SLinkedList<Shape>    shapelist = new SLinkedList<Shape>();
SLinkedList<Student>  studentlist = new SLinkedList<Student>();
```

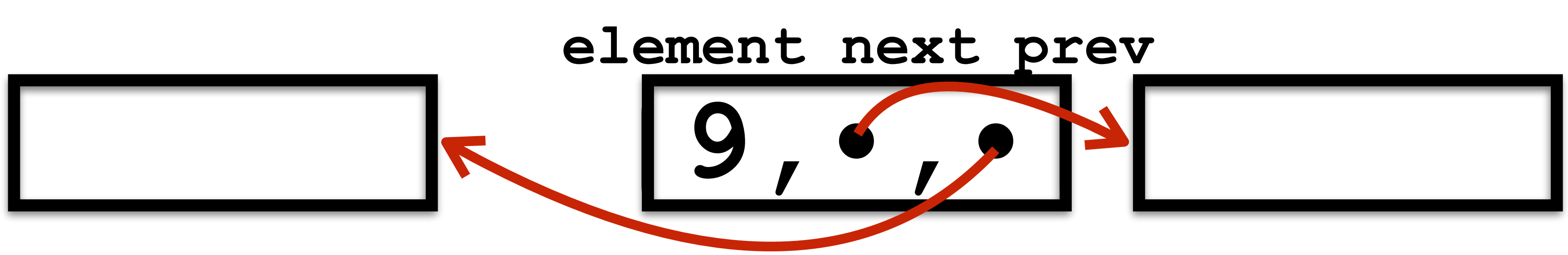
Java Generics

```
class  DNode<E>{
    E               element;
    DNode<E>           next;
    DNode<E>           prev;
       :
}

class  DLinkedList<E>{
    DNode<E>           head;
    DNode<E>           tail;
    int                size;
       :
}
```

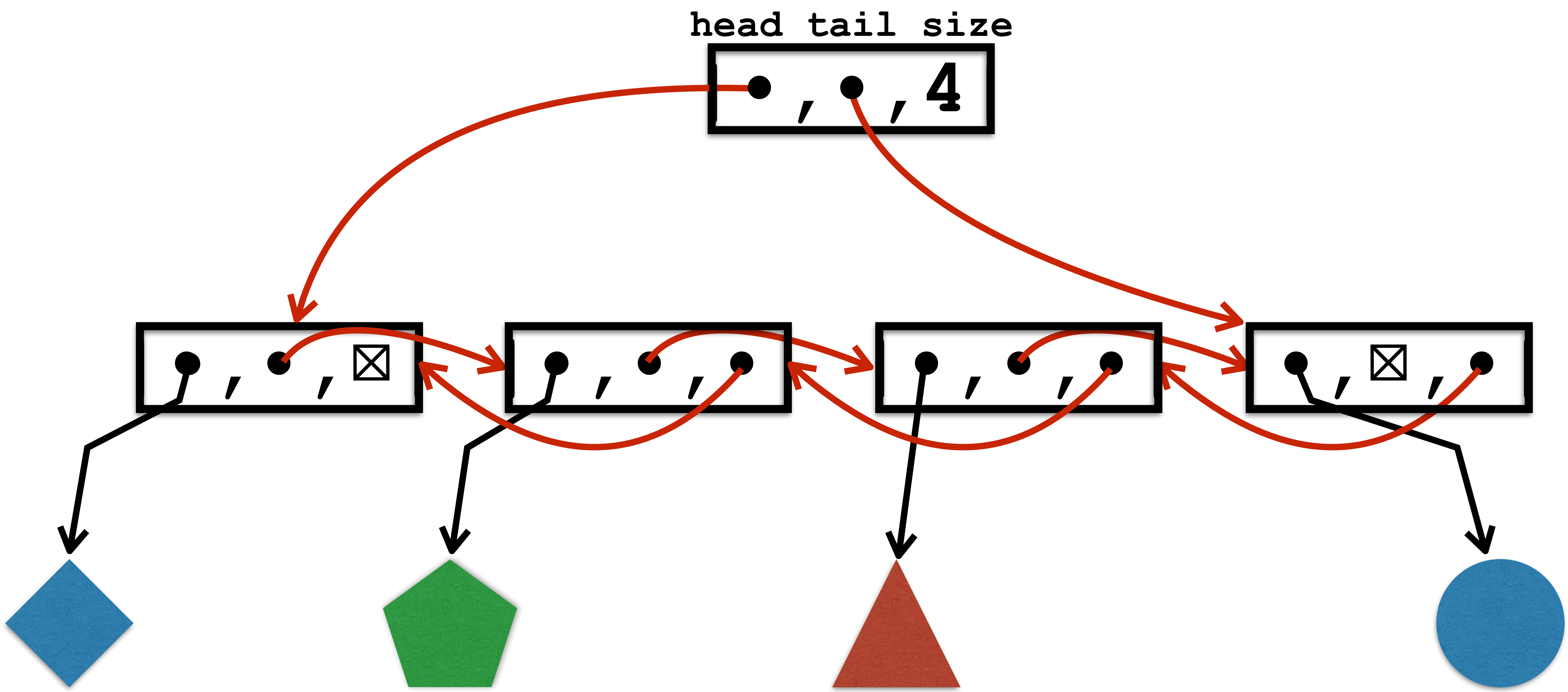
```
DLinkedList<Shape>     shapelist = new DLinkedList<Shape>();
DLinkedList<Student>   studentlist = new DLinkedList<Student>();
```

(Doubly) Linked List Node

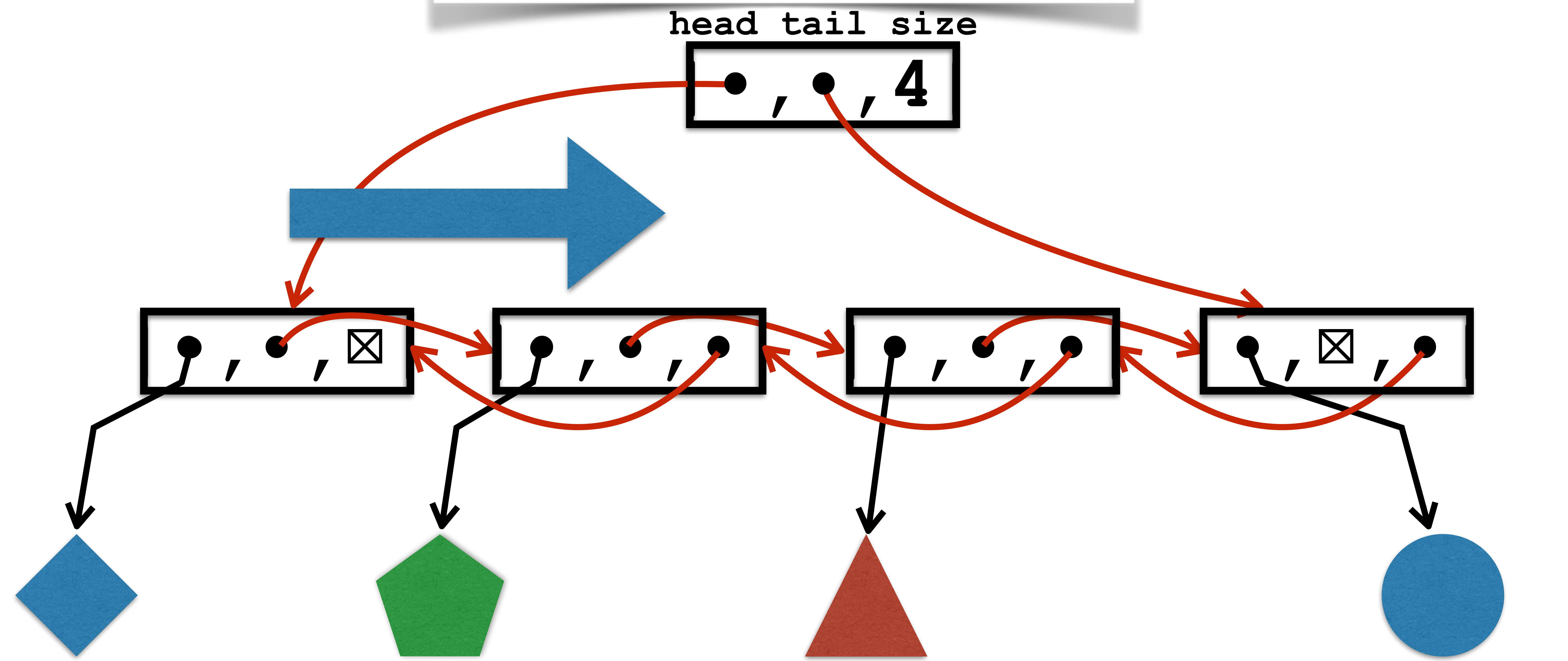
```
class  DNode<E>{
    E                element;
    DNode<E>            next;
    DNode<E>            prev;
        :
}
```



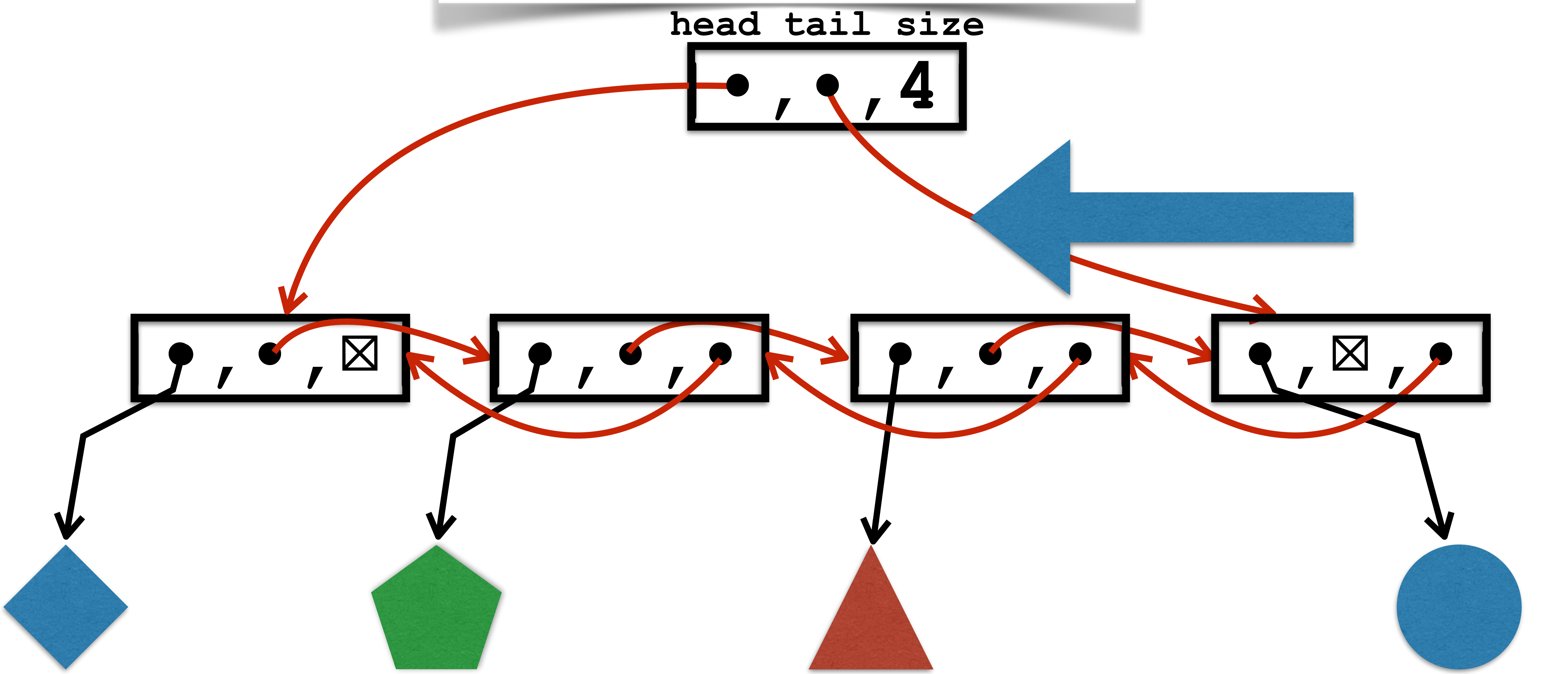
```
class  DLinkedList<E>{
    DNode<E>           head;
    DNode<E>           tail;
    int                size;
        :
}
```



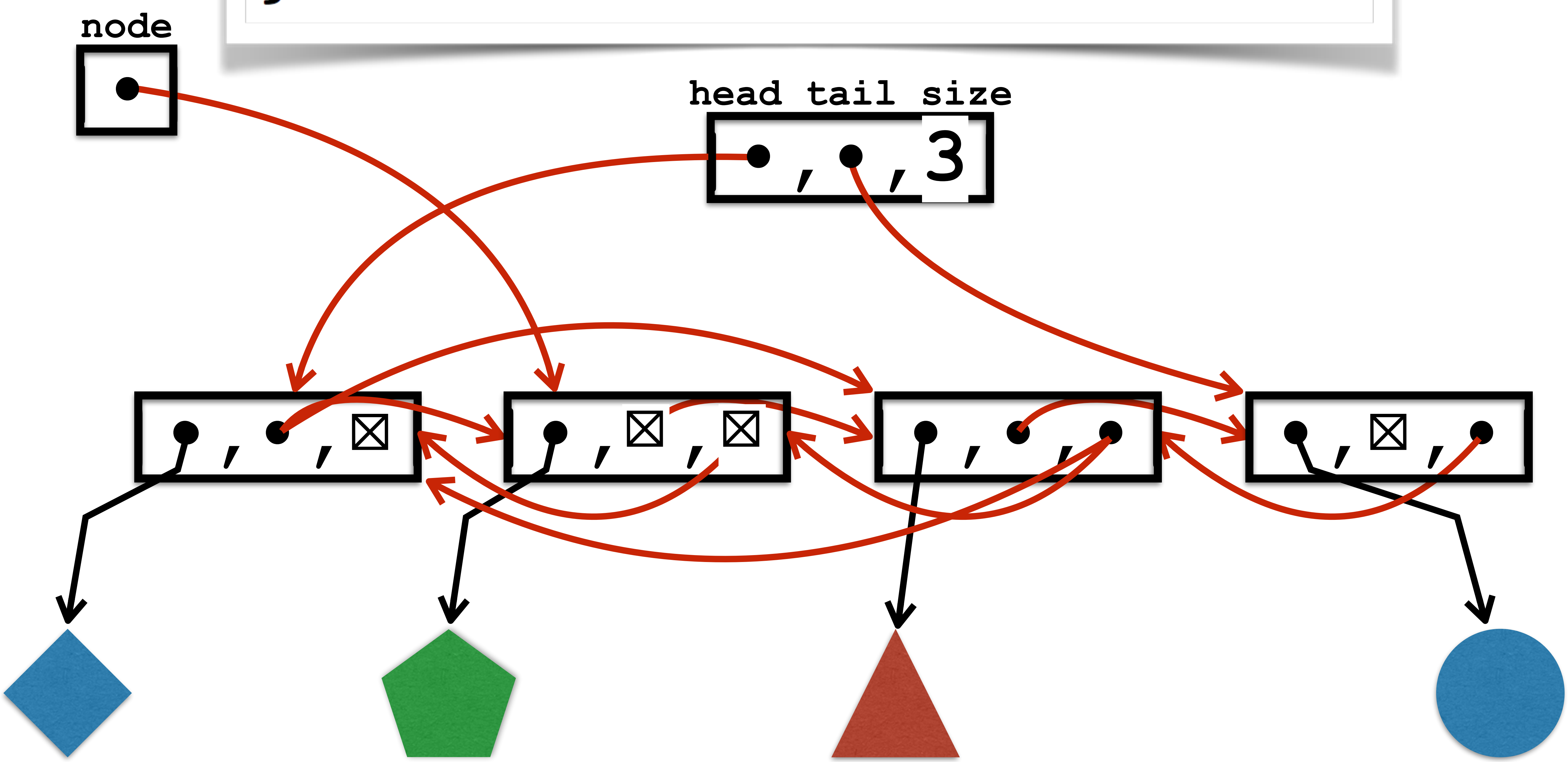
```
getNode(i){
    if (i < size/2){
        tmp = head
        index = 0
        while (index < i){
            tmp = tmp.next
            index++
        }
```



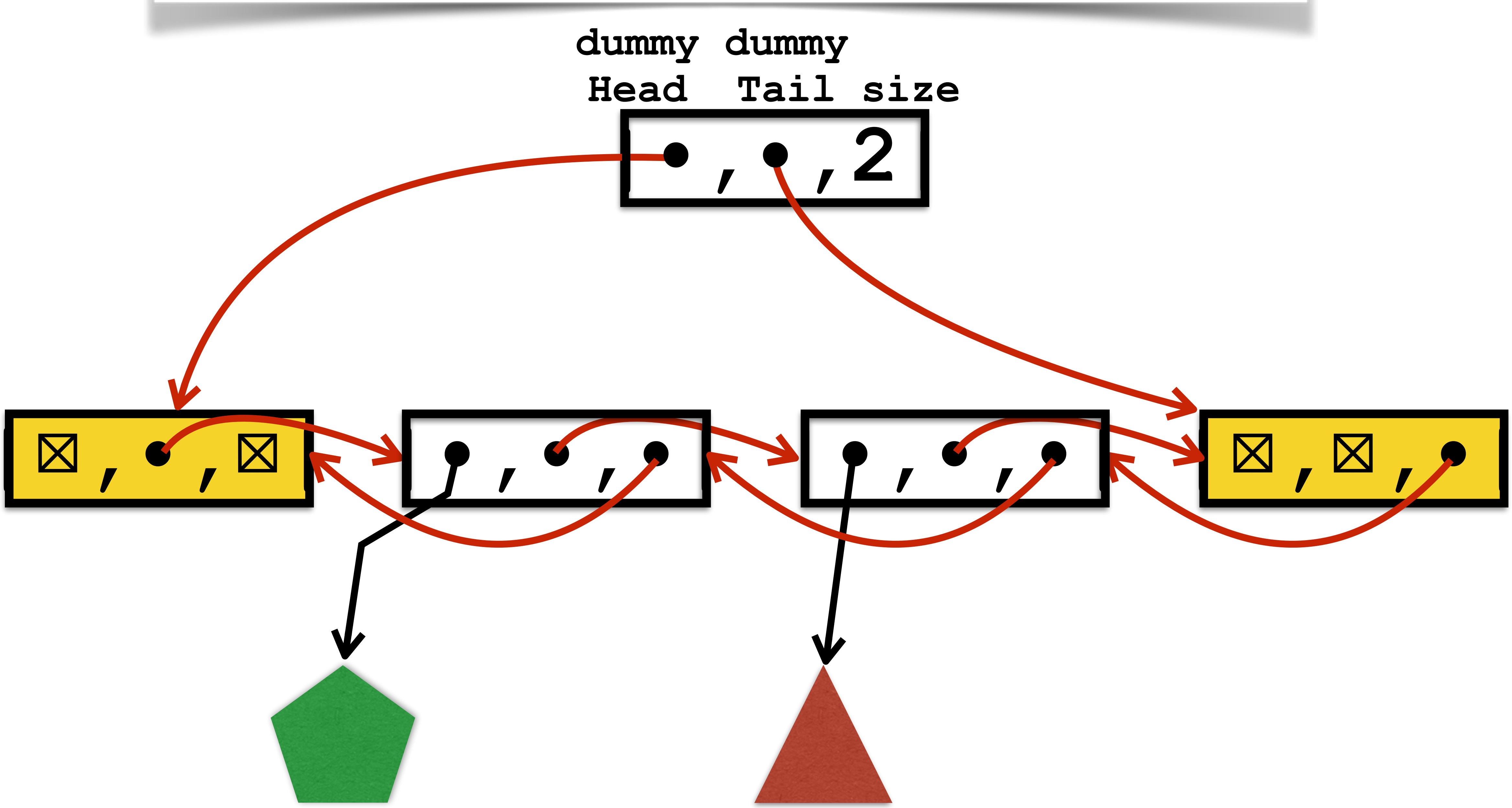
```
else{
    tmp = tail
    index = size - 1
    while (index > i){
        tmp = tmp.prev
        index--
    }
return tmp
```



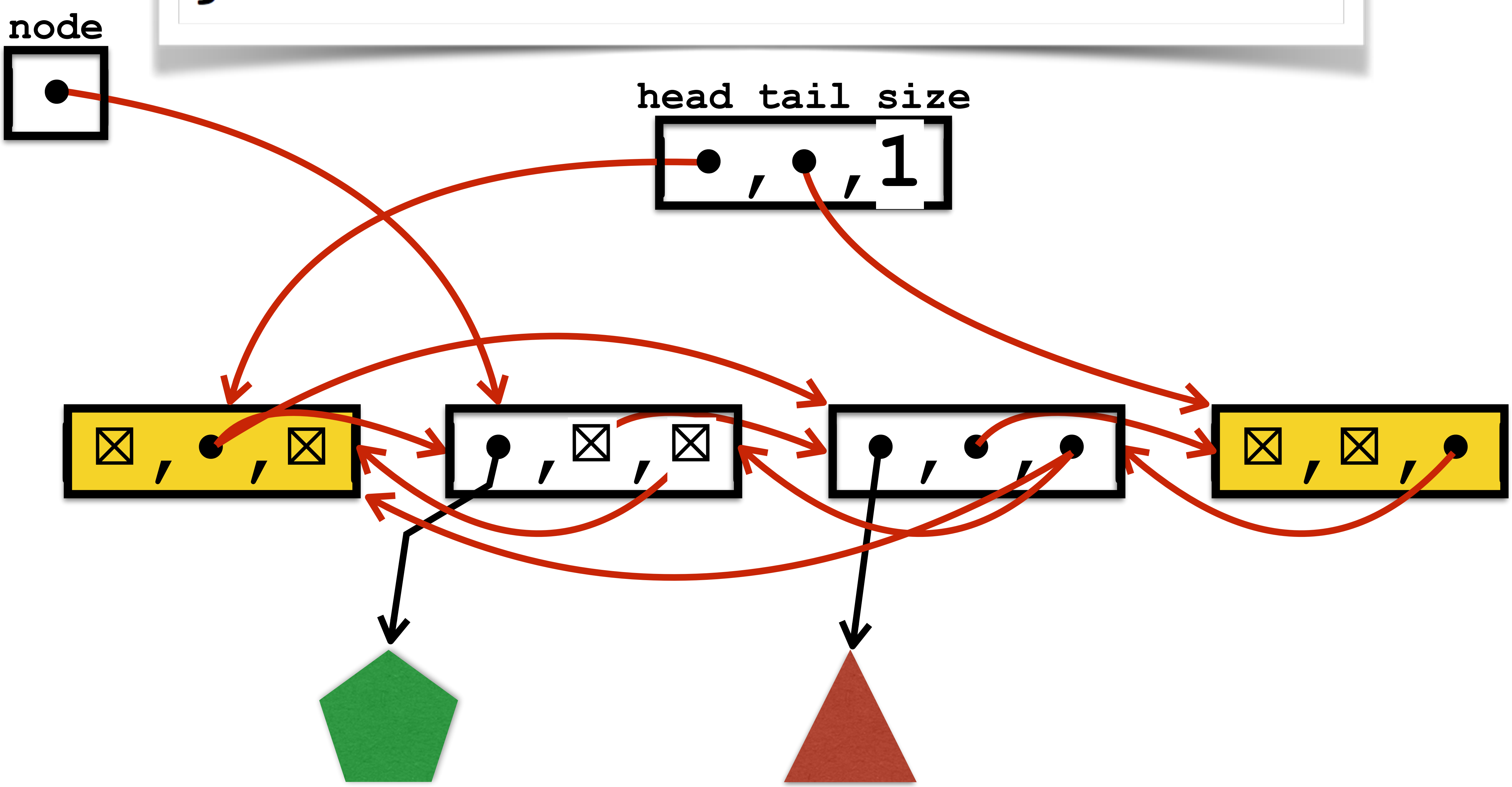
```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



```
class  DLinkedList<E>{
    DNode<E>            dummyHead;
    DNode<E>            dummyTail;
    int                 size;
    }
```



```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



Array vs Linked List

	array	singly linked list	doubly linked list
	------	----------	----------
addFirst	N	1	1
removeFirst	N	1	1
addLast	1	1	1
removeLast	1	N	1
getNode(i)	1	i	min(i, N/2 - i)

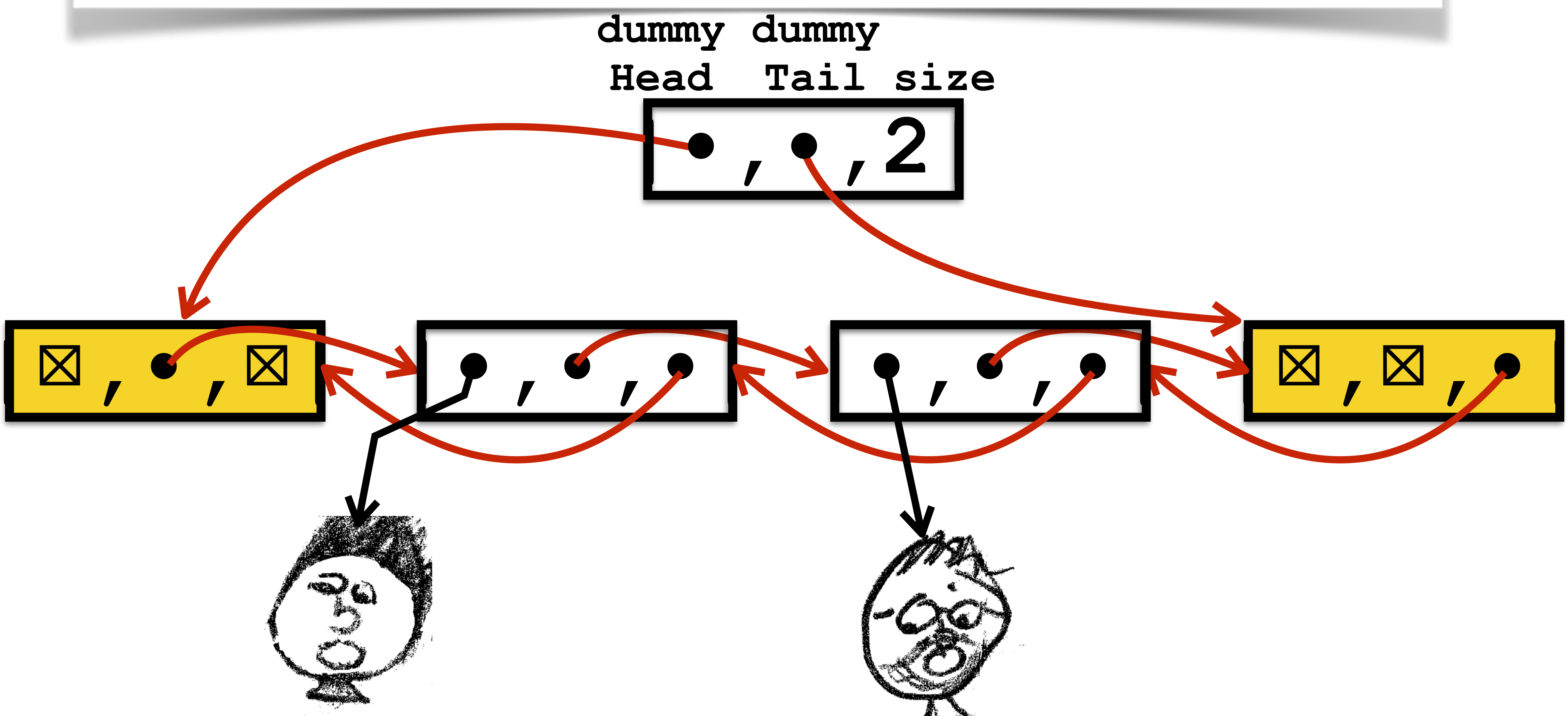
Linked List operations

```
add(i,element)    //  Inserts element into the i-th position
                  //  (and increments the indices of elements that were
                  //  previously at index i or up)
set(i,element)    //  Replaces the element in the i-th position
remove(i)         //  Removes the i-th element from list
get(i)            //  Returns the i-th element (but doesn't alter list)
clear()           //  Empties list.
isEmpty()         //  Returns true if empty, false if not empty.
size()            //  Returns number of elements in the list
  :
```

```
LinkedList<Student>
studentList  =  new  LinkedList<Student>();
```

Java LinkedList

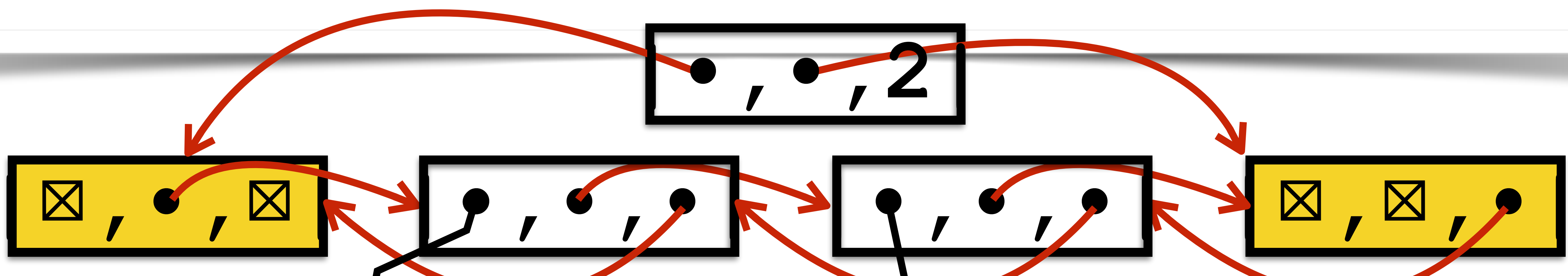
- implemented as doubly linked list (with dummies)
- Node class is private



Java LinkedList

	LinkedList
add(element)	1
add(i,element)	n
set(i,element)	n
remove(i)	n
get(i)	n
clear()	1
isEmpty()	1
size()	1

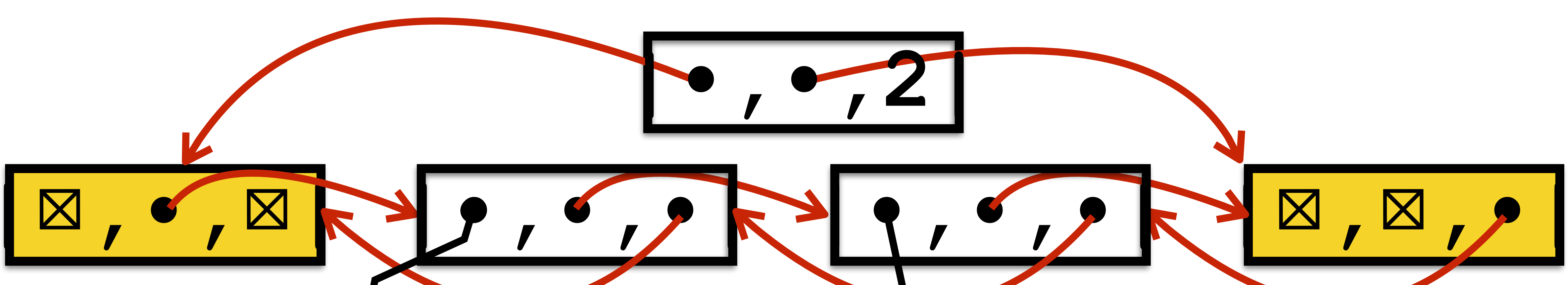
expensive



Java LinkedList

```
for (j = 1; j < n;   j++)
    print( studentList.get(j) )
```

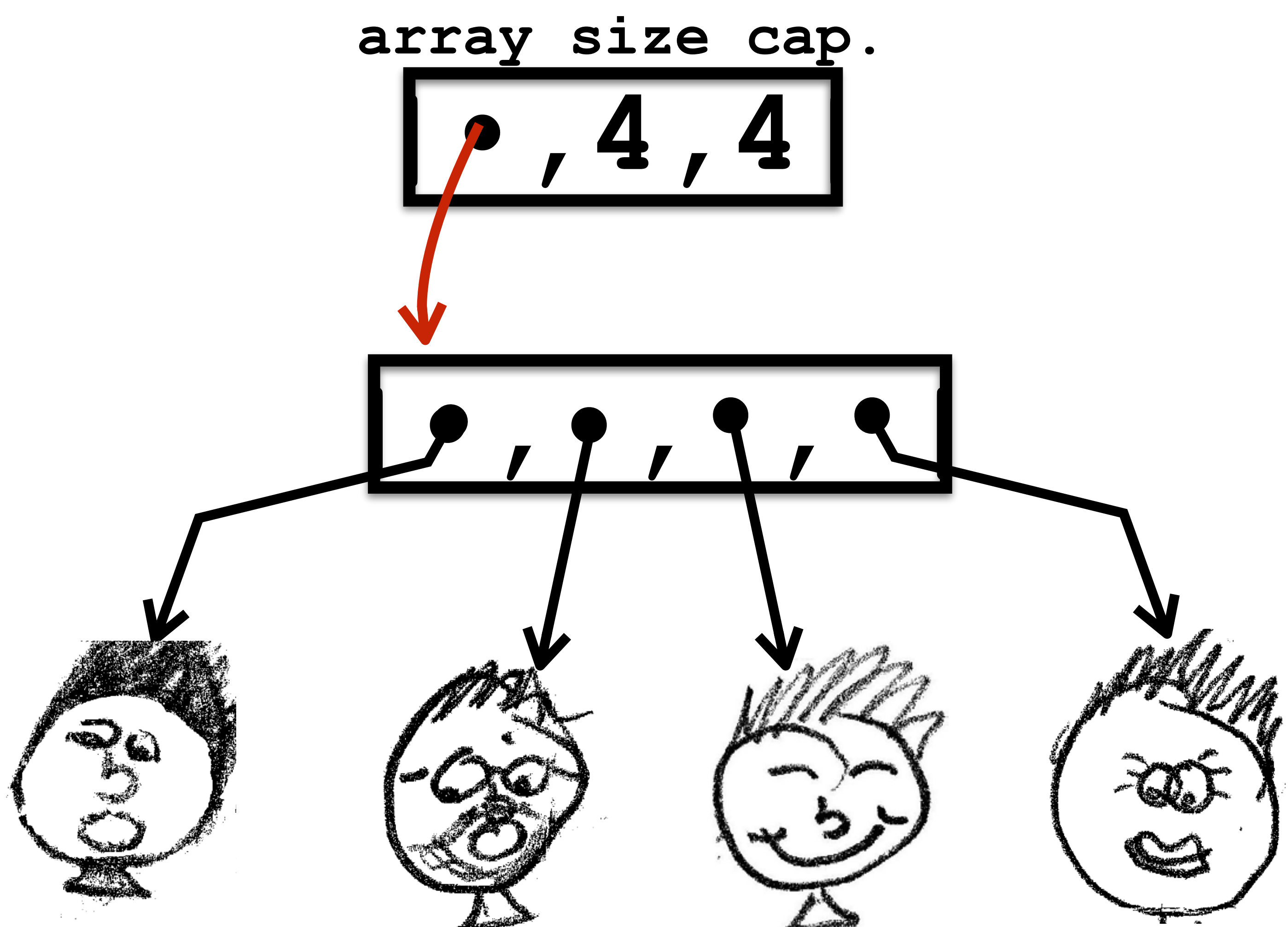
Time(n) **is** Ω(n^2)



Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



LinkedList vs ArrayList

	LinkedList	ArrayList
add(element)	1	1
add(i,element)	n	n
set(i,element)	n	1
remove(i)	n	n
get(i)	n	1
clear()	1	1
isEmpty()	1	1
size()	1	1

ADTs

- An Abstract Data Type is an abstraction of a data structure: no coding is involved.
- The ADT specifies:
  - what can be stored in it
  - what operations can be done on/by it.
- There are lots of formalized and standardized ADTs (in Java).

ADTs

- For example, if we are going to model a bag of marbles as an ADT, we could specify that
  - this ADT stores marbles
  - this ADT supports putting in a marble and getting out a marble.
- In this course we are going to learn a lot of different standard ADTs. (stacks, queues, trees...)
- (A bag of marbles is not one of them.)

Stack

- A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "pushing" onto the stack.
- "Popping" off the stack is synonymous with removing an item.

Stack

- A stack is an ADT that supports two main methods:
  - push(*o*): Inserts object *o* onto top of stack
  - pop(): Removes the top object of stack and returns it; if the stack is empty then an error occurs.
- The following support methods should also be defined:
  - size(): returns the number of objects in stack
  - isEmpty(): returns a boolean indicating if stack is empty.
  - top(): returns the top object of the stack, without removing it; if the stack is empty then an error occurs.

Examples:

```
push(3)
push(6)
push(4)
push(1)
pop()
push(5)
pop()
pop()
```

			1		5		
		4	4	4	4	4	
	6	6	6	6	6	6	6
3	3	3	3	3	3	3	3
--	--	--	--	--	--	--	--

Examples:

```
3 + (4 - x)* 7 + (y - 2 * (2 + x)).
```

```
                        (
      (           (     (     (
---   ---   ---   ---   ---   ---   ---
```

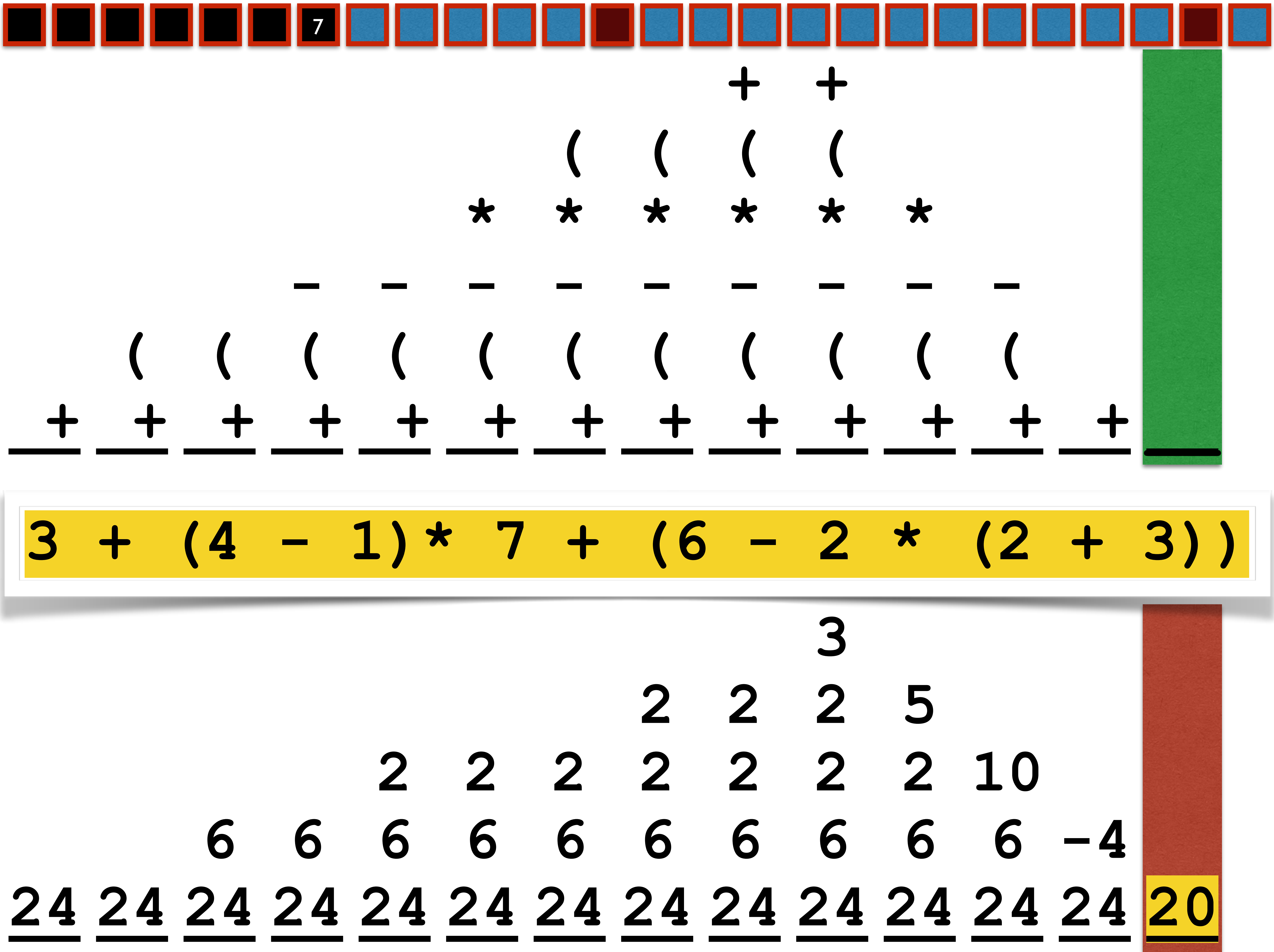
Examples:

```
( ( [ ] ) ) [ ] { [ ] }
```

			[									
		(	(	(						[		
	(	(	(	(	(		[		{	{	{	
_	_	_	_	_	_	_	_	_	_	_	_	_

Examples:

```
3 + (4 - 1)* 7 + (6 - 2 * (2 + 3))
```


7
3 + (4 - 1)* 7 + (6 - 2 * (2 + 3))
20

Processing arithmetics

```
t=gettoken()
while type(t)≠eol do
   if type(t)=number then
   if type(t)=operator then
   if t="(" then
   if t=")" then
   t=gettoken()

while not isemptyO() do
   op=popO()
   arg2=popA()
   arg1=popA()
   pushA(exec(arg1,op,arg2))
return popA()
```

Processing arithmetics

```
  if type(t)=number then pushA(t)

  if type(t)=operator then
   if prio(t)≤prio(topO())
    then op=popO()
         arg2=popA()
         arg1=popA()
         pushA(exec(arg1,op,arg2))
    pushO(t)
```

Processing arithmetics

```
if t="(" then pushO(t)

if t=")" then
    op=popO()
    while op≠"(" do
       arg2=popA()
       arg1=popA()
       pushA(exec(arg1,op,arg2))
       op=popO()
```

```
t=gettoken()
while type(t)≠eol do
  if type(t)=number then pushA(t)
  if type(t)=operator then
   if prio(t)≤prio(topO())
    then op=popO()
         arg2=popA()
         arg1=popA()
         pushA(exec(arg1,op,arg2))
    pushO(t)
  if t="(" then pushO(t)
  if t=")" then
     op=popO()
     while op≠"(" do
       arg2=popA()
       arg1=popA()
       pushA(exec(arg1,op,arg2))
       op=popO()
  t=gettoken()
while not isemptyO() do
  op=popO()
  arg2=popA()
  arg1=popA()
  pushA(exec(arg1,op,arg2))
return popA()
```


TEXAS INSTRUMENTS
TI Programmable 57
2nd INV lnx CE CLR
log C.t
D.MS P→R sin cos tan
LRN x⇌t x² √x 1/x
Pause Ins Exc Prd π
SST STO RCL SUM yˣ
Nop Del I.xI
BST EE ( ) ÷
Dsz Deg
GTO 7 8 9 ×
x=t Rad
SBR 4 5 6 −
x≥t Grad
RST 1 2 3 +
Lbl Σ+ σ²
R/S 0 • +/− =

TI

vs

HP


HP 50g
Graphing Calculator
hp
EDIT CURS BIG EVAL FACTO SIMP
Y= WIN GRAPH 2D/3D TBLSET TABLE
F1 A F2 B F3 C F4 D F5 E F6 F
APPS MODE TOOL
VAR STO NXT
HIST EVAL SYMB
SIN COS TAN
EEX +/− 1/X
ALPHA 7 8 9 ×
4 5 6 −
1 2 3 +
ON 0 • SPC ENTER
CANCEL

Examples:

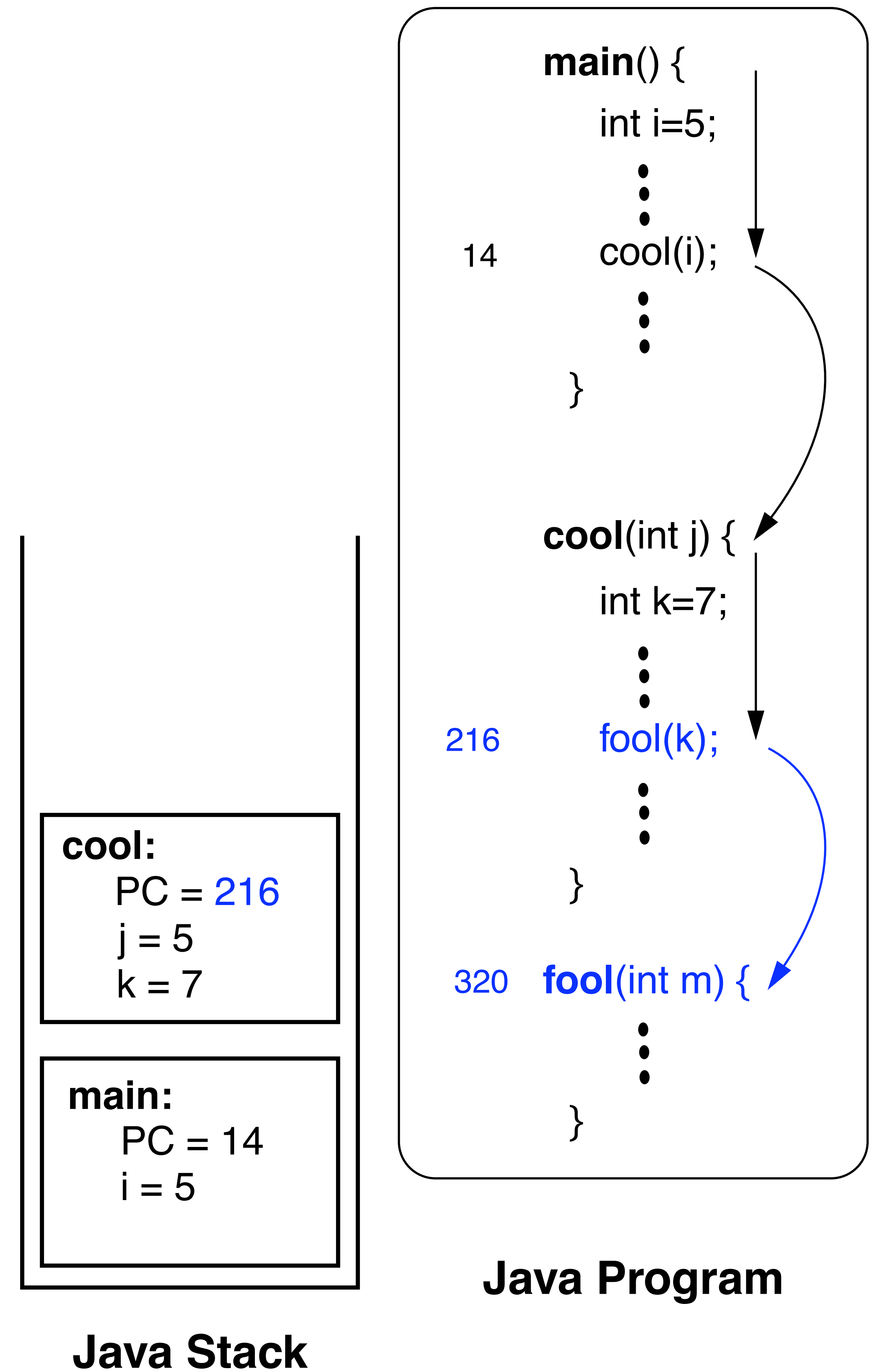
```
3 + (4 - 1)* 7 + (6 - 2 * (2 + 3))
```

```
3 4 1 - 7 * + 6 2 2 3 + * - +
```

											3				
										2	2	5			
			1		7				2	2	2	2	10		
		4	4	3	3	21		6	6	6	6	6	6	-4	
	3	3	3	3	3	3	24	24	24	24	24	24	24	24	20

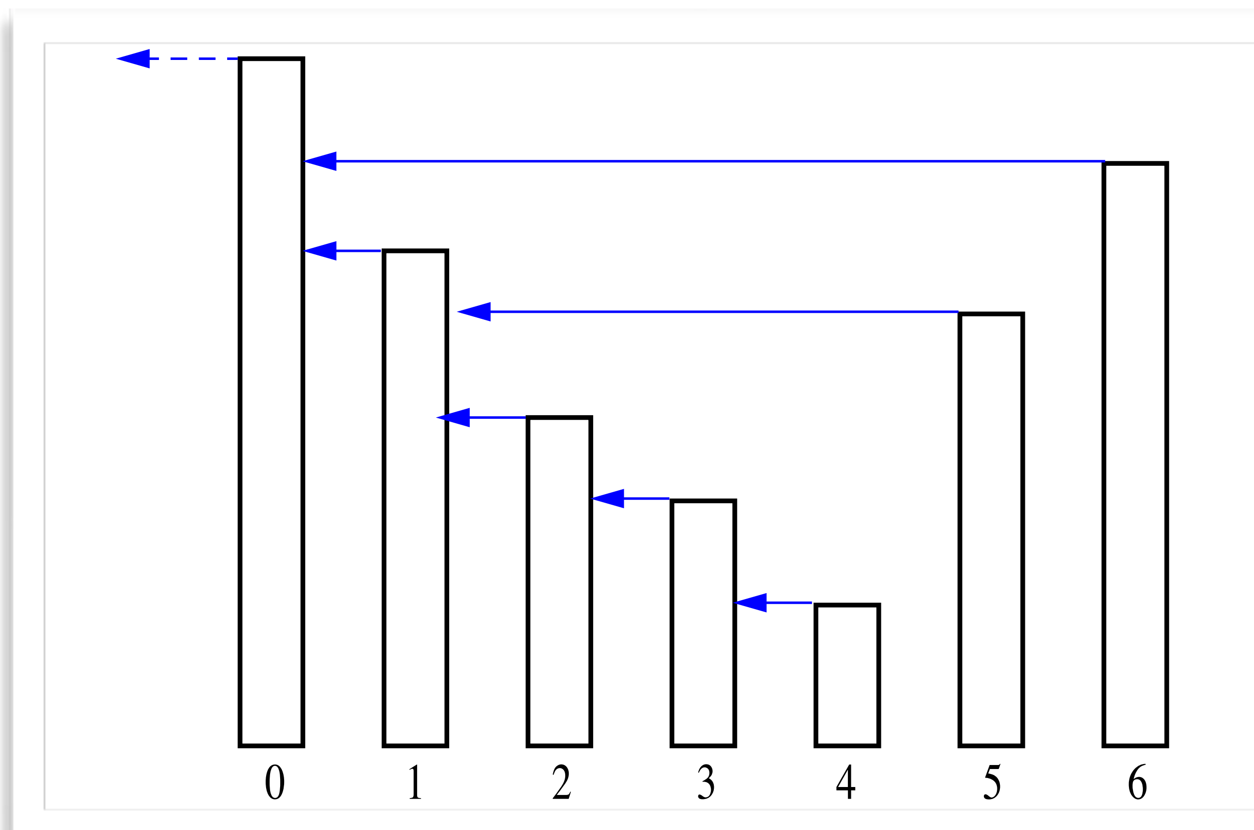
Stacks in the Java Virtual Machine

- Each process running in a Java program has its own Java Method Stack.
- Each time a method is called, it is pushed onto the stack.
- The choice of a stack for this operation allows Java to do several useful things:
  - Perform recursive method calls
  - Print stack traces to locate an error



- The code for our new algorithm:

```
Algorithm computeSpan2(P):
  Input: An n-element array P of numbers representing
            stock prices
  Output: An n-element array S of numbers such that
            S[i] is the span of the stock on day i
  Let D be an empty stack
  for i ← 0 to n − 1 do
    done ← false
    while not(D.isEmpty() or done) do
      if P[i] ≥ P[D.top()] then
        D.pop()
      else
        done ← true
      if D.isEmpty() then
        h ← −1
      else
        h ← D.top()
      S[i] ← i − h
      D.push(i)
  return S
```



Queue ADT

Queue

- A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the rear (enqueued) and removed from the front (dequeued).

Queue

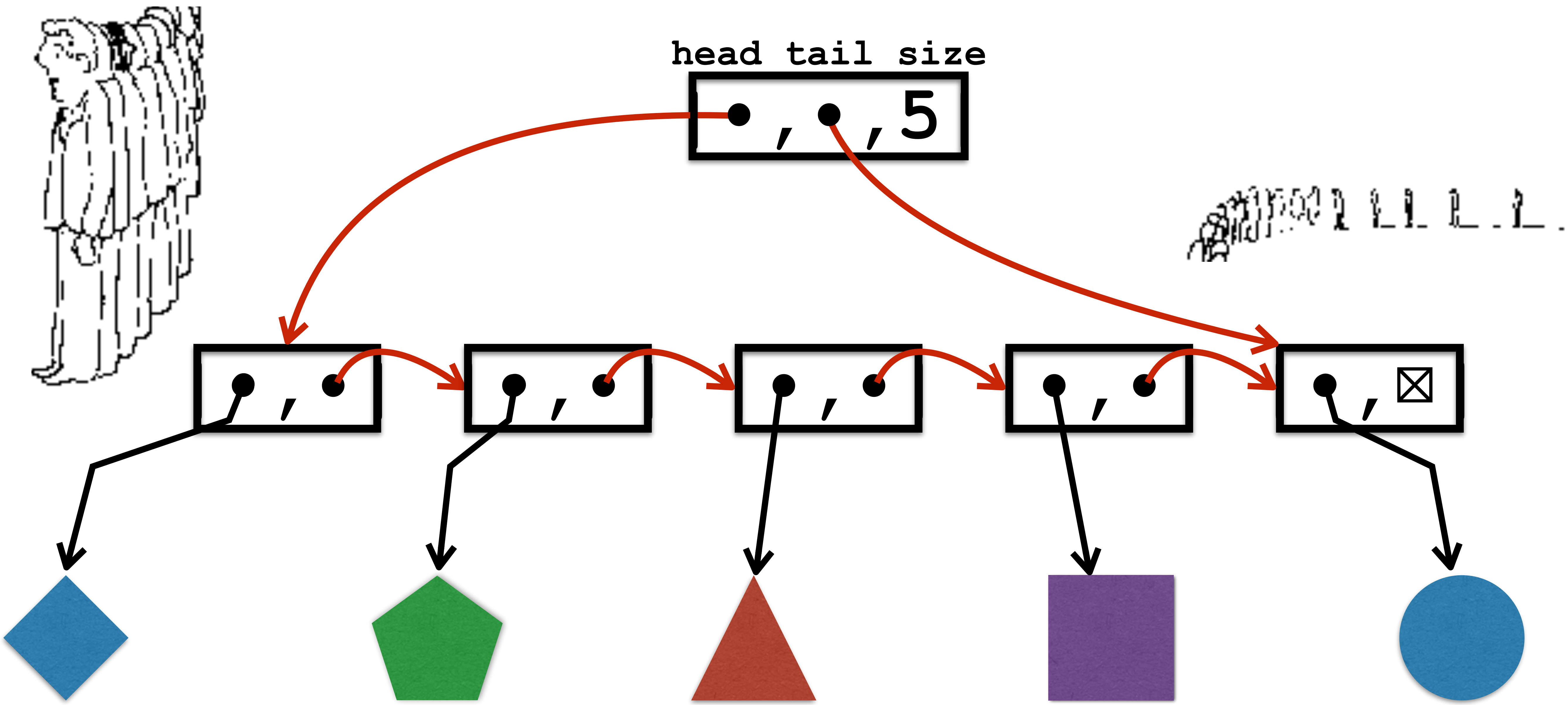
- The queue has two fundamental methods:
  - enqueue(*o*): Inserts object *o* at rear of the queue
  - dequeue(): Removes object from front of queue and returns it; an error occurs if queue is empty.
- These support methods should also be defined:
  - size(): Returns number of objects in the queue
  - isEmpty(): Returns a boolean value that indicates whether the queue is empty
  - front(): Returns, but not remove, the front object in the queue; an error occurs if queue is empty.

OPERATION	0123456	head	size
		0	0
add(a)	a	0	1
add(b)	ab	0	2
remove()	b	1	1
add(c)	bc	1	2
add(d)	bcd	1	3
add(e)	bcde	1	4
remove()	cde	2	3
add(f)	cdef	2	4
remove()	def	3	3
add(g)	defg	3	4

Queue as List

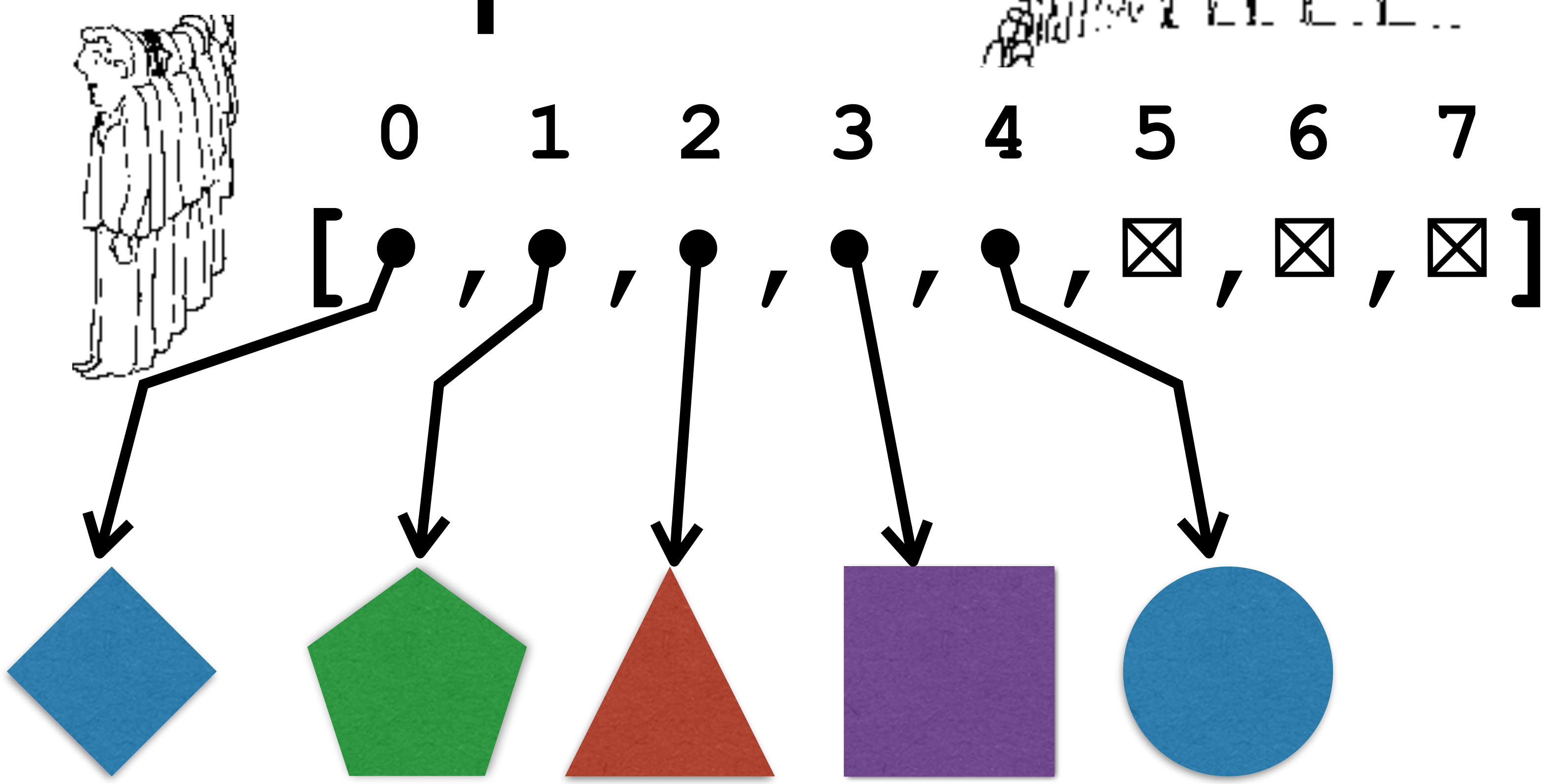
```
removeFirst(){
    tmp = head;
    head = head.next;
    tmp.next = null;
    size = size - 1;
}
```

```
addLast( newNode ){
    tail.next = newNode;
    tail = tail.next;
    size = size + 1;
}
```



Queue as Array

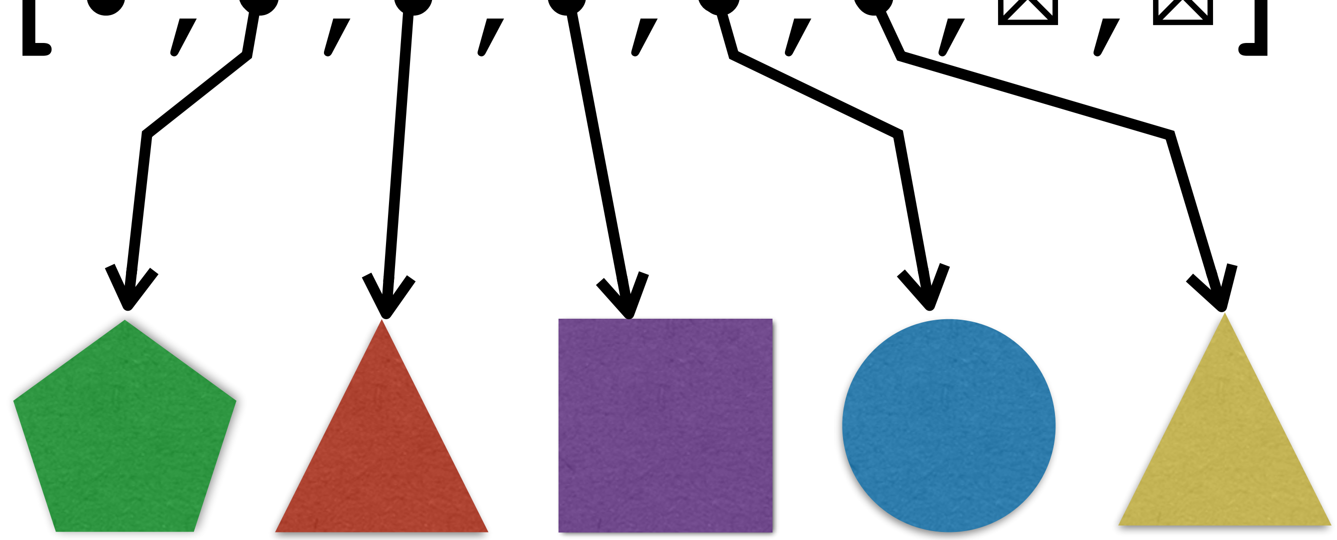
Array of shapes:



Size=5

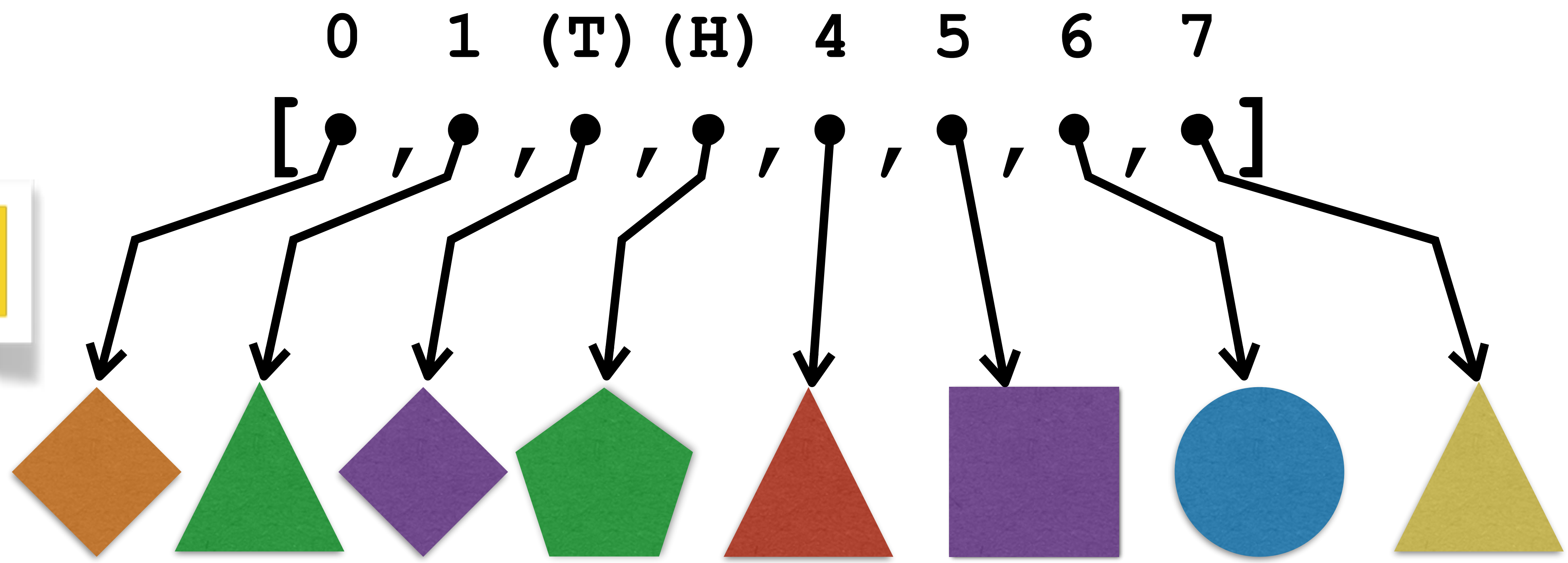
Queue as Array

```
0  (H)  2   3   4  (T)  6   7
[•,  •,  •,  •,  •,  •,  ⊠,  ⊠]
```



head=1, tail=5, (size=5)

Queue as Array



FULL !!

head=3, tail=2, (size=8)

Queue as Array

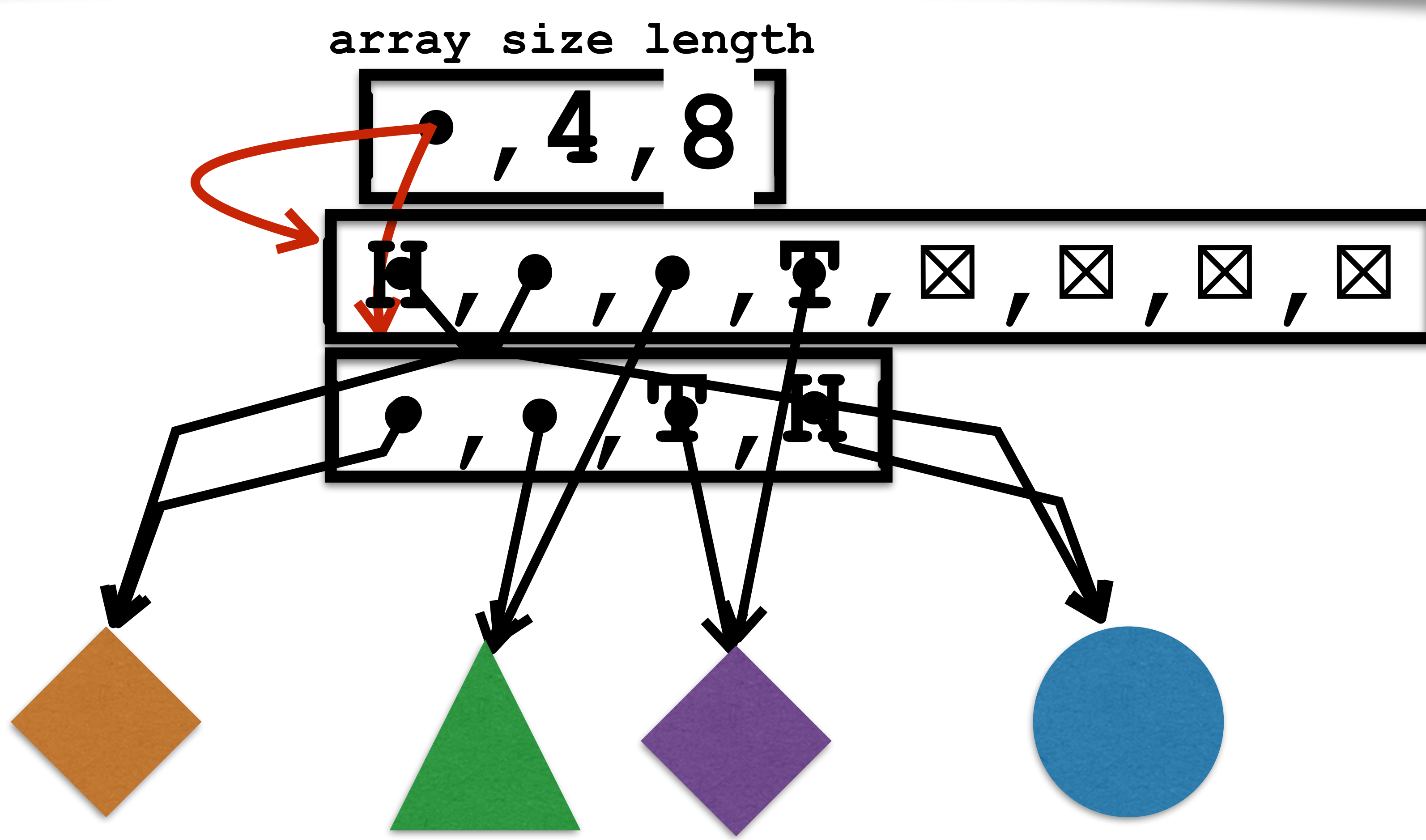
```
enqueue( element ){    //  array implementation
   if ( size == length)
      increase length of array  //  *** SEE BELOW  **

   a[ (head + size) % length ] = element
   size = size + 1
}
```

```
dequeue(){
   out = a[head]
   head = (head + 1) % length
   size = size - 1
   return out
}
```

Queue as Array

```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```



Running Times and Asymptotic Notation

Computational Tractability

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that tries every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

even worse : N ! for some problems

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants a > 0 and d > 0 such that on every input of size N, its running time is bounded by a N^d steps.

Def. An algorithm is poly-time if the above scaling property holds.

choose $C = 2^d$

Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Worst Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

Primality testing

simplex method
Unix grep

Why it matters ?

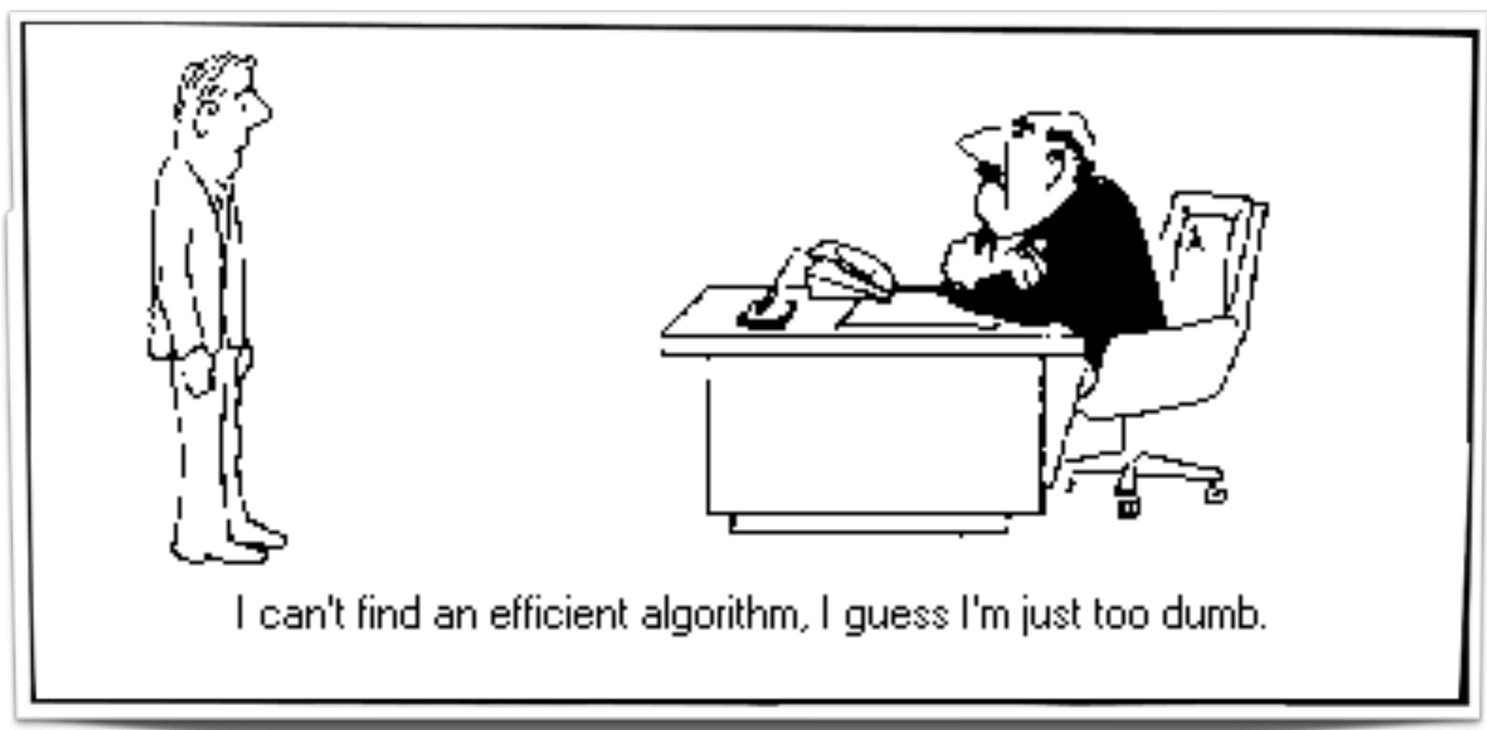
Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Note: age of Universe ~ 10^{10} years…

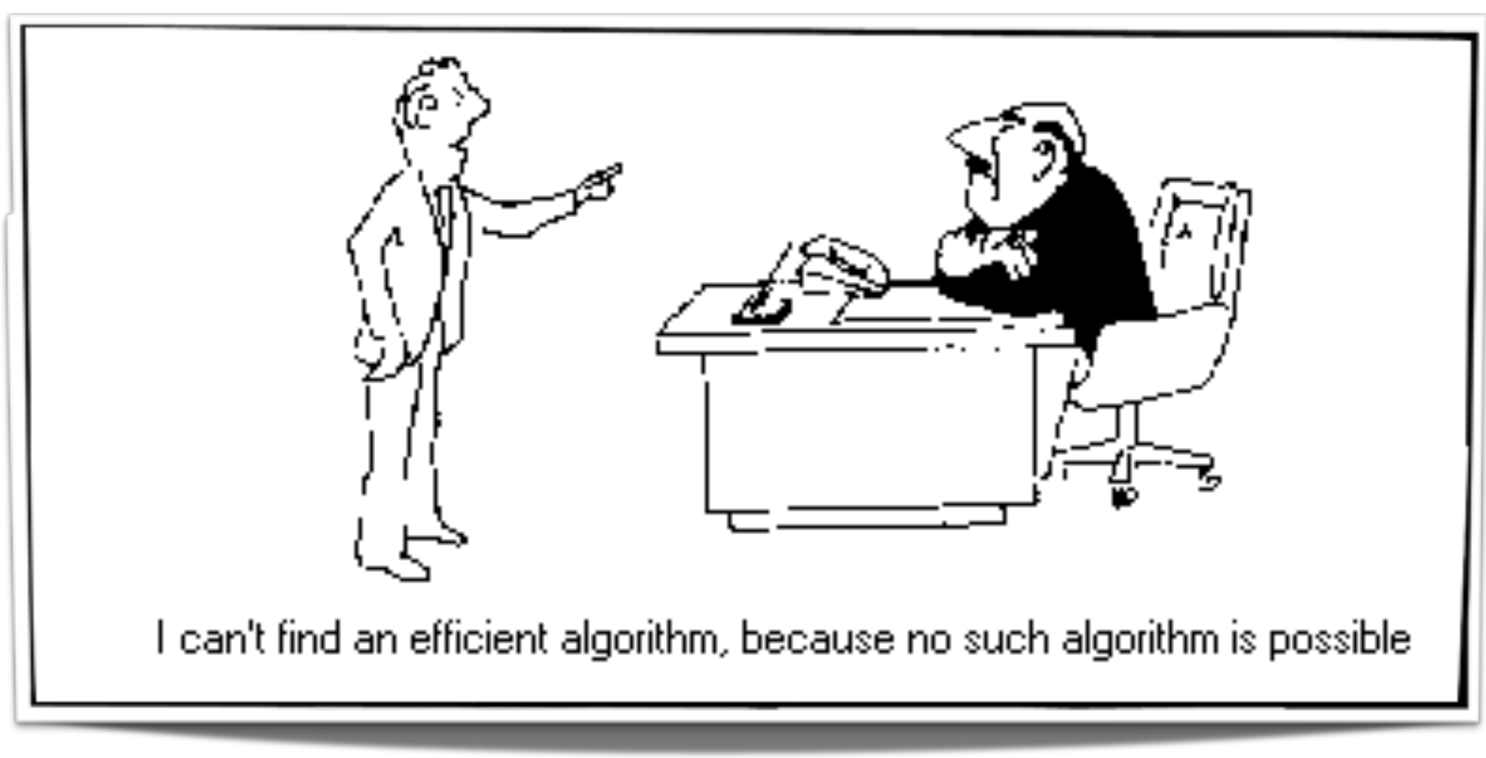
Computer Science Approach to problem solving

- If my boss / supervisor / teacher formulates a problem to be solved urgently, can I write a program to efficiently solve this problem ???



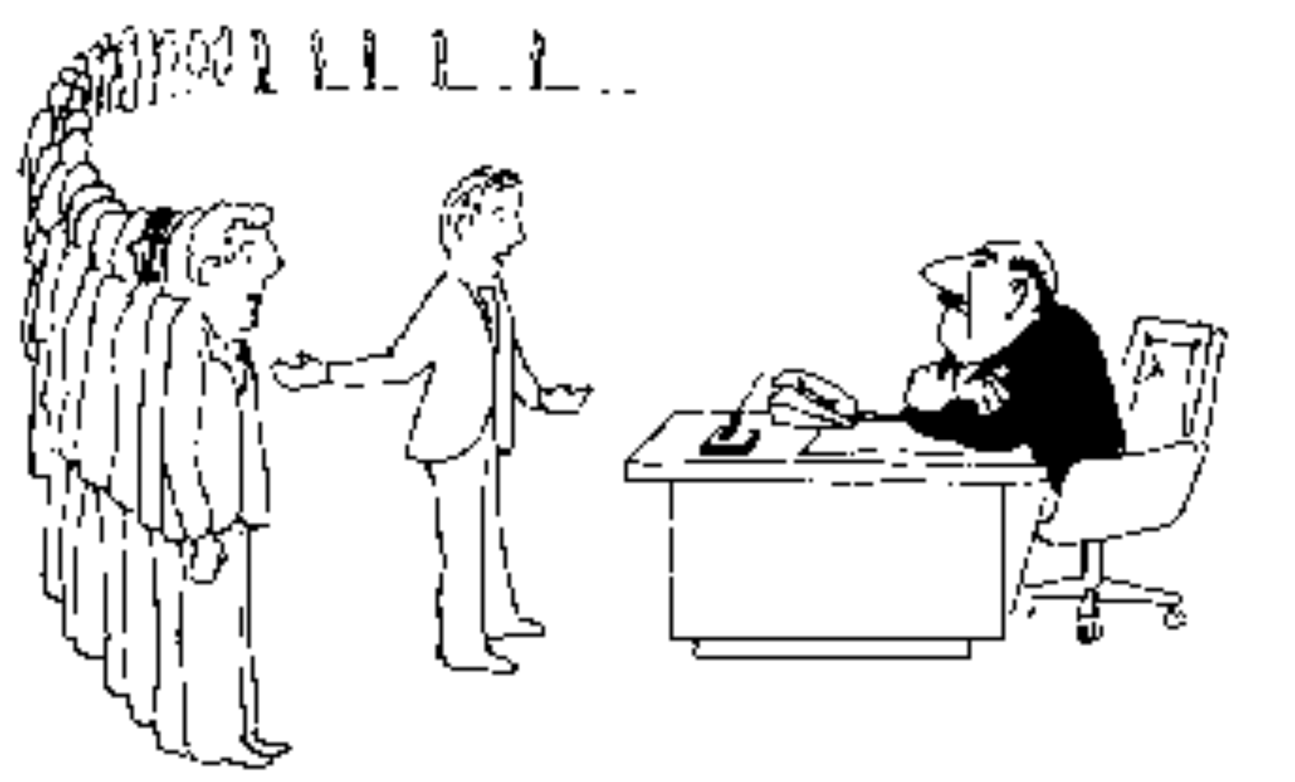
Computer Science Approach to problem solving

- Are there some problems that cannot be solved at all ? and, are there problems that cannot be solved efficiently ??



Computer Science Approach to problem solving

- If my boss / supervisor / teacher formulates a problem to be solved urgently, can I write a program to efficiently solve this problem ???



I can't find an efficient algorithm, but neither can all these famous people.

Asymptotic order of Growth and Notation

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Asymptotic order of Growth and Notation

Upper bounds. T(n) is O(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. T(n) is Ω(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- T(n) is $O(n^2)$ since there exists $c = 81$ and $n_0 = 1$ such that for all $n \geq 1$ we have $T(n) \leq 32n^2 + 17n^2 + 32n^2 = 81n^2$.
- T(n) is $\Omega(n^2)$ since there exists $c = 1$ and $n_0 = 0$ such that for all $n \geq 0$ we have $T(n) \geq n^2$.
- T(n) is not O(n) since for all $c > 0$ and $n_0 \geq 0$ there exists $n = \lceil c + 1/c + n_0 \rceil$ such that $T(n) > 32(c+1/c+n_0)^2 + 17(c+1/c+n_0) + 32 \geq c^2 + c \cdot n_0 + 32 \geq cn$.

Asymptotic Notation

Frequent Abuse of notation. $T(n) = O(f(n))$.

- Not transitive:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3)$ and $g(n) = O(n^3)$
  - but $f(n) \neq g(n)$ and $f(n) \neq O(g(n))$.
- Better notations: $T(n) \in O(f(n))$, $T(n)$ **is** $O(f(n))$.

Meaningless statement. "Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons."

- Statement doesn't "type-check".
- The constant function $f(n)=1$ is $O(n \log n)$.
- Use Ω for lower bounds.

Frequently Used Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

can avoid specifying the base

Logarithms. For every $x > 0$, $\log n$ is $O(n^x)$.

log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$.

every exponential grows faster than every polynomial

Linear Time: O(n)

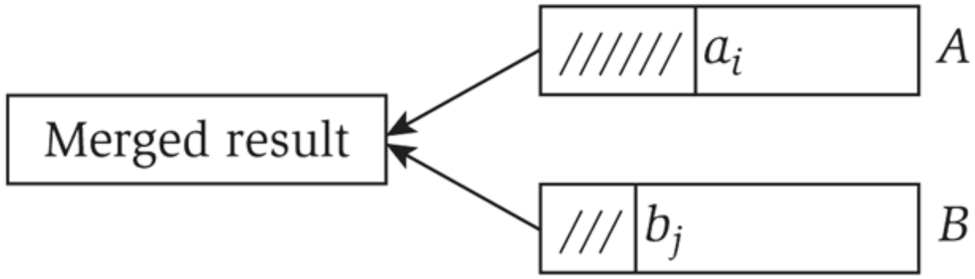
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a₁
for i = 2 to n {
   if (aᵢ > max)
      max ← aᵢ
}
```

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into a sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else          append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.

Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

Sorting. **Mergesort** and **Heapsort** are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      if (d < min)
         min ← d
   }
}
```

don't need to take square roots

Remark. This algorithm is $\Omega(n^2)$ and it seems inevitable in general, but this is just an illusion.

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pair of sets, determine if they are disjoint.

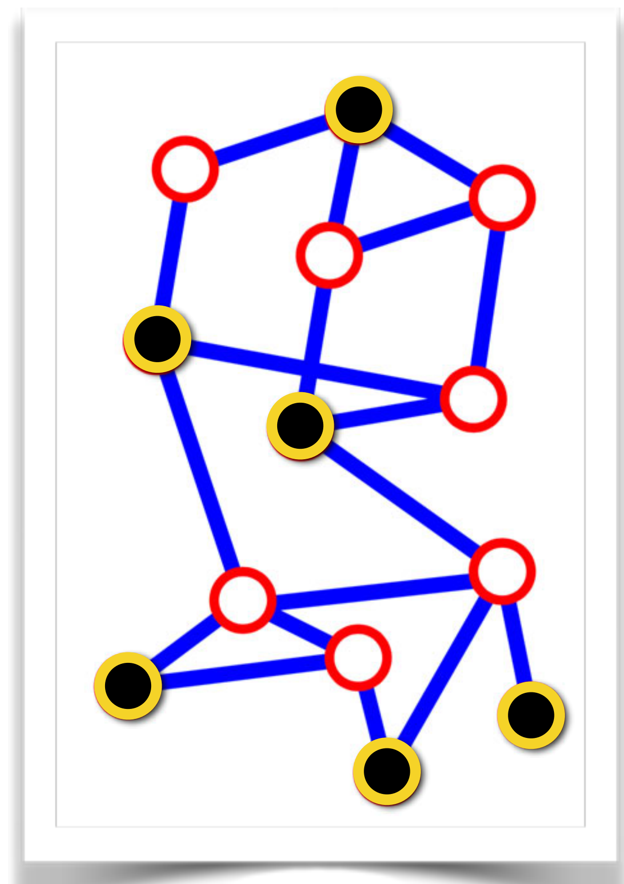
```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial Time: $O(n^k)$

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.



```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets : $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$
- $O(k^2\ n^k\ /\ k!)$ is $O(n^k)$.

poly-time for k=17,
but not practical

Exponential Time: $O(c^n)$

Independent set. Given a graph, what is the maximum size of an independent set?

$O(n^2\, 2^n)$ solution. Enumerate all subsets.

```
S* ← ∅
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

Induction Proofs

Predicate.

- P(n) : f(n) = some formula in n

Statement.
$\forall n \geq 1$, P(n) is true.

Proof.

- Base case: proof that P(1) is true.

- Induction step: $\forall n \geq 1$, $P(n) \implies P(n+1)$.

Let $n \geq 1$.
Assume for induction hypothesis that P(n) is true and prove P(n+1) is also true.

Iteration vs Recursion

- $f(n) = 1 + 2 + \ldots + n = \sum_{i=1}^{n} i$

```
f(n)
sum ← 0
for i = 2 to n {
    sum ← sum + i
}
return sum
```

- $f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1)+n & \text{if } n > 0 \end{cases}$

```
f(n)
if n = 0 { return 0 }
else { return f(n-1)+n }
```

Generalized Induction Proofs

Predicate.

- P(n) : f(n) = some formula in n

Statement.

For all $n \geq 1$, P(n) is true.

Proof.

- Base case: proof that P(1) is true.

- Induction step: let $n \geq 1$. Assume for induction hypothesis that P(1)…P(n) are all true. We show P(n+1) is also true.

Recursion:
Fibonacci Sequence

- $\text{fib}(n) = \begin{cases} n & \text{if } n \leq 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1 \end{cases}$

  Fibonacci sequence:
  0,1,1,2,3,5,8,13,21,34,55,89,144,…

- NOT so easy to define iteratively…

Recursion vs Iteration

- $\text{fib}(n) = \begin{cases} n & \text{if } n \leq 1 \\ \text{fib}(n\text{-}1) + \text{fib}(n\text{-}2) & \text{if } n > 1 \end{cases}$

```
fib(n)
if n < 2 { return n }
else { return fib(n-1)+ fib(n-2) }
```

```
fib(n)
a ← 0
b ← 1
for i = 1 to n {
    b ← a + b
    a ← b - a
}
return a
```

Weak Binet Formula

Statements.
For all $n \geq 1$, $\mathrm{fib}(n) \leq \varphi^n$ is true.
whenever $0 \leq \varphi^2 - \varphi - 1$ and $\varphi \geq 1$.

For all $n \geq 1$, $\mathrm{fib}(n) \geq \varphi^{n-2}$ is true.
whenever $0 \geq \varphi^2 - \varphi - 1$ and $\varphi \geq 1$.

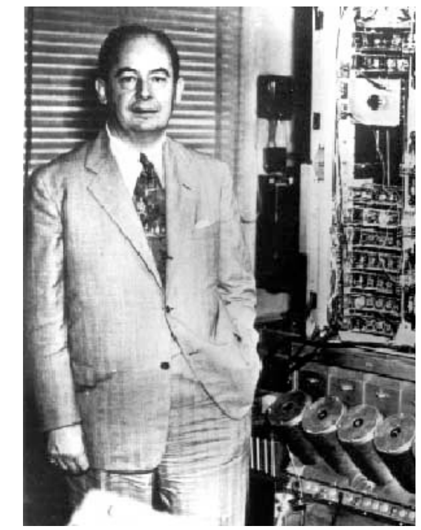
Therefore:
For all $n \geq 1$, $\varphi^n/\varphi^2 \leq \mathrm{fib}(n) \leq \varphi^n$ is true.
whenever $0 = \varphi^2 - \varphi - 1$ and $\varphi \geq 1$.
Only solution φ = golden ration = $(1+\sqrt{5})/2$.

$\mathrm{fib}(n)$ is $\boldsymbol{\theta}(\varphi^n)$.

Merge Sort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

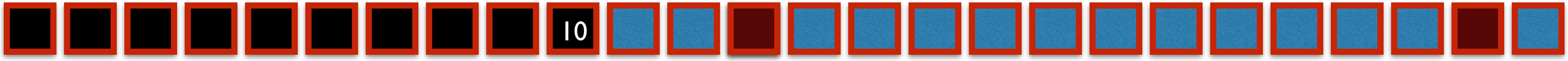


Jon von Neumann
(1945)

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R	I	T	H	M	S	divide	O(1)
A	G	L	O	R	H	I	M	S	T	sort	2T(n/2)

A	G	H	I	L	M	O	R	S	T	merge	O(n)

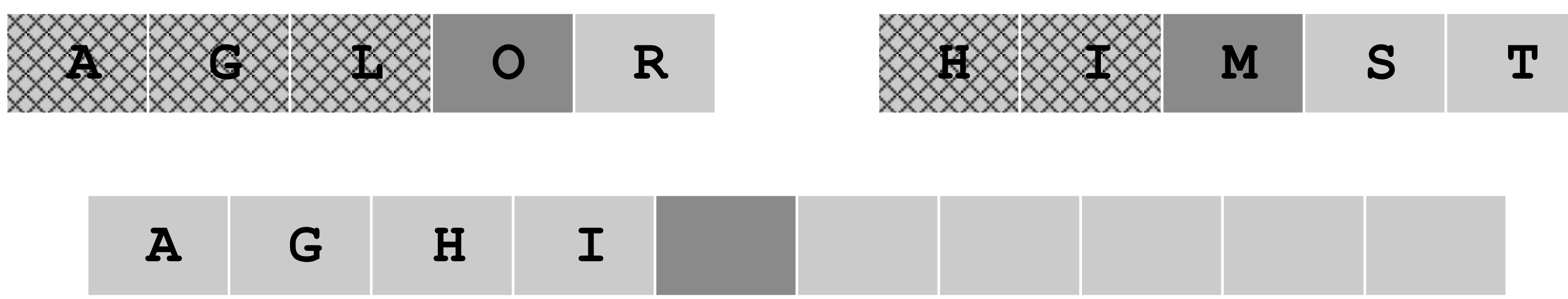


Merge

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.



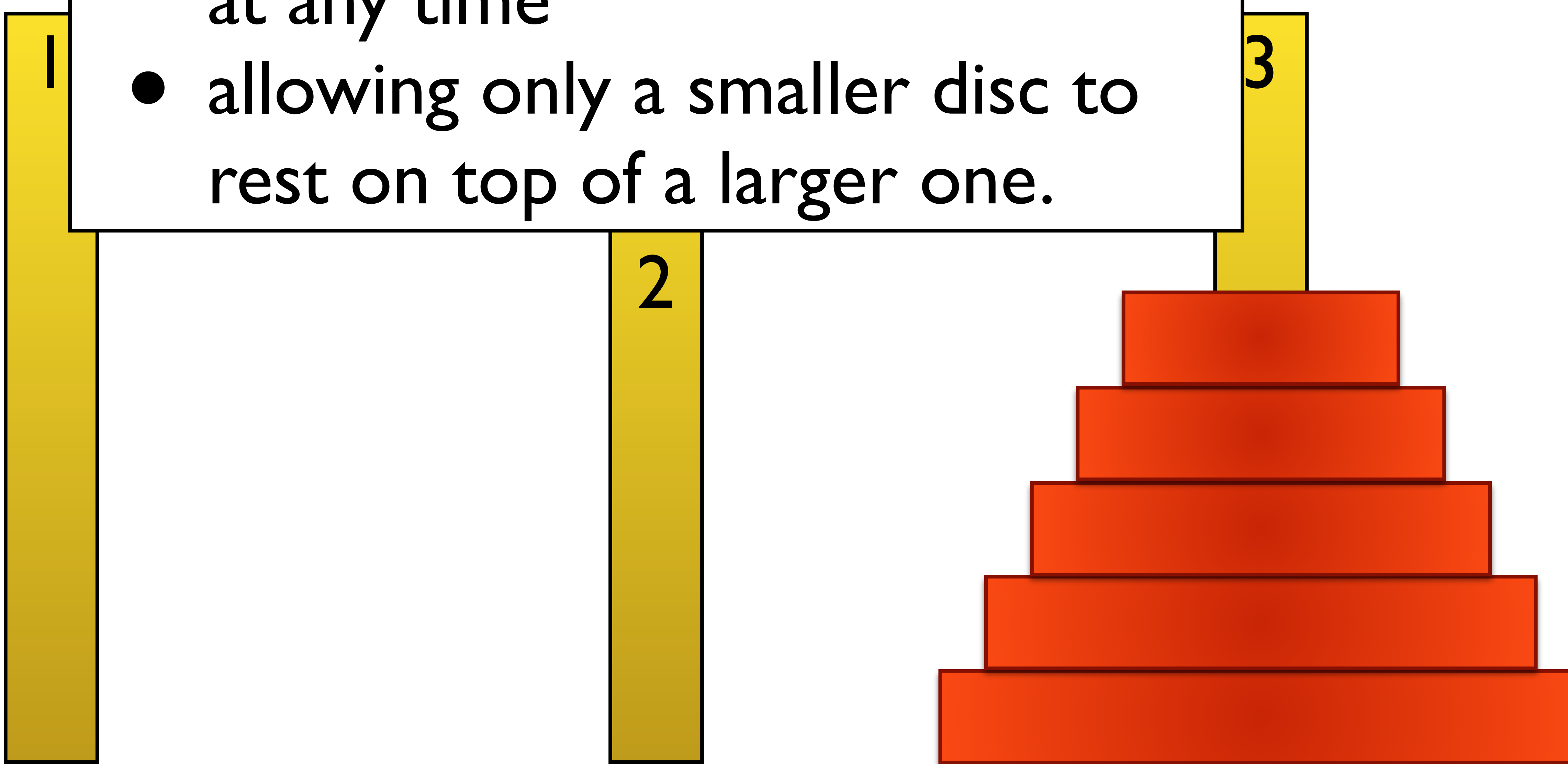
Challenge for the bored. In-place merge. [Kronrod, 1969]

using only a constant amount of extra storage

Tower of Hanoi

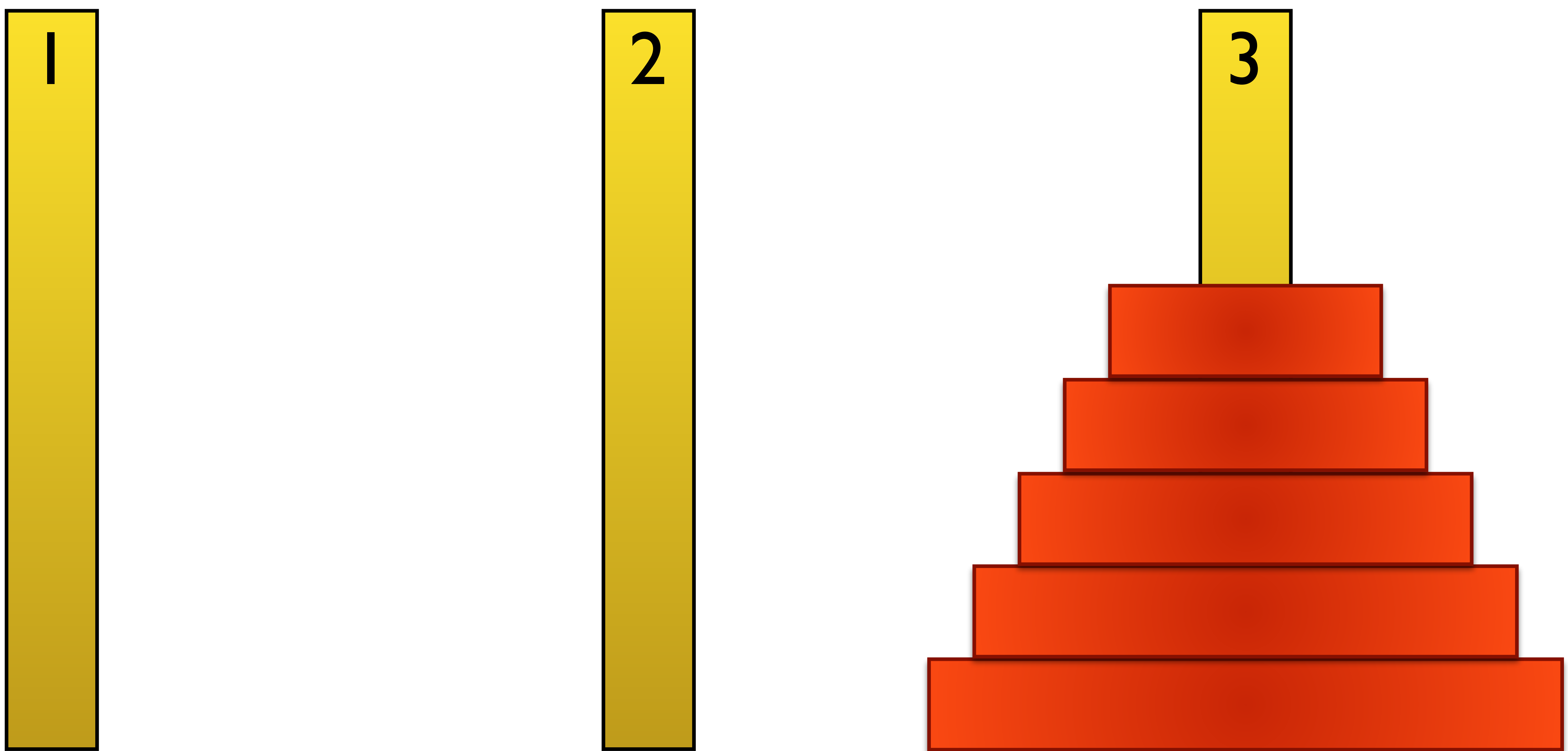
Goal: move the n discs from stack #3 to stack #2 while

- allowing only one disc removed at any time
- allowing only a smaller disc to rest on top of a larger one.



Hanoi(n,S3,S2,S1) // $n \geq 1$

if n>1 then Hanoi(n-1,S3,S1,S2)
move disc n from S3 to S2
if n>1 then Hanoi(n-1,S1,S2,S3)



Recurrence Relation

Def. T(n) = number of moves to Hanoi of n.

Hanoi recurrence.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n\text{-}1) + 1 & \text{if } n > 1 \end{cases}$$

Solution. T(n) is $O(2^n)$.

Assorted proofs. We describe several ways to prove this recurrence.

Telescoping Proof

Claim. If $T(n)$ satisfies this recurrence, then $T(n) = 2^n - 1$.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

Pf. For $n > 1$:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(\ 2T(n-2) + 1\ ) + 1 \\ &= 4T(n-2) + 2 + 1 \\ &= 4(\ 2T(n-3) + 1\ ) + 2 + 1 \\ &= 8T(n-3) + 4 + 2 + 1 \\ &\dots \\ &= 2^k T(n-k) + 2^{k-1} + \dots + 2 + 1 \\ &\dots \\ &= 2^{n-1} T(1) + 2^{n-2} + \dots + 2 + 1 \\ &= 2^n - 1. \end{aligned}$$

Induction Proof

Claim. If T(n) satisfies this recurrence, then $T(n) = 2^n - 1$.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

Pf. (by induction on n)

- Base case: $n = 1 = 2^1 - 1$.
- Inductive hypothesis: for $n \geq 1$, $T(n) = 2^n - 1$.
- Goal: show that $T(n+1) = 2^{n+1} - 1$.

$$\begin{aligned} T(n+1) &= 2T(n) + 1 && \text{by definition} \\ &= 2(\,2^n - 1\,) + 1 && \text{by I.H.} \\ &= 2^{n+1} - 2 + 1 \\ &= 2^{n+1} - 1. \end{aligned}$$

Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. T(n) is $O(n \log_2 n)$.

Assorted proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace ≤ with =.

Telescoping Proof

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. For n > 1:

$$\begin{aligned} \frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\ &= \frac{T(n/2)}{n/2} + 1 \\ &= \frac{T(n/4)}{n/4} + 1 + 1 \\ &\cdots \\ &= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

Induction Proof

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on k such that $n=2^k$)

- Base case: $n = 2^0 = 1$.
- Inductive hypothesis: $T(n) = T(2^k) = n \log_2 n$.
- Goal: show that $T(2n) = T(2^{k+1}) = 2n \log_2 (2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n\log_2 n + 2n \\ &= 2n\left(\log_2(2n) - 1\right) + 2n \\ &= 2n\log_2(2n) \end{aligned}$$

Generalized Induction Proof

Claim. If T(n) satisfies the following recurrence, then T(n) $\leq n \lceil \underset{\uparrow \atop \log_2 n}{\lg n} \rceil$.

$$
T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}
$$

Pf. (by induction on n)

- Base case: n = 1. T(1) = 0 = $1 \lceil \lg 1 \rceil$.
- Define $n_1 = \lfloor n/2 \rfloor$, $n_2 = \lceil n/2 \rceil$. (note $1 \leq n_1 < n$, $1 \leq n_2 < n$)
- Induction step: Let $n \geq 2$, assume true for 1, 2, ... , n–1.

$$
\begin{aligned}
T(n) &\leq T(n_1) + T(n_2) + n \\
&\leq n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\
&\leq n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\
&= n \lceil \lg n_2 \rceil + n \\
&\leq n(\lceil \lg n \rceil - 1) + n \\
&= n \lceil \lg n \rceil
\end{aligned}
$$

$$
\begin{aligned}
n_2 &= \lceil n/2 \rceil \\
&\leq \left\lceil 2^{\lceil \lg n \rceil} / 2 \right\rceil \\
&= 2^{\lceil \lg n \rceil} / 2 \\
\Rightarrow \lg n_2 &\leq \lceil \lg n \rceil - 1
\end{aligned}
$$

Master Theorem

Used for many divide-and-conquer recurrences

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

where $a \geq 1$, $b > 1$, and $f(n) > 0$.

a = (constant) number of sub-instances,
b = (constant) size ration of sub-instances,
f(n) = time used for dividing and recombining.

Based on the ***master theorem*** (Theorem 4.1).

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

Master Theorem

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

Case 1: $f(n)$ **is** $O(n^L)$ for some constant $L < \log_b a$.

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

Case 2: $f(n)$ **is** $\Theta(n^{\log_b a} \log^k n)$, for some $k \geq 0$.

Solution: $T(n)$ **is** $\Theta(n^{\log_b a} \log^{k+1} n)$

Case 3: $f(n)$ **is** $\Omega(n^L)$ for some constant $L > \log_b a$

and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some $c<1$ and all large n.

Solution: $T(n)$ **is** $\Theta(f(n))$

~~Master Theorem~~

Case 2: $f(n)$ **is** $\Theta(n^{\log_b a} \log^k n)$, for some $k \geq 0$.

Solution: $T(n)$ **is** $\Theta(n^{\log_b a} \log^{k+1} n)$

$T(n) = 27T(n/3) + \Theta(n^3/\log n)$

Compare $n^{\log_3 27}$ vs. n^3.

Since $3 = \log_3 27$ use **Case 2**

but $n^3/\log n$ **is not** $\Theta(n^3 \log^k n)$ for $k \geq 0$

Cannot use Master Method.

Divide-and-Conquer

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.

- Break up problem of size n into **two** equal parts of size $^n/_2$.
- Solve two parts recursively.
- Combine two solutions into overall solution in **linear time**.

Consequence.

- Straightforward: n^2.
- Divide-and-conquer: n log n.

Divide et impera.
Veni, vidi, vici.
- Julius Caesar

Binary Search

Find a value v in a sorted array of elements.

$$[a_0 \leq a_1 \leq, \ldots, \leq a_{Size-1}]$$

Size = number of elements.

Binary Search

Algorithm: binarySearch$(a, v, low, high)$
Input: array a, value v, lower and upper bound indices $low, high$ ($low = 0, high = n - 1$ initially)
Output: the index i of element v (if it is present), -1 (if v is not present)

```
if low == high then
   if  a[low] == v then
      return low
   else
      return −1
   end if
else
   mid ← (low + high)/2
   if v ≤ a[mid] then
      return binarySearch(a, v, low, mid )
   else
      return binarySearch(a, v, mid + 1, high)
   end if
end if
```

Recurrence Relation

Def. T(n) = number of comparisons to find v among n sorted elements.

Binary Search recurrence.

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

Solution. T(n) is O(log n) (Master Theorem Case 2).

D&C Multiplication

To multiply two n-digit integers:

- Multiply four $^n/_2$-digit integers.
- Add two $^n/_2$-digit integers, and shift to obtain result.

$$
\begin{aligned}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0
\end{aligned}
$$

$$
\mathrm{T}(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow \mathrm{T}(n) \text{ is } \Theta(n^2)
$$

assumes n is a power of 2

Karatsuba Multiplication

To multiply two n-digit integers:

- Add two $^n/_2$ digit integers.
- Multiply **three** $^n/_2$-digit integers.
- Add, subtract, and shift $^n/_2$-digit integers to obtain result.

$$
\begin{aligned}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \\
&= 2^n \cdot \underset{A}{x_1 y_1} + 2^{n/2} \cdot \left(\underset{B}{(x_1 + x_0)(y_1 + y_0)} - \underset{A}{x_1 y_1} - \underset{C}{x_0 y_0} \right) + \underset{C}{x_0 y_0}
\end{aligned}
$$

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit operations.

$$
T(n) \le \underbrace{T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + T\left(1 + \lceil n/2 \rceil\right)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}
$$

$$
\Rightarrow T(n) \text{ is } O(n^{\log_2 3}) \text{ is } O(n^{1.585})
$$

Karatsuba Multiplication

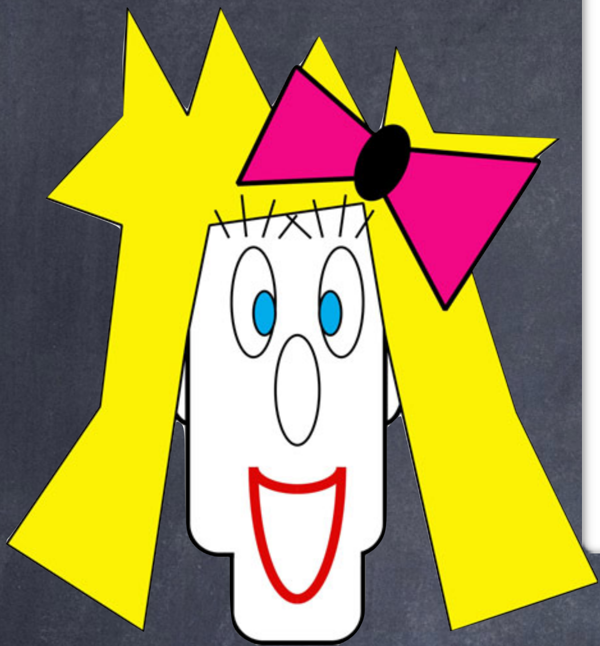
Generalization: $O(n^{1+\varepsilon})$ for any $\varepsilon > 0$.

Best known: $n \log n \, 2^{O(\log^* n)}$

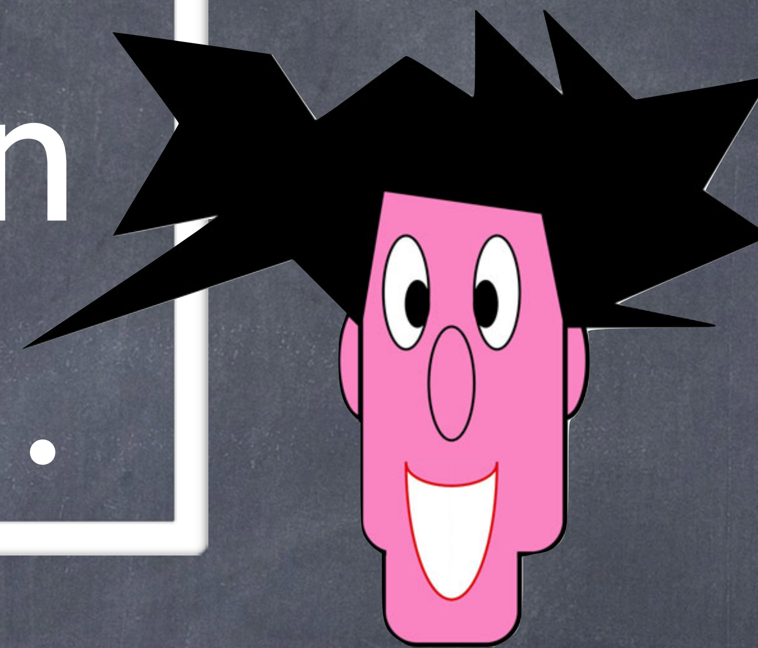
$$\text{where } \log^*(x) = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^*(\log x) & \text{if } x > 1 \end{cases}$$

Conjecture: $\Omega(n \log n)$ but not proven yet.

Alice and Bob's Adventures in Cryptoland...



Alice



Bob


8RdewtU5qkLa$es!T9@
I(D%eXhDqliykl#2cV7dEwnMs
H&fs@tyHvFGhaOKpTrGbl.Z/rUih*
B7B3tdsjUila

Public-Key Cryptography

Fast Modular Exponentiation

- **Input:** base x ,modulus N and exponent e.
- **Output:** $x^e\ \%N$.

y = 1
WHILE e>0 **DO**
  IF e%2 = 1 **THEN** $y = xy\ \%N$
  e = e/2; $x = x^2\ \%N$
return y

- running time is $O(|e|*|x|^2) = O(|x|^3)$

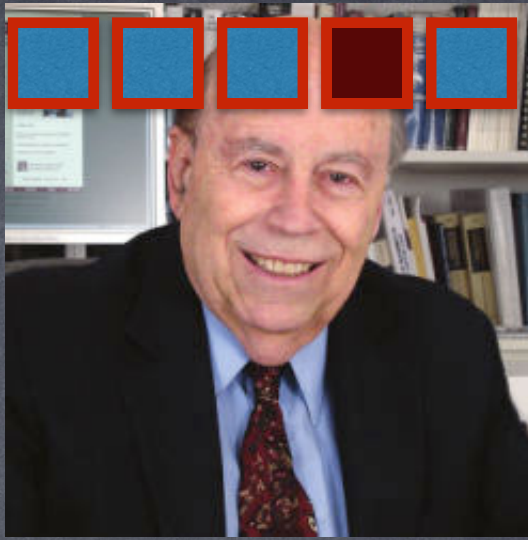
Euclidian Algorithm

- **Input**: integers a,b.
- **Output**: g,x,y such that g=GCD(a,b).

```
g = a; g' = b;
WHILE g'>0 DO
   k = g/g'
   g'' = g-kg';                         // g'' = g %g'
   g = g';
   g' = g'';
return g
```

- running time is $O(|a|*|b|)$

Primality Testing



- **Input**: base a, modulus N.
- **Output**: Is N a base-a pseudo-prime? .

  IF GCD(a,N) > 1 **THEN return False**
  set $s \geq 0$ and t (odd) s.t. N-1 = $t2^s$
  $x = a^2 \% N$; $y = N$-1
  FOR i = 1 **TO** s
      IF x = 1 AND y = N-1 **THEN return True**
      $y = x$; $x = x^2 \% N$
  return False

- running time is $O(|N|^4)$

RSA Encryption

- **Gen**: on input 1^n run GenRSA(1^n) and obtain (N,e,d). Let $\langle N,e \rangle$ be the public-key and $\langle d \rangle$ the private key.

- **Enc**: on input $\langle N,e \rangle$ and a message $0<m<N$ compute
$$c = m^e \bmod N$$

- **Dec**: on input $\langle d \rangle$ and a ciphertext $0<c<N$ compute
$$m = c^d \bmod N$$

Quantum Factoring

1990's

NIST's Plan for the Future

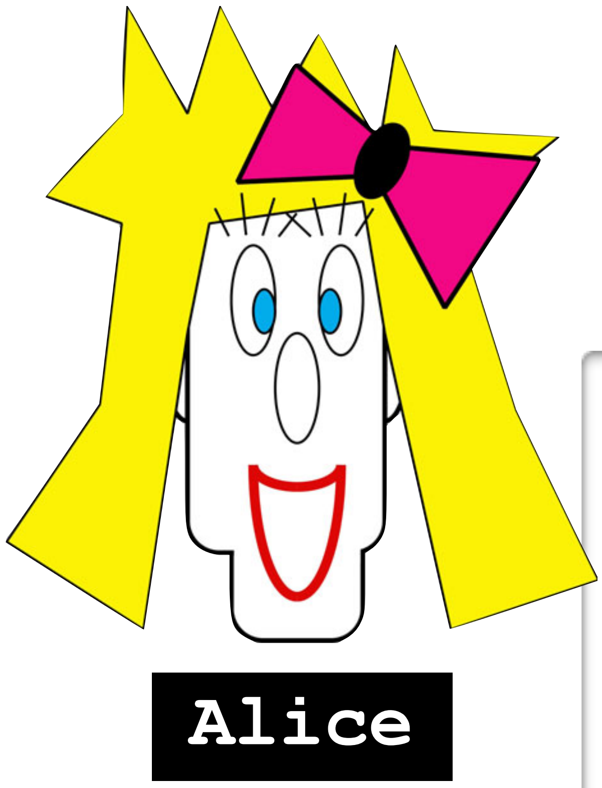
Dustin Moody
Post Quantum Cryptography Team
National Institute of Standards and Technology (NIST)

24 Feb 2016

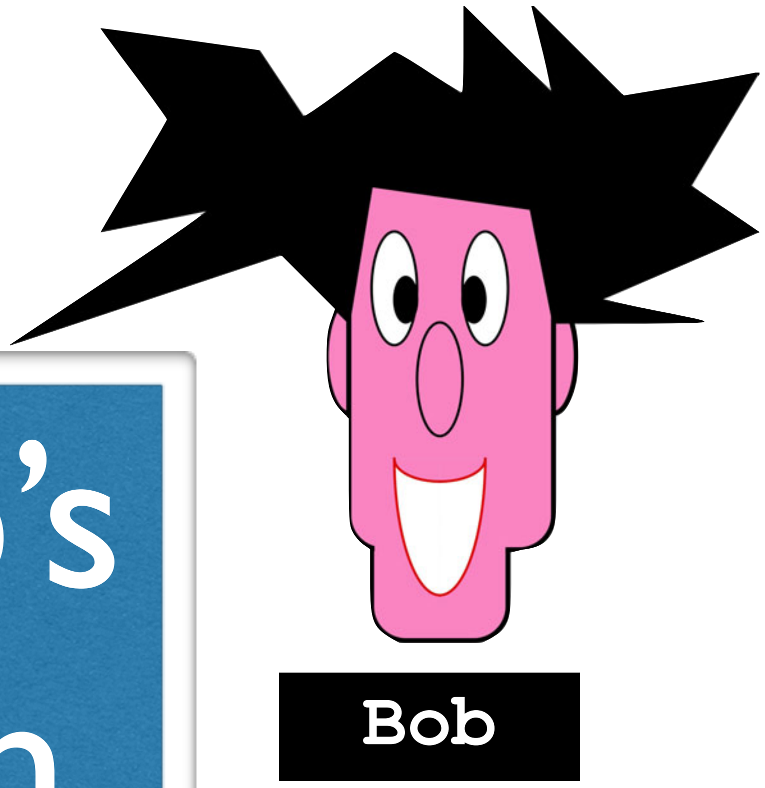
Timeline

- Fall 2016 – formal Call For Proposals
- Nov 2017 – Deadline for submissions
- 3-5 years – Analysis phase
  - NIST will report its findings
- 2 years later – Draft standards ready

- Workshops
  - Early 2018 – submitter's presentations
  - One or two during the analysis phase

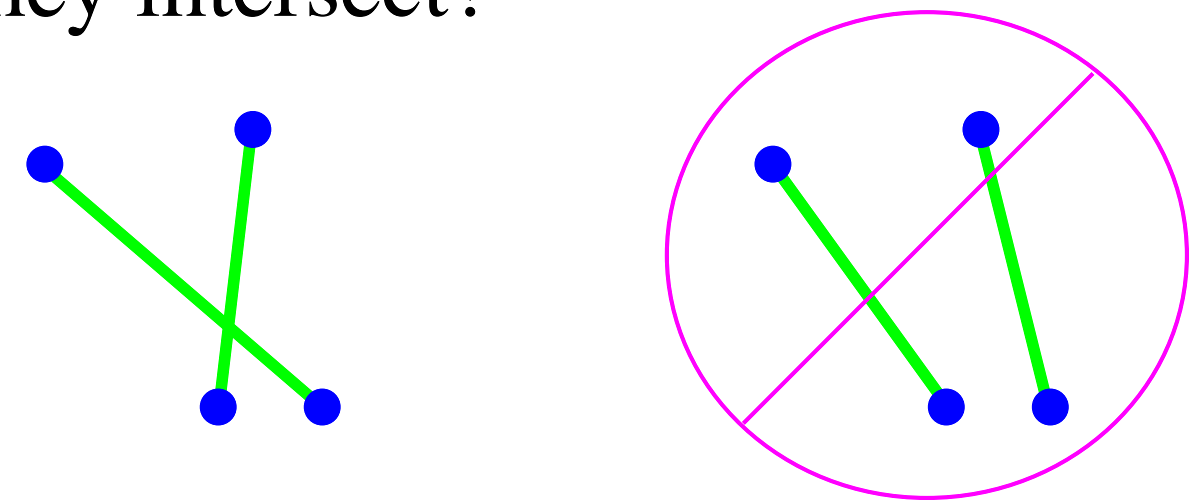


Alice and Bob's Adventures in GEOM-land...



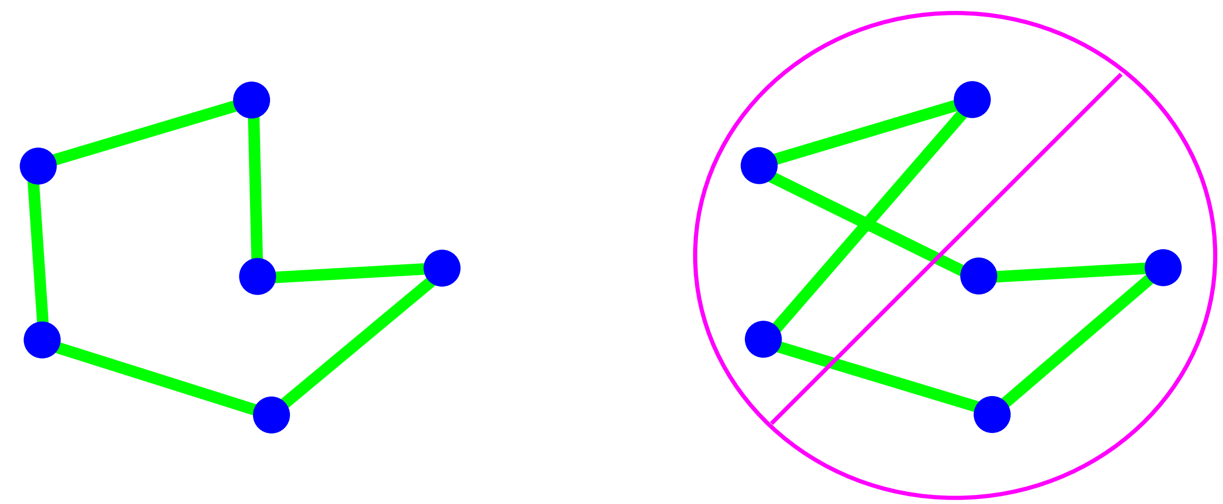
Some Geometric Problems

Segment intersection: Given two segments, do they intersect?



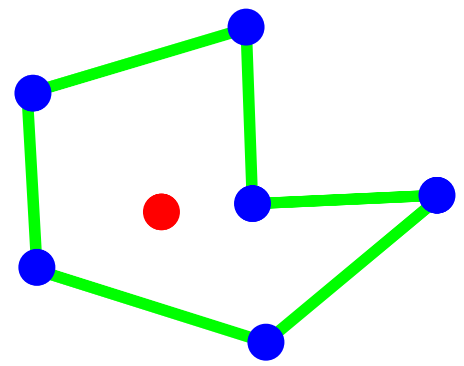
Some Geometric Problems

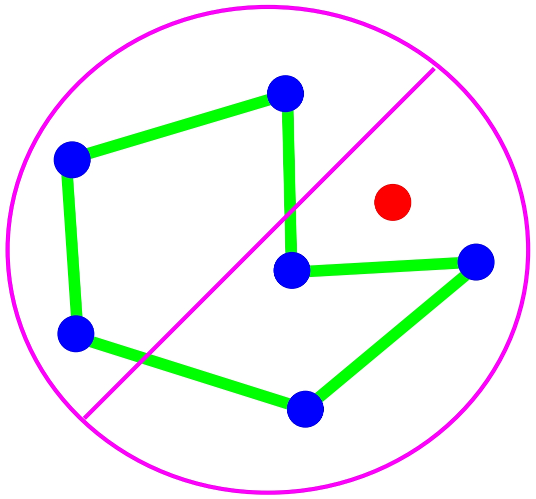
Simple closed path: Given a set of points, find a nonintersecting polygon with vertices on the points.



Some Geometric Problems

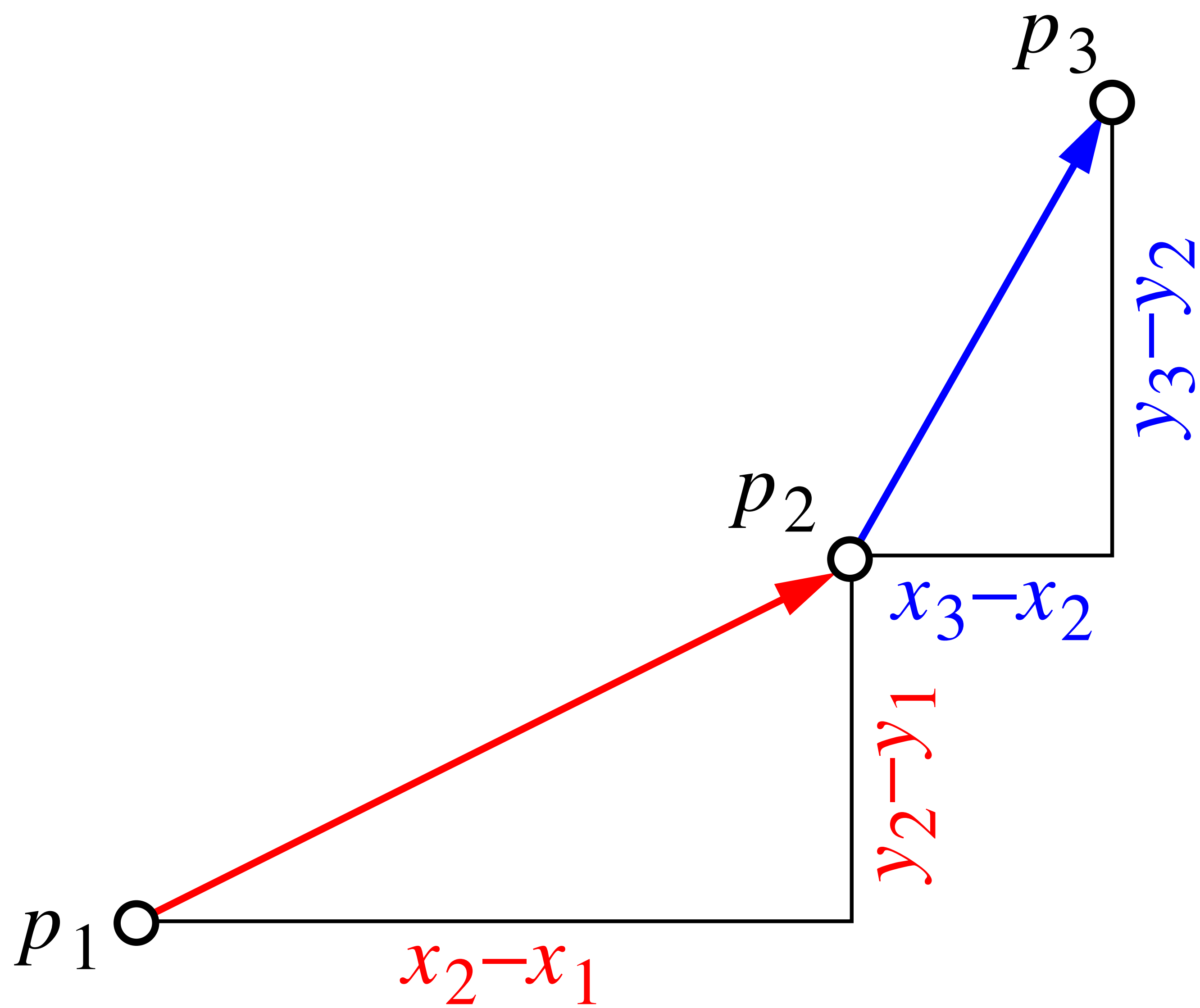
Inclusion in polygon: Is a point inside or outside a polygon?





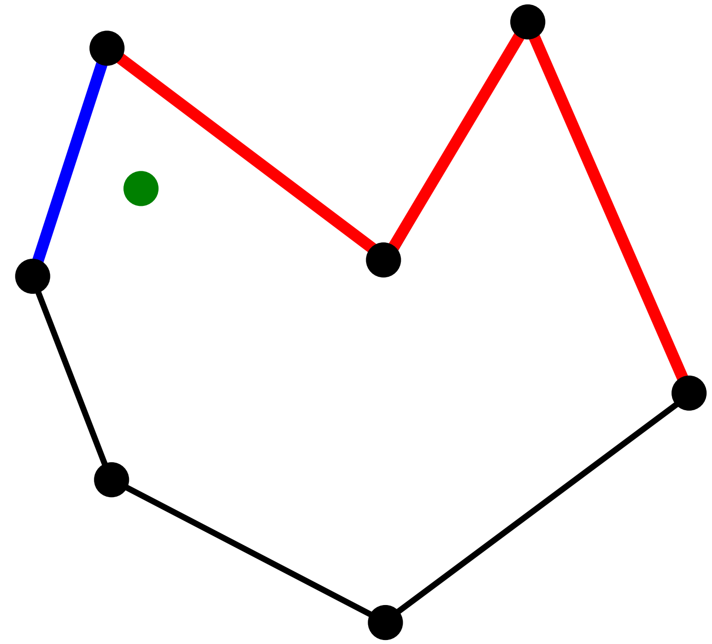
How to Compute the Orientation

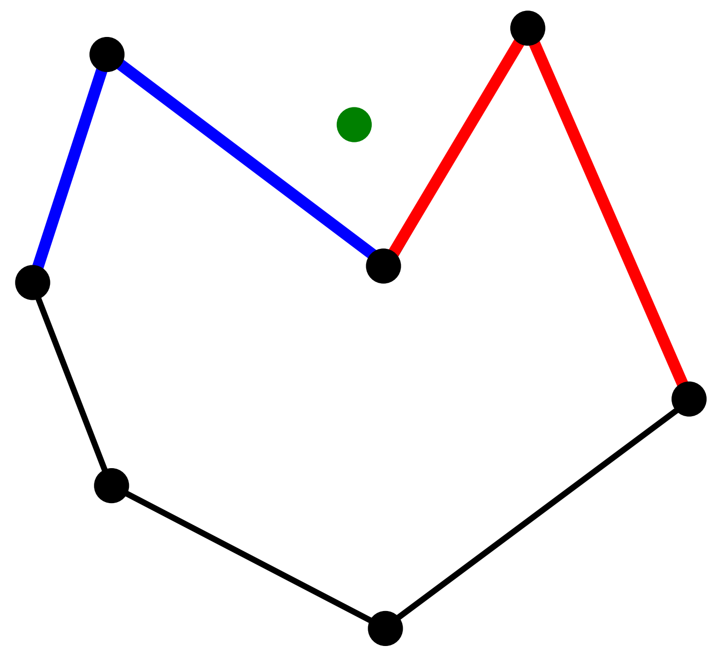
- slope of segment (p_1,p_2): $\sigma = (y_2 - y_1) / (x_2 - x_1)$
- slope of segment (p_2,p_3): $\tau = (y_3 - y_2) / (x_3 - x_2)$



Point Inclusion

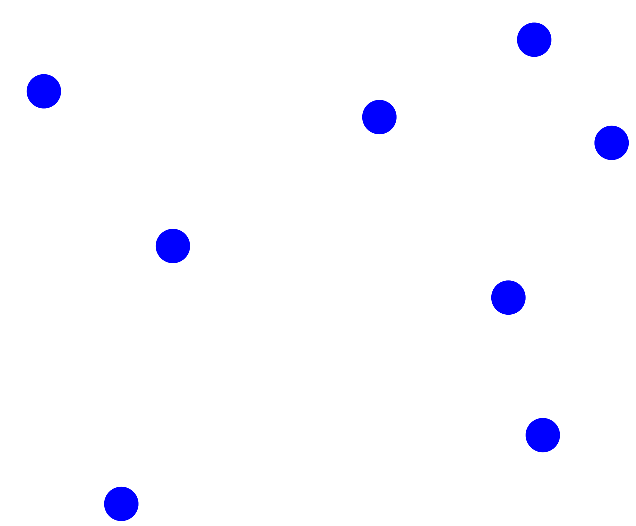
- given a polygon and a point, is the point inside or outside the polygon?
- orientation helps solving this problem in linear time



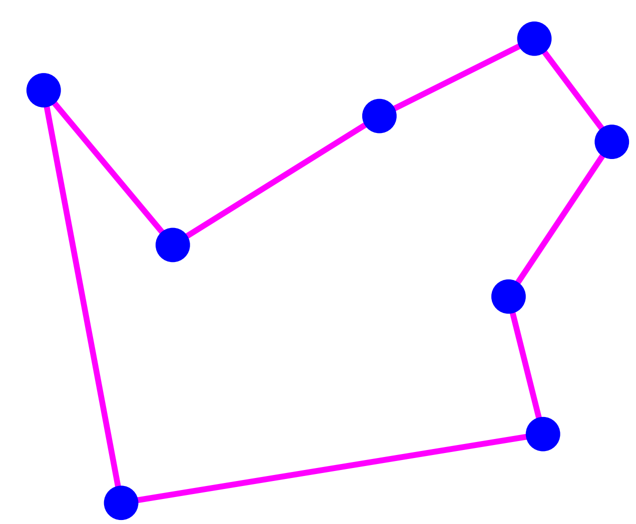


Simple Closed Path — Part I

- Problem: Given a set of points ...



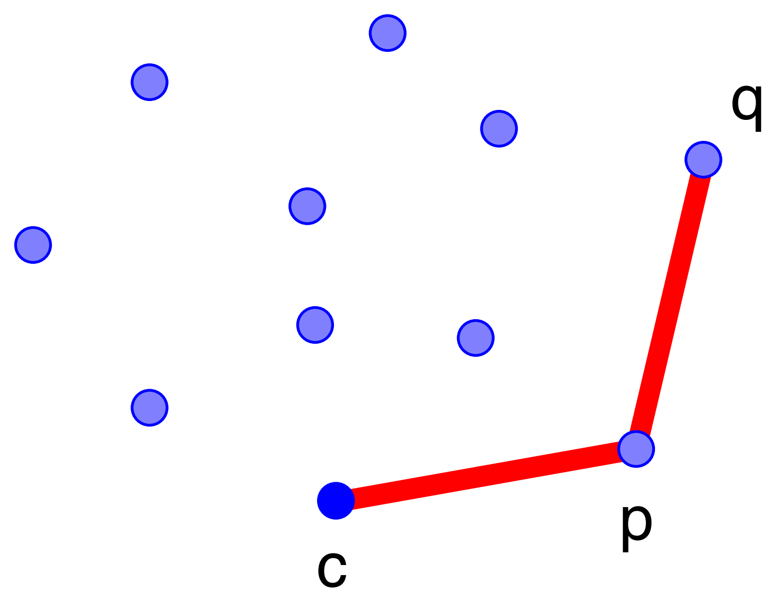
- “Connect the dots” without crossings



Package Wrap

- given the current point, how do we compute the next point?
- set up an orientation tournament using the current point as the anchor-point...
- the next point is selected as the point that beats all other points at CCW orientation, i.e., for any other point, we have

orientation(c, p, q) = CCW

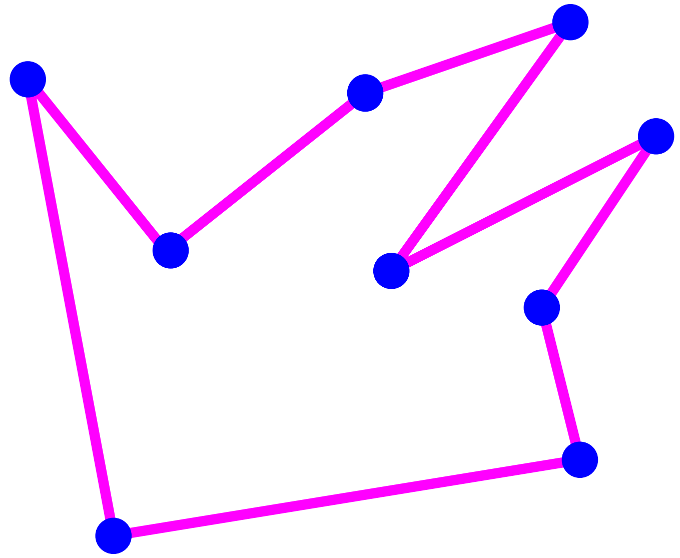


Time Complexity of Package Wrap

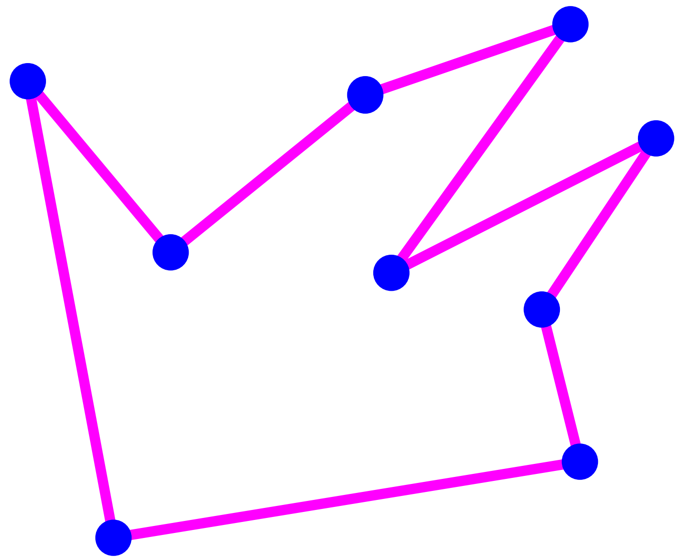
- For every point on the hull we examine all the other points to determine the next point
- Notation:
  - N: number of points
  - M: number of hull points ($M \leq N$)
- Time complexity:
  - $\Theta(MN)$
- Worst case: $\Theta(N^2)$
  - all the points are on the hull ($M=N$)
- Average case: $\Theta(N \log N)$ — $\Theta(N^{4/3})$
  - for points randomly distributed inside a ***square***, $M = \Theta(\log N)$ on average
  - for points randomly distributed inside a ***circle***, $M = \Theta(N^{1/3})$ on average

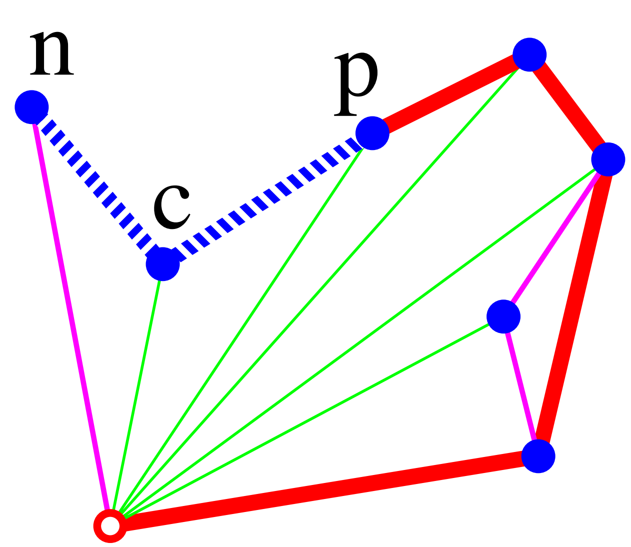
Graham Scan

- Form a simple polygon (connect the dots as before)

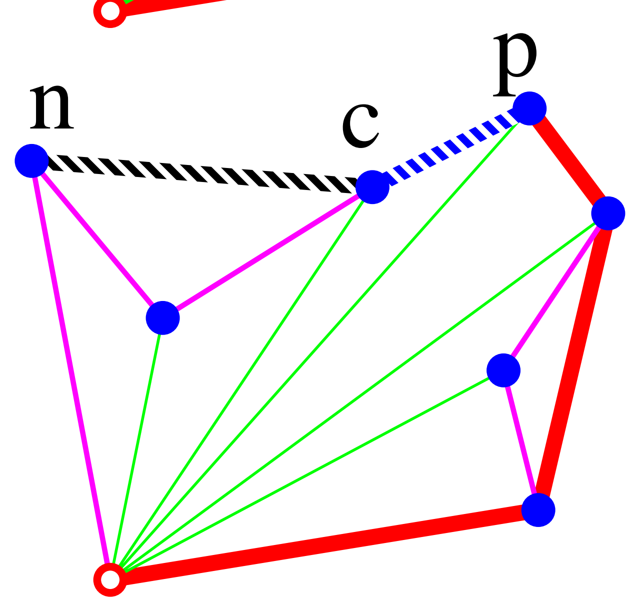


- Remove points at concave angles

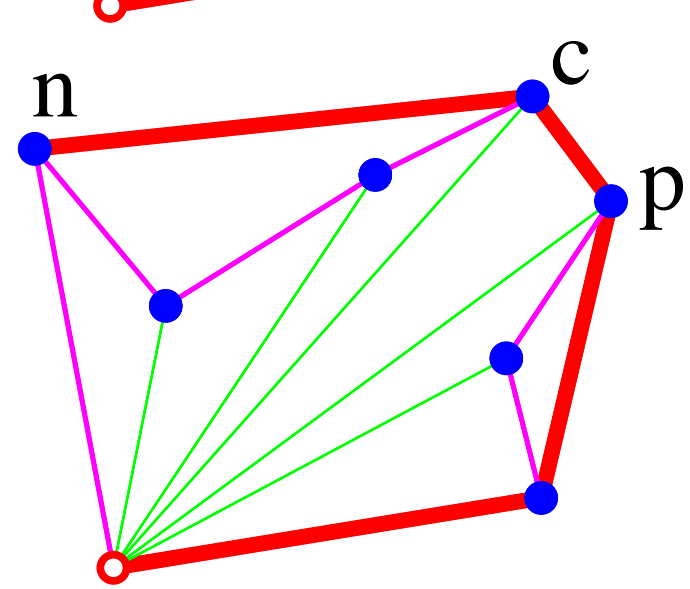




(p,c,n) is a
right turn!



(p,c,n) is a
right turn!

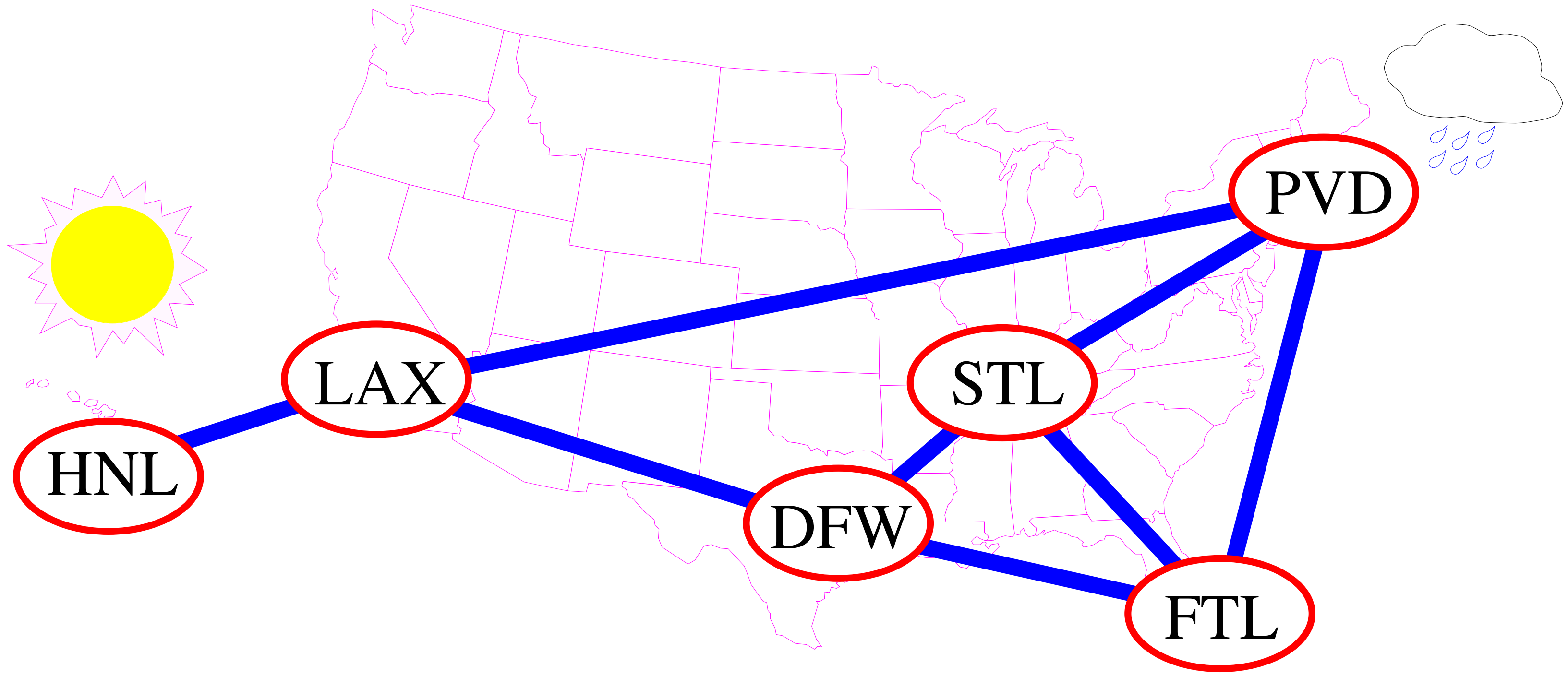


Time Complexity of Graham Scan

- Phase 1 takes time O(N logN)
  - points are sorted by angle around the anchor
- Phase 2 takes time O(N)
  - each point is inserted into the sequence exactly once, and
  - each point is removed from the sequence at most once
- Total time complexity O(N log N)

GRAPHS

- Definitions
- Examples
- The Graph ADT



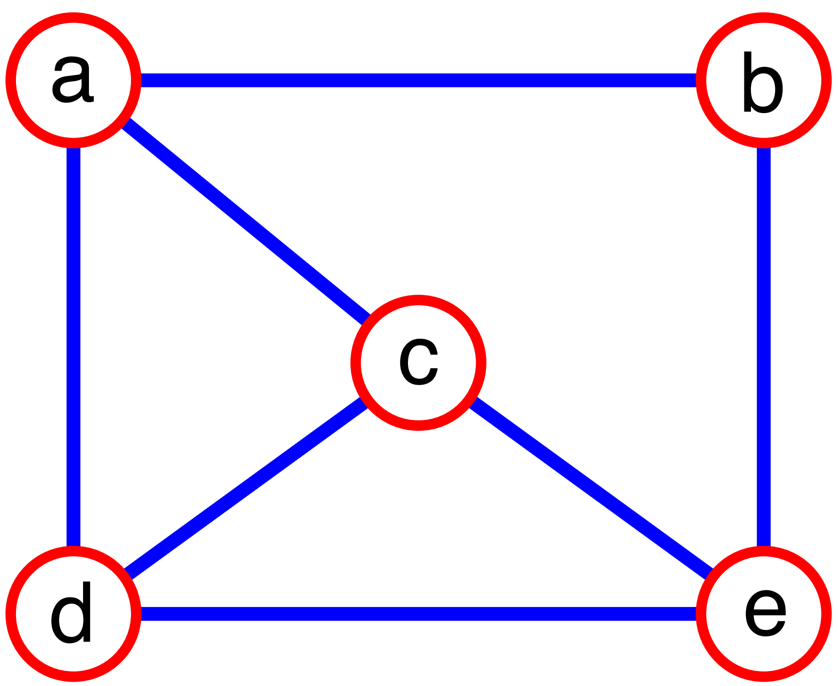
What is a Graph?

- A **graph G** = (**V**,**E**) is composed of:

  V: set of *vertices*

  E: set of *edges* connecting the *vertices* in **V**

- An **edge e** = (u,v) is a pair of vertices
- Example:



V= {a,b,c,d,e}

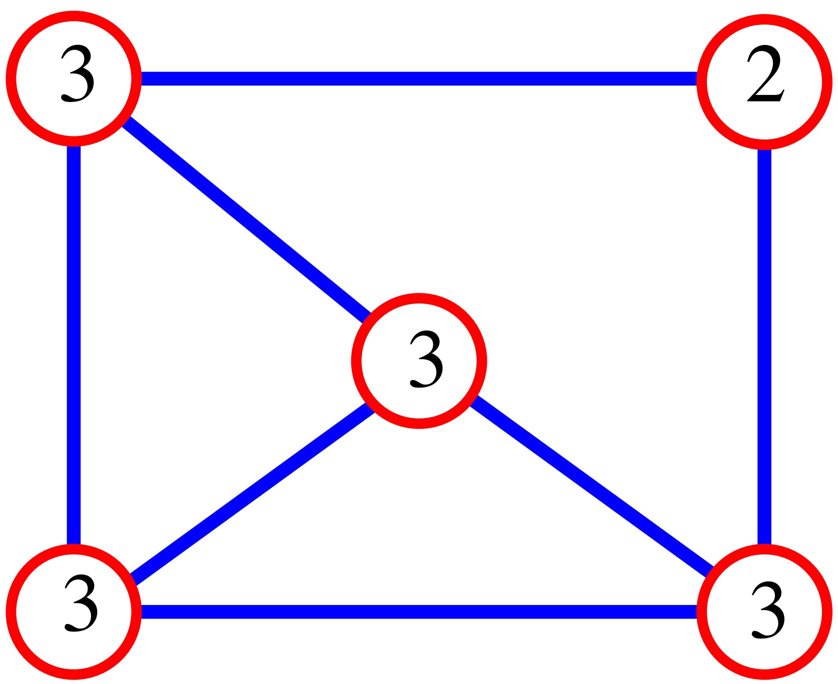
E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

Applications

Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

Graph Terminology

- adjacent vertices: connected by an edge
- degree (of a vertex): # of adjacent vertices

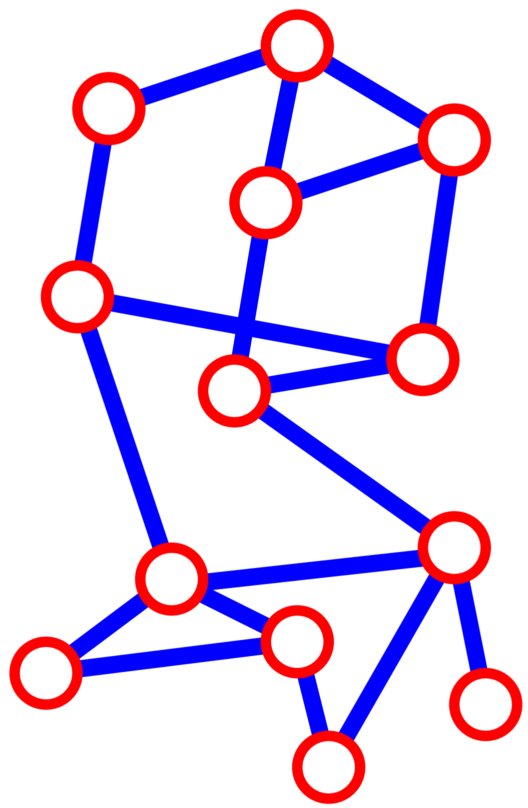


$$\sum_{v \in V} \deg(v) = 2(\#\ \text{edges})$$

- Since adjacent vertices each count the adjoining edge, it will be counted twice

Spanning Tree

- A **spanning tree** of **G** is a subgraph which
  - is a tree
  - contains all vertices of **G**

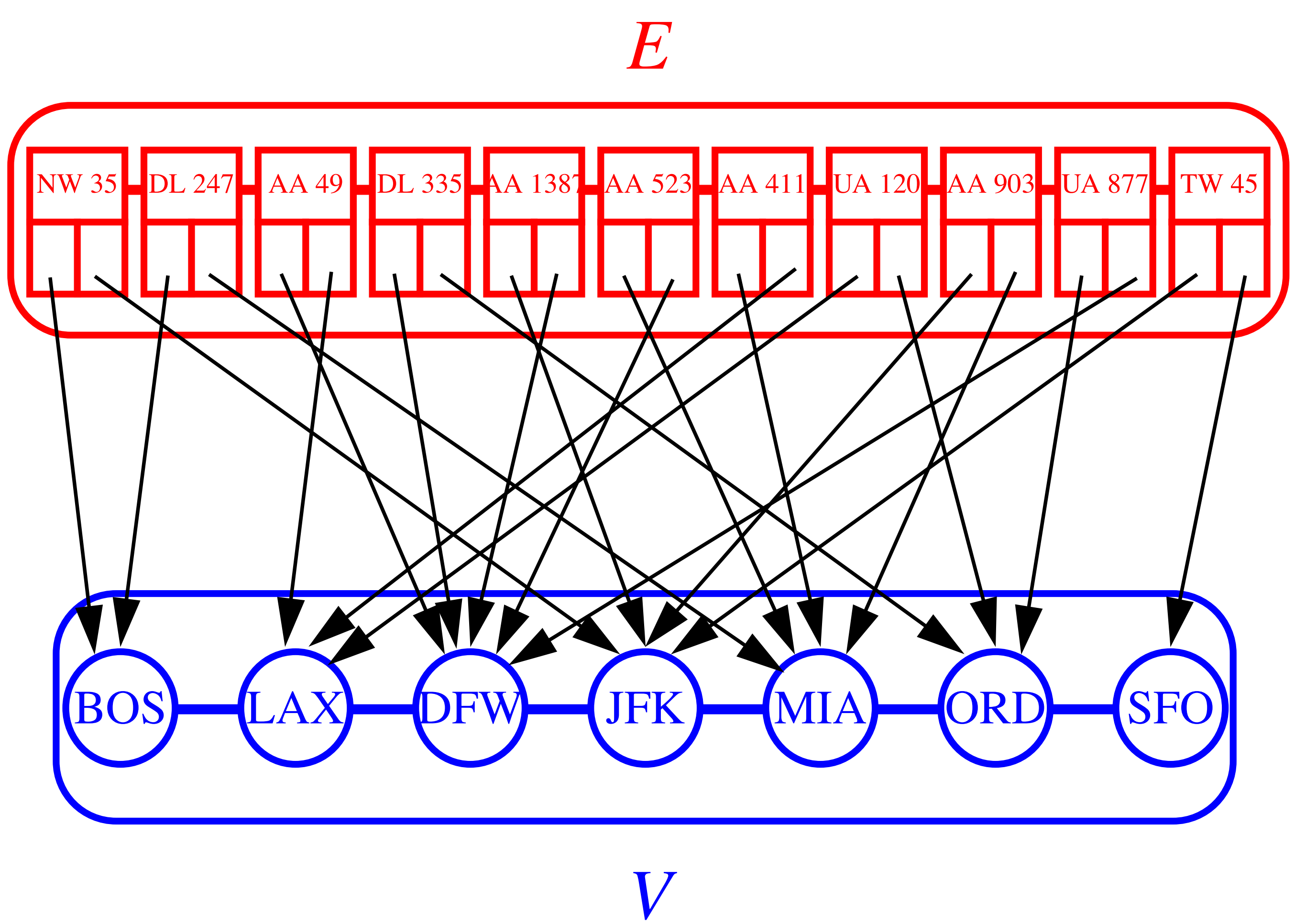


G   spanning tree of **G**

- Failure on any edge disconnects system (least fault tolerant)

Data Structures for Graphs

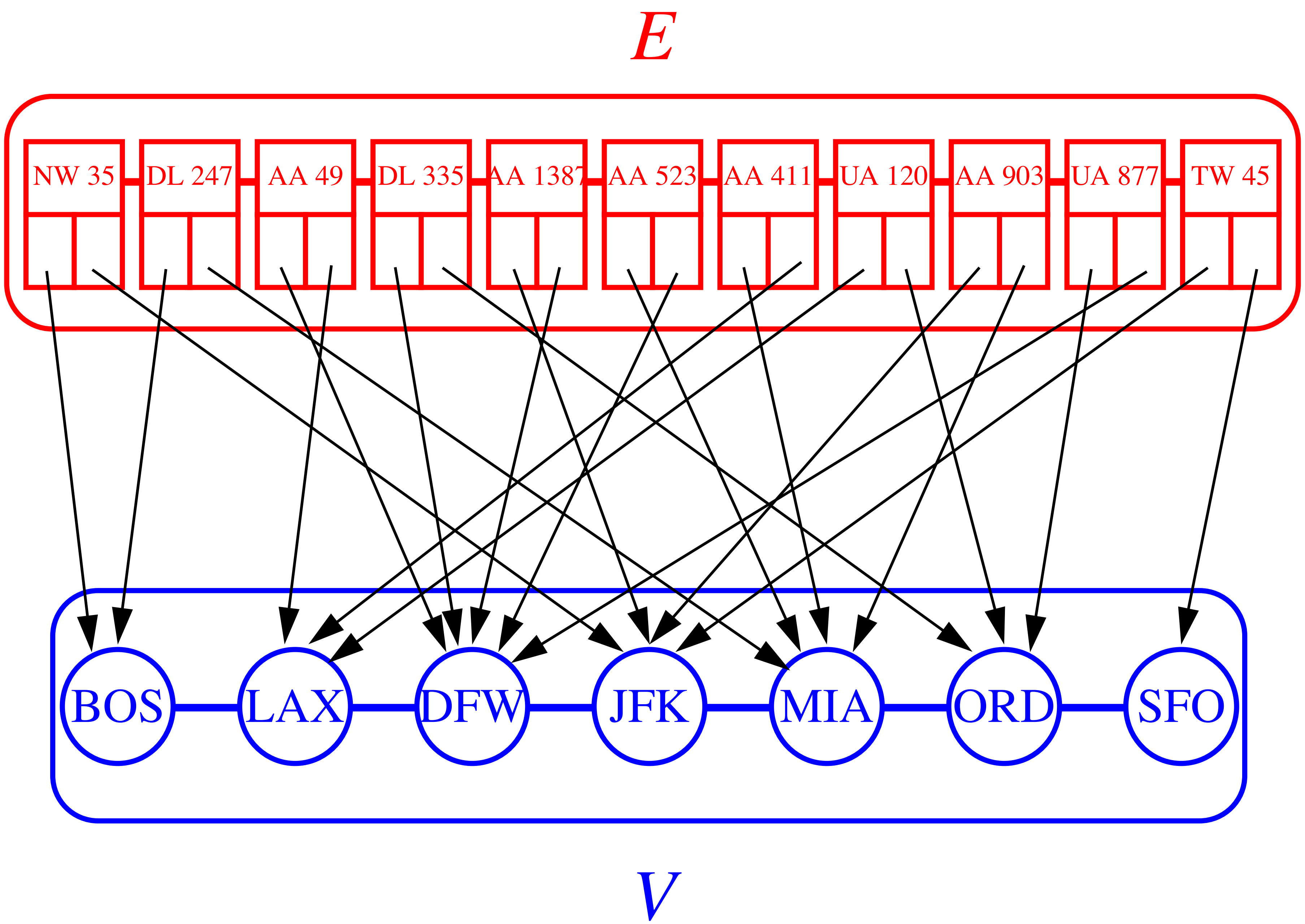
- Edge list
- Adjacency lists
- Adjacency matrix



Edge List

[redacted]
the edges into unsorted sequences.

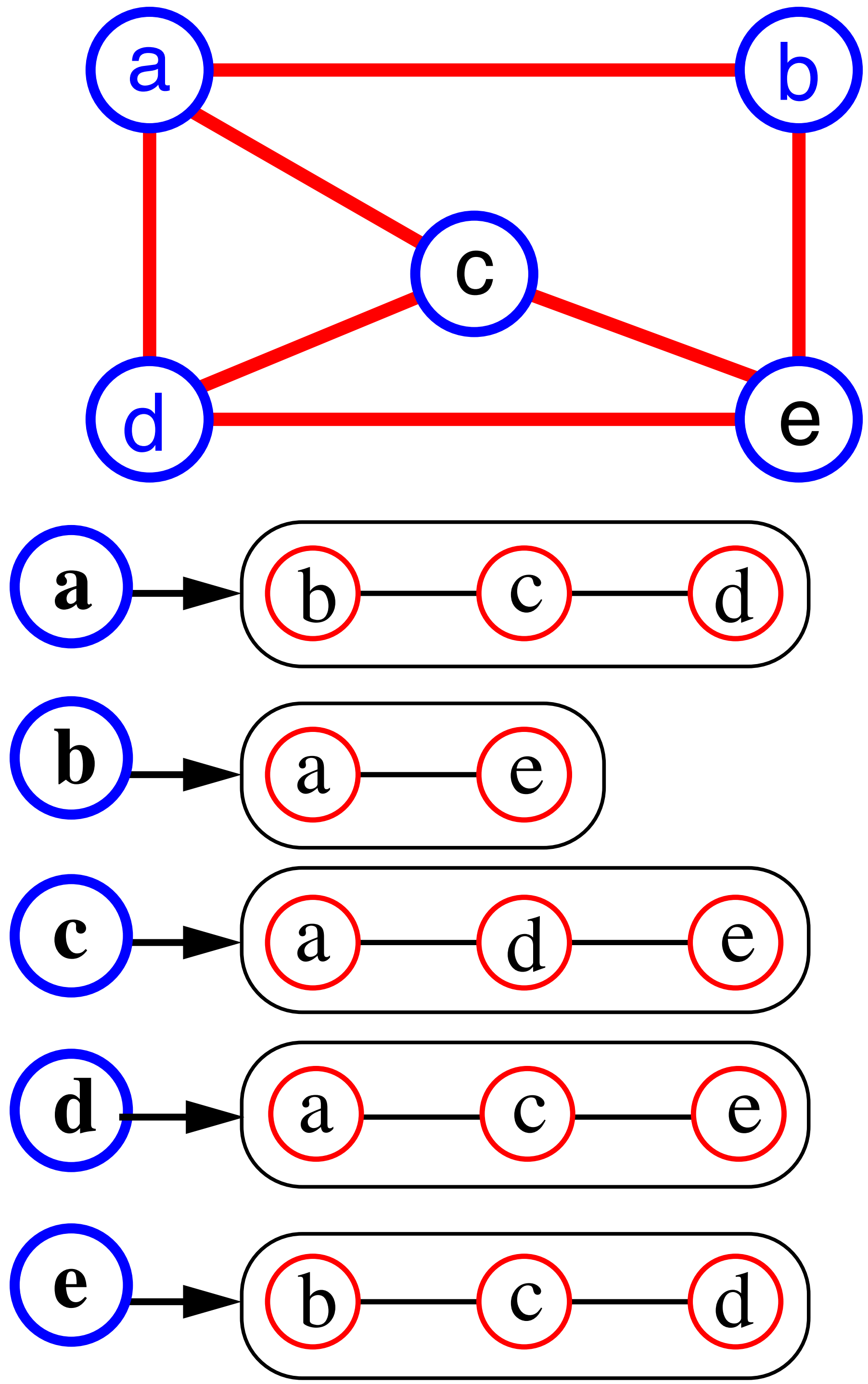
- Easy to implement.
- Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence



sequence of vertices adjacent to v

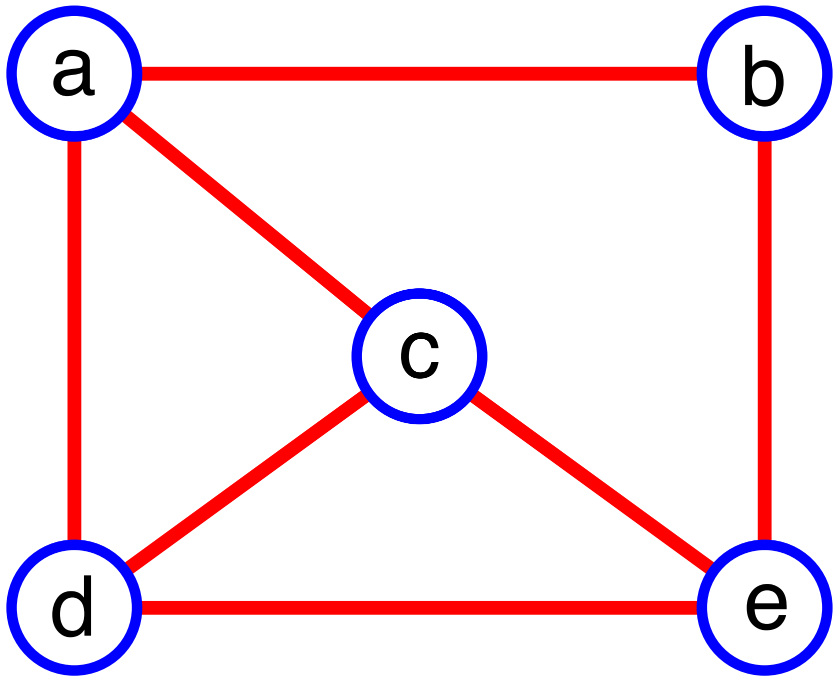
Adjacency List (traditional)

- represent the graph by the adjacency lists of all the vertices



- Space = $\Theta(\mathbf{N} + \sum \deg(\mathbf{v})) = \Theta(\mathbf{N} + \mathbf{M})$

Adjacency Matrix (traditional)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- matrix M with entries for all pairs of vertices
- M[i,j] = true means that there is an edge (i,j) in the graph.
- M[i,j] = false means that there is no edge (i,j) in the graph.
- There is an entry for every possible edge, therefore:
  Space = $\Theta(\mathbf{N^2})$

Edge List

Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices, areAdjacent, degree, inDegree, outDegree	O(m)
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
removeVertex	O(m)

Adjacency List

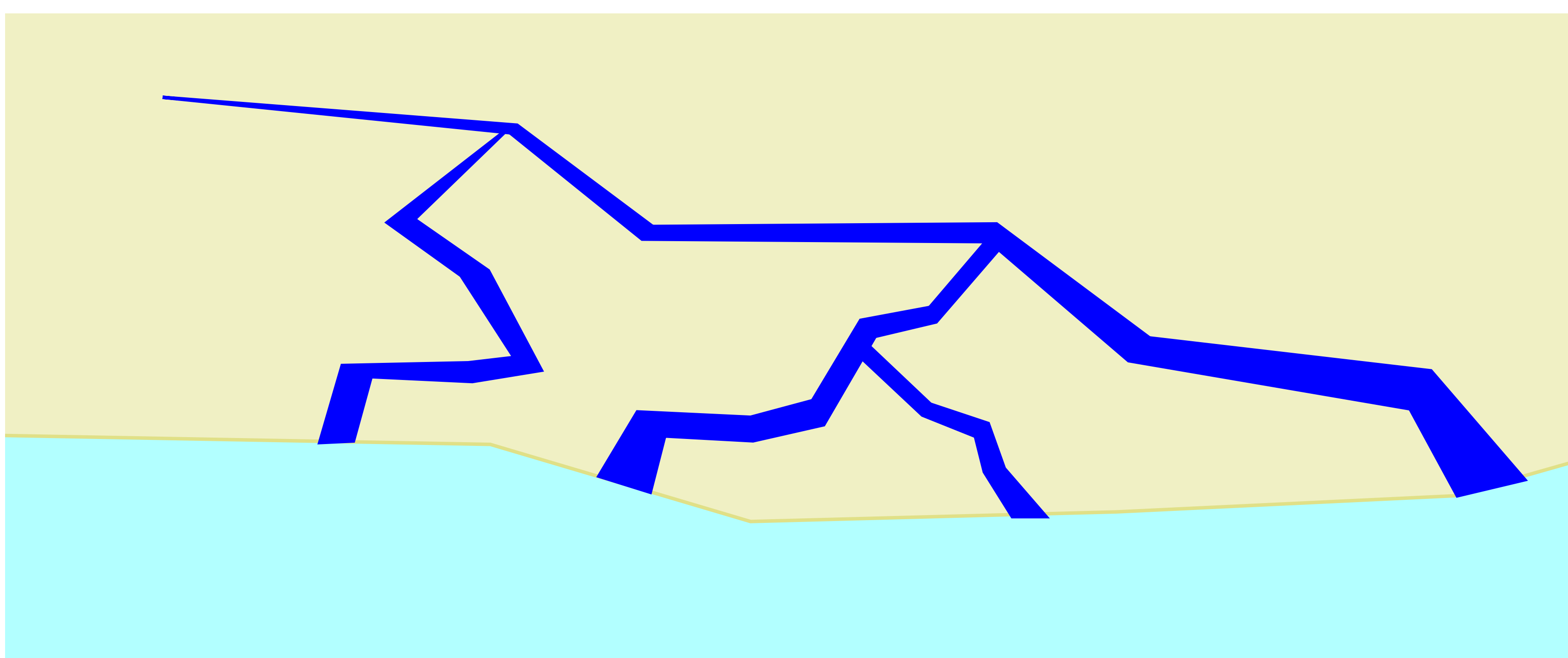
Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v)	O(deg(v))
areAdjacent(u, v)	O(min(deg(u), deg(v)))
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection,	O(1)
removeVertex(v)	O(deg(v))

Adjacency Matrix

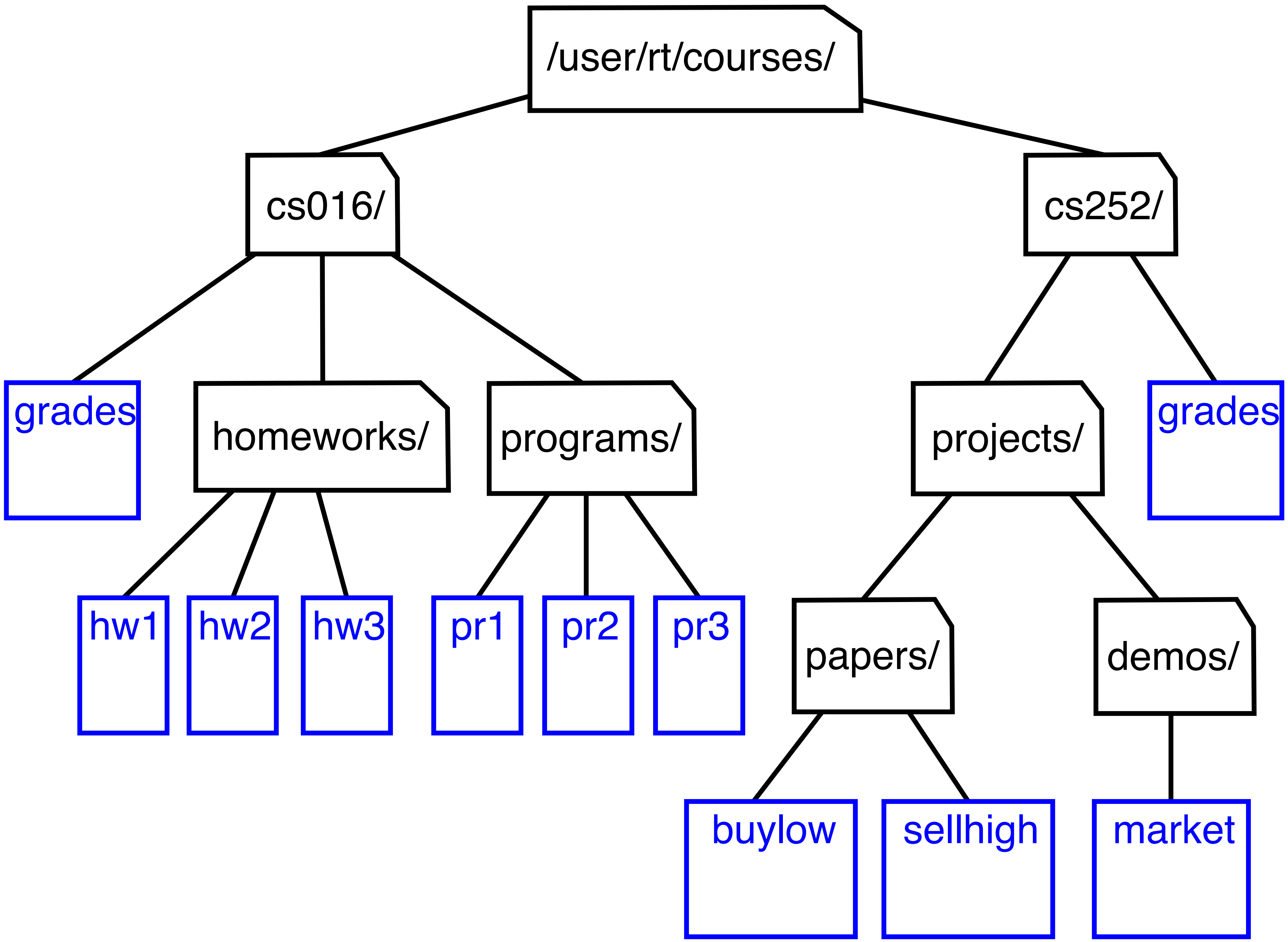
Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices,	O(n)
areAdjacent	O(1)
insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
insertVertex, removeVertex	$O(n^2)$

TREES

- trees
- binary trees
- traversals of trees
- template method pattern
- data structures for trees



- Unix or DOS/Windows file system



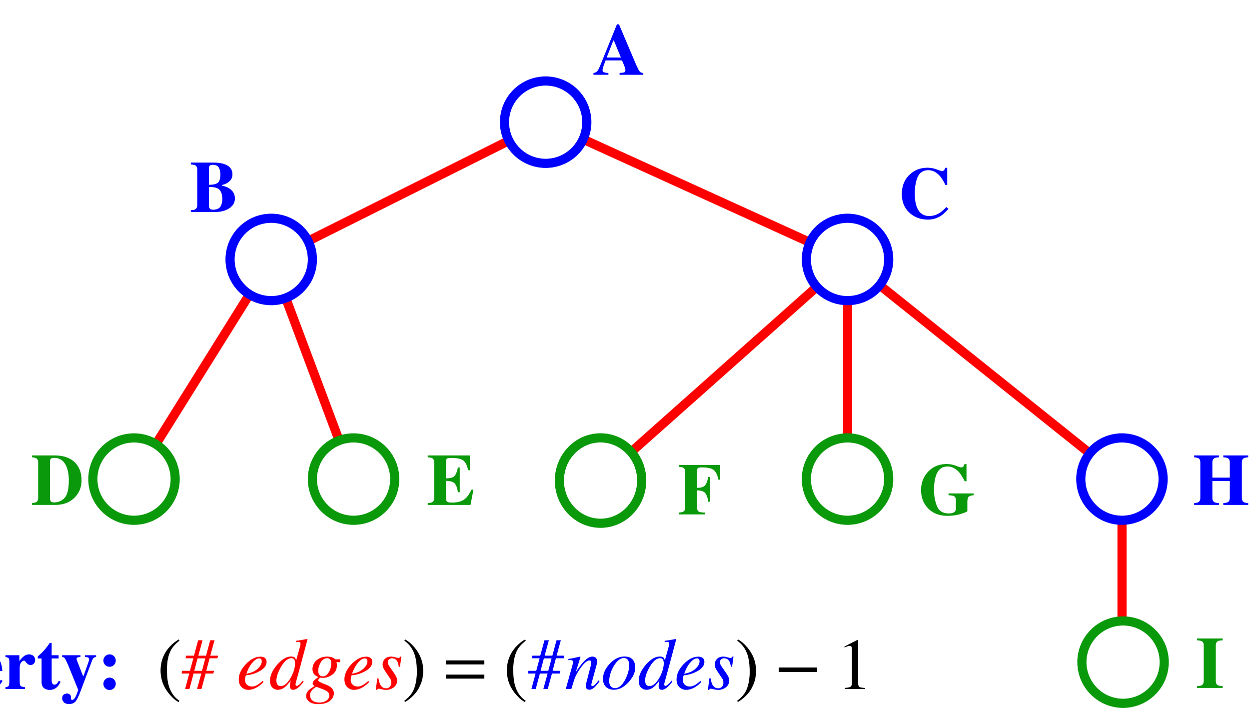
Terminology

- ***A*** is the ***root*** node.
- ***B*** is the ***parent*** of D and E.
- ***C*** is the ***sibling*** of B
- ***D*** and ***E*** are the ***children*** of B
- ***D, E, F, G***, *I* are ***external nodes*, or *leaves***
- ***A, B, C, H*** are ***internal nodes***

•The ***depth*** (***level***) of ***E*** is ***2***

•The ***height*** of the tree is ***3***

•The ***degree*** of node ***B*** is ***2***

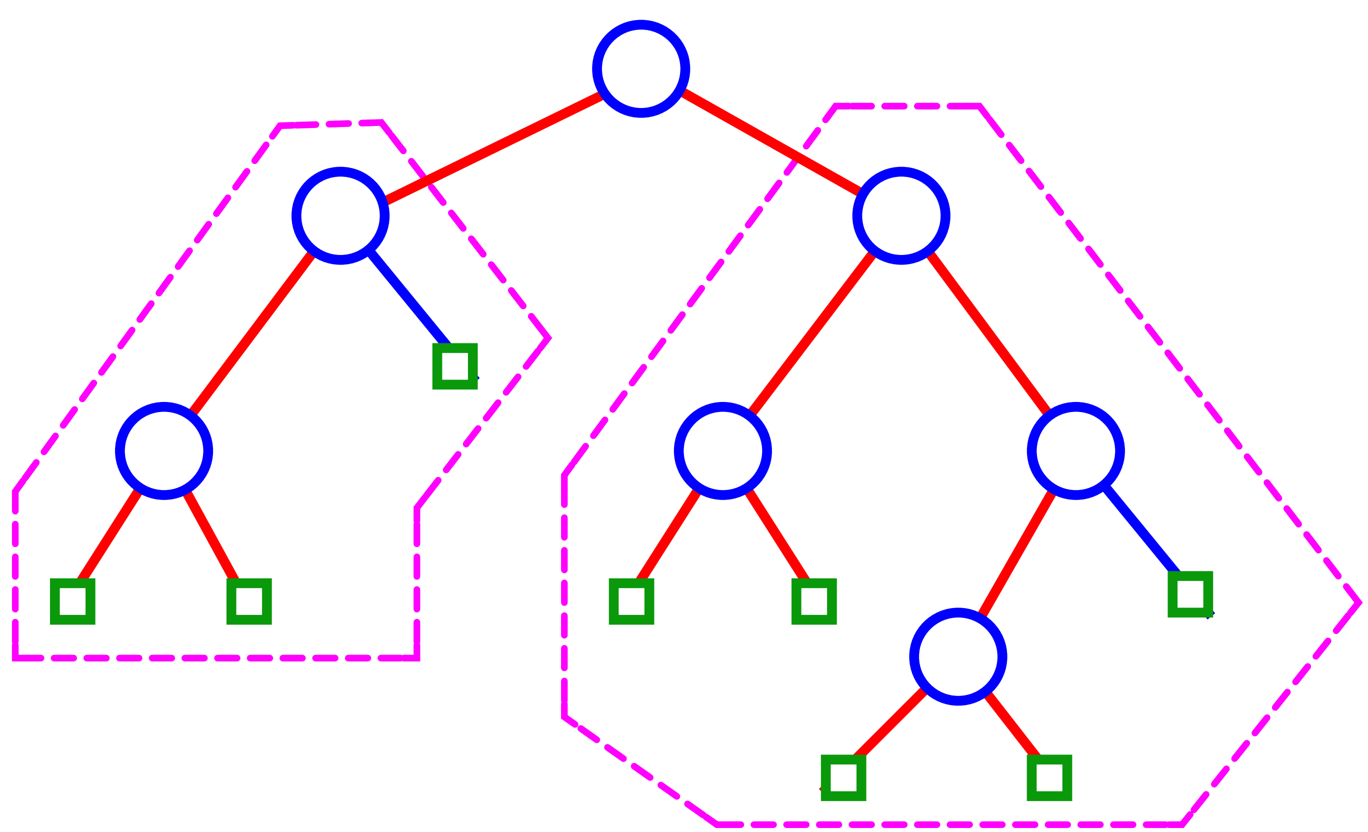


Property: (*# edges*) = (*#nodes*) – 1

- ***Ordered tree:*** the children of each node are ordered.
- ***Binary tree:*** ordered tree with all internal nodes of ***degree 2***.

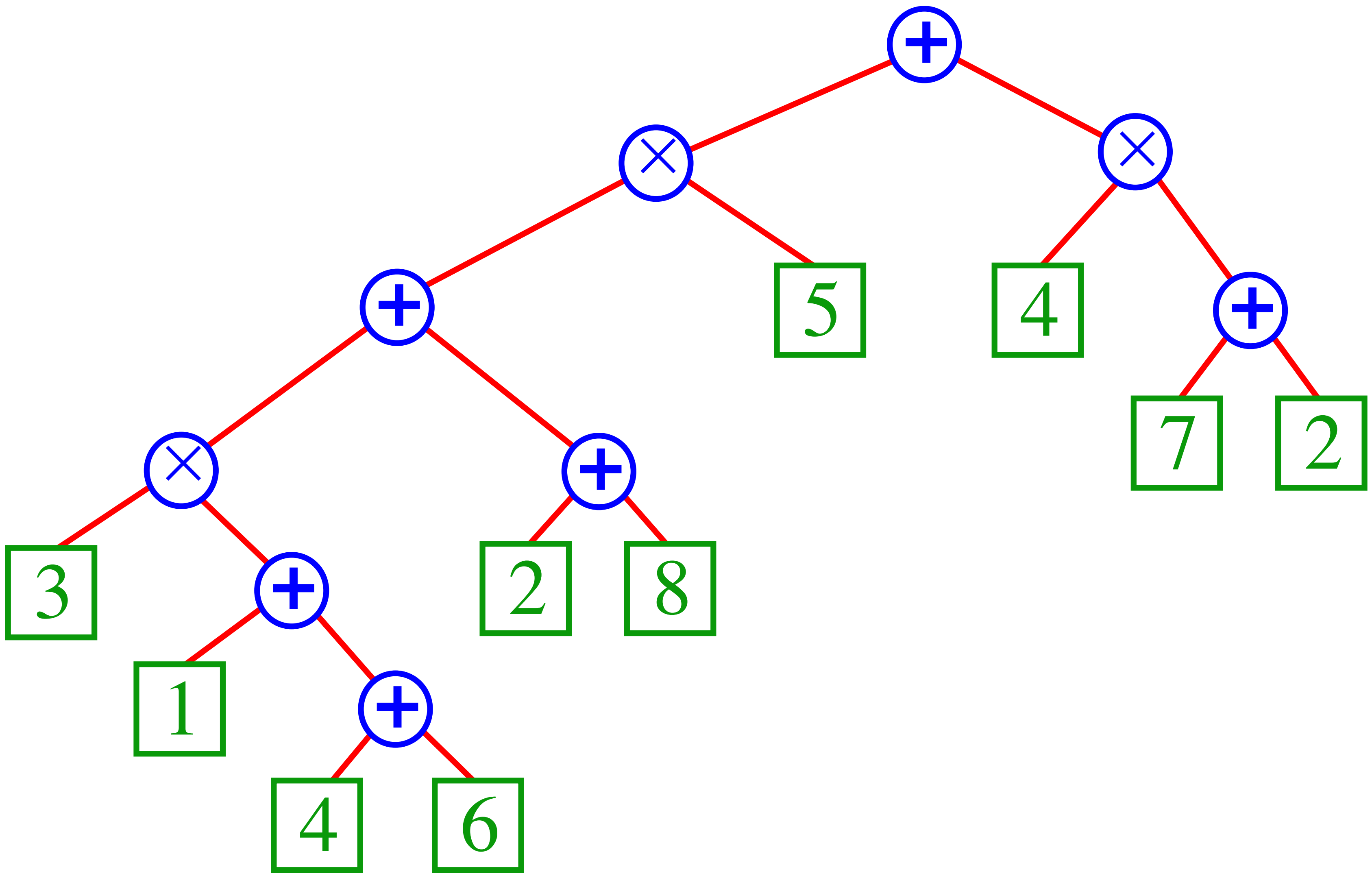
•Recursive definition of binary tree:

- A ***binary tree*** is either
  - a n external node (leaf), **or**
  - a n internal node (the ***root***) and two binary trees (***left subtree*** and ***right subtree***)



Examples of Binary Trees

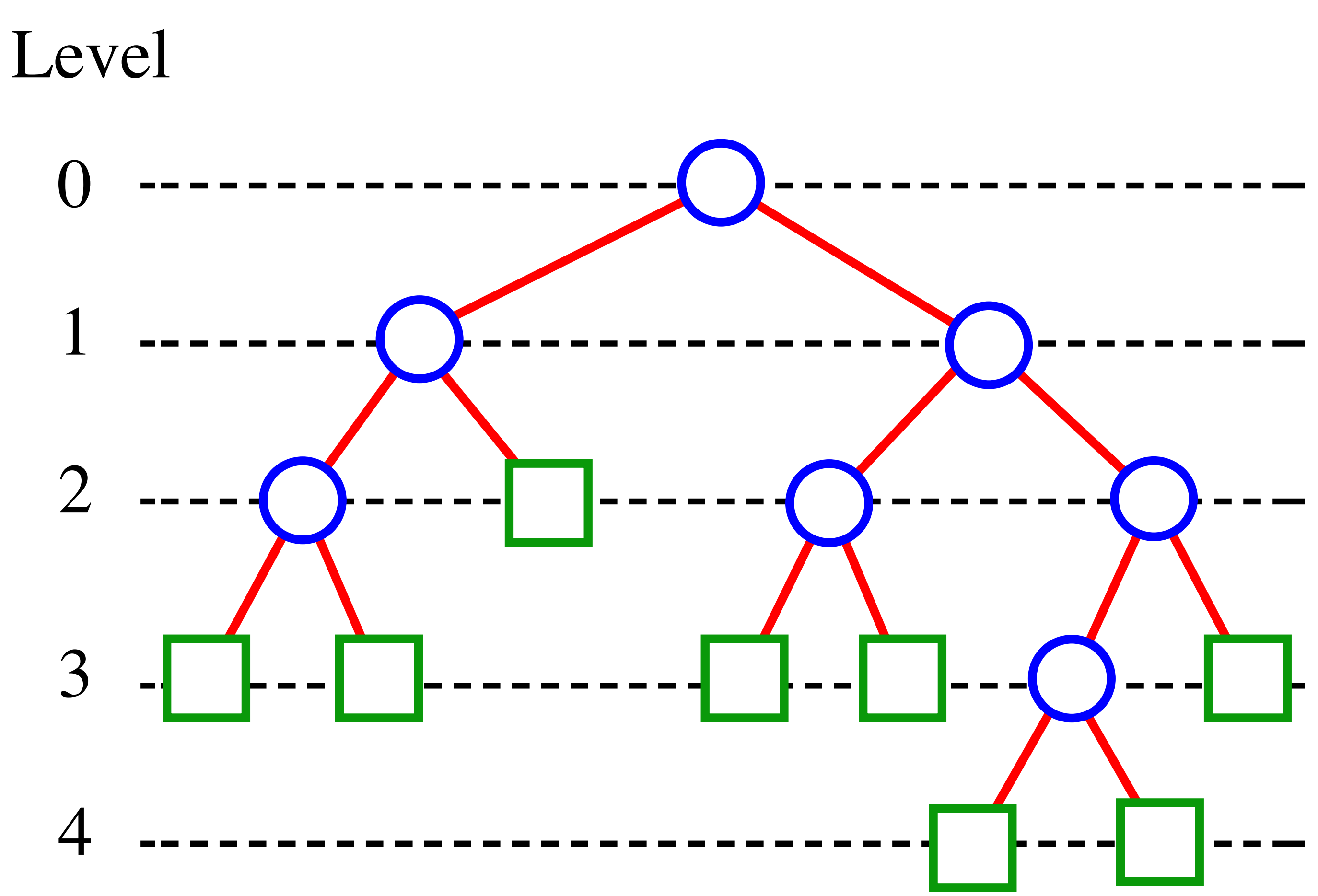
- arithmetic expression



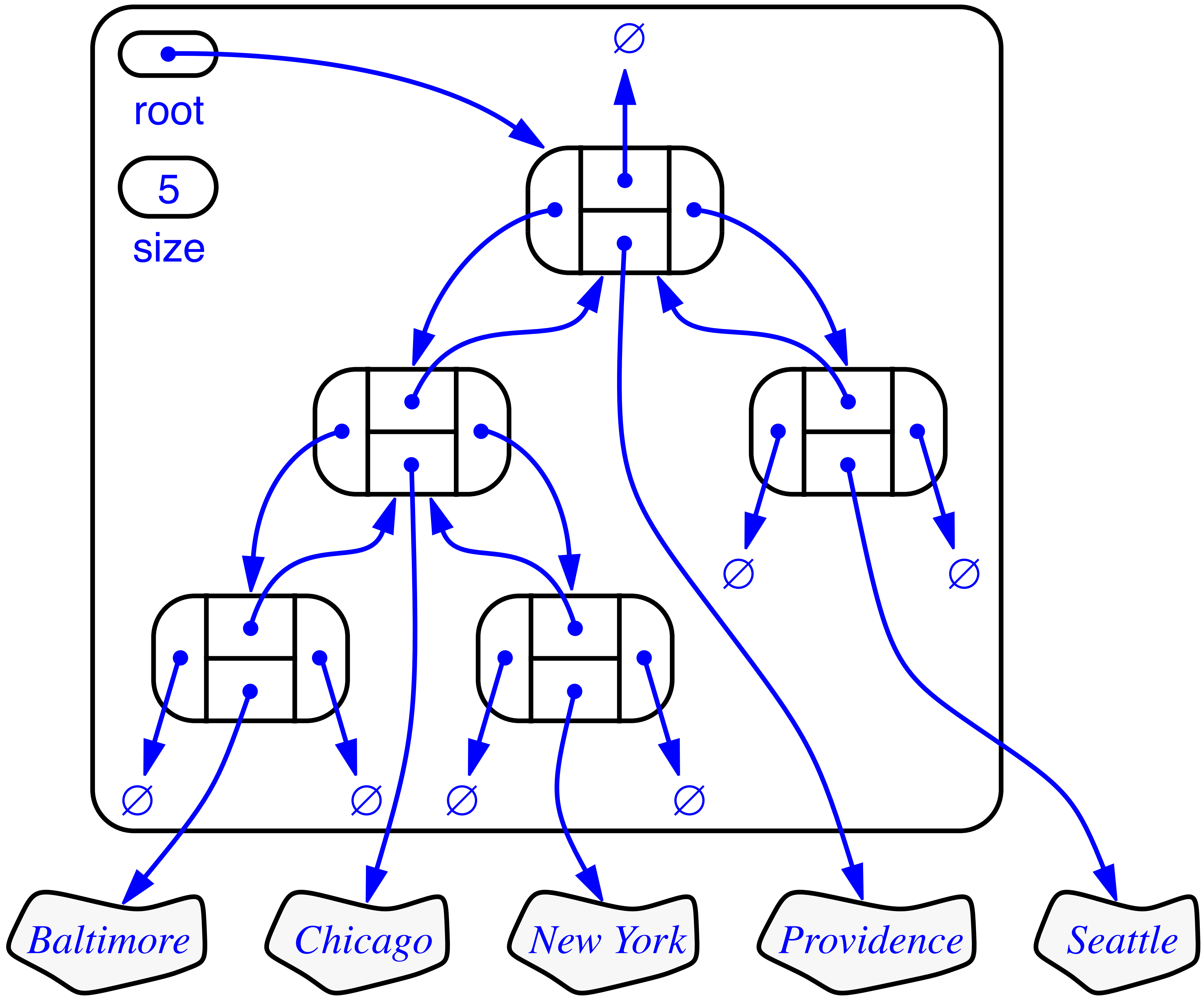
$((((3 \times (1 + (4 + 6))) + (2 + 8)) \times 5) + (4 \times (7 + 2)))$

Properties of Binary Trees

- (# external nodes) = (# internal nodes) + 1
- (# nodes at level i) $\leq 2^{i}$
- (# external nodes) $\leq 2^{(\text{height})}$
- (height) $\geq \log_2$ (# external nodes)
- (height) $\geq \log_2$ (# nodes) – 1
- (height) $\leq$ (# internal nodes) = ((# nodes) – 1)/2

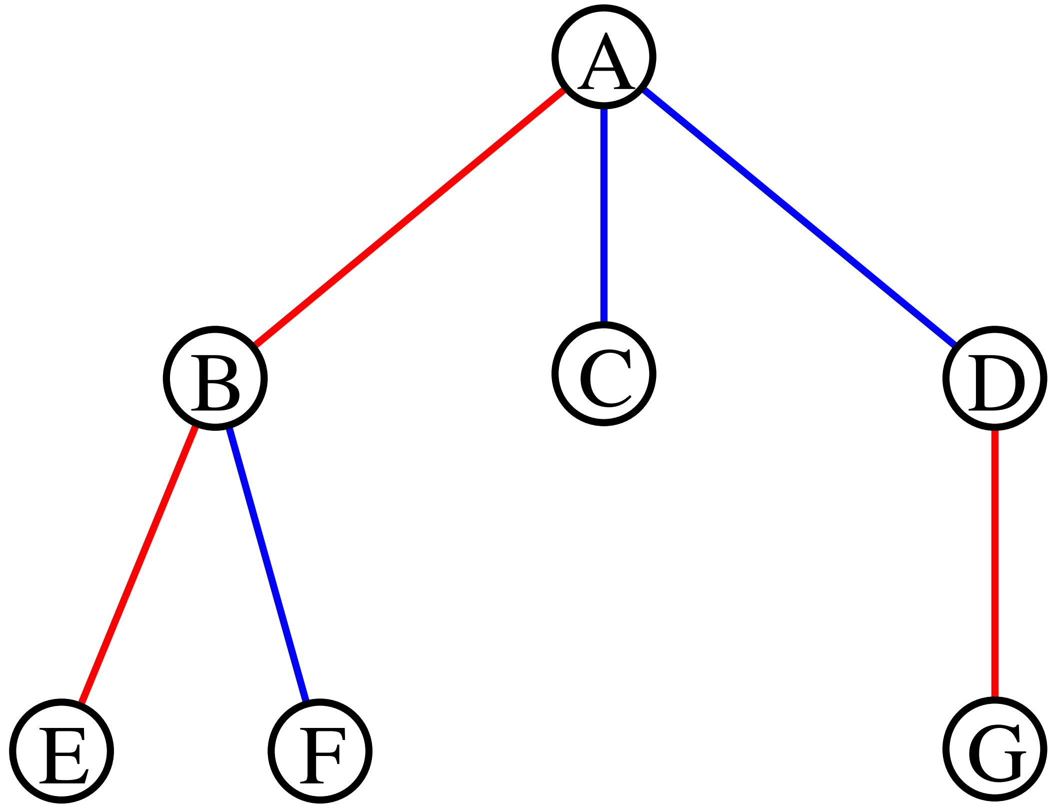


Linked Data Structure for Binary Trees



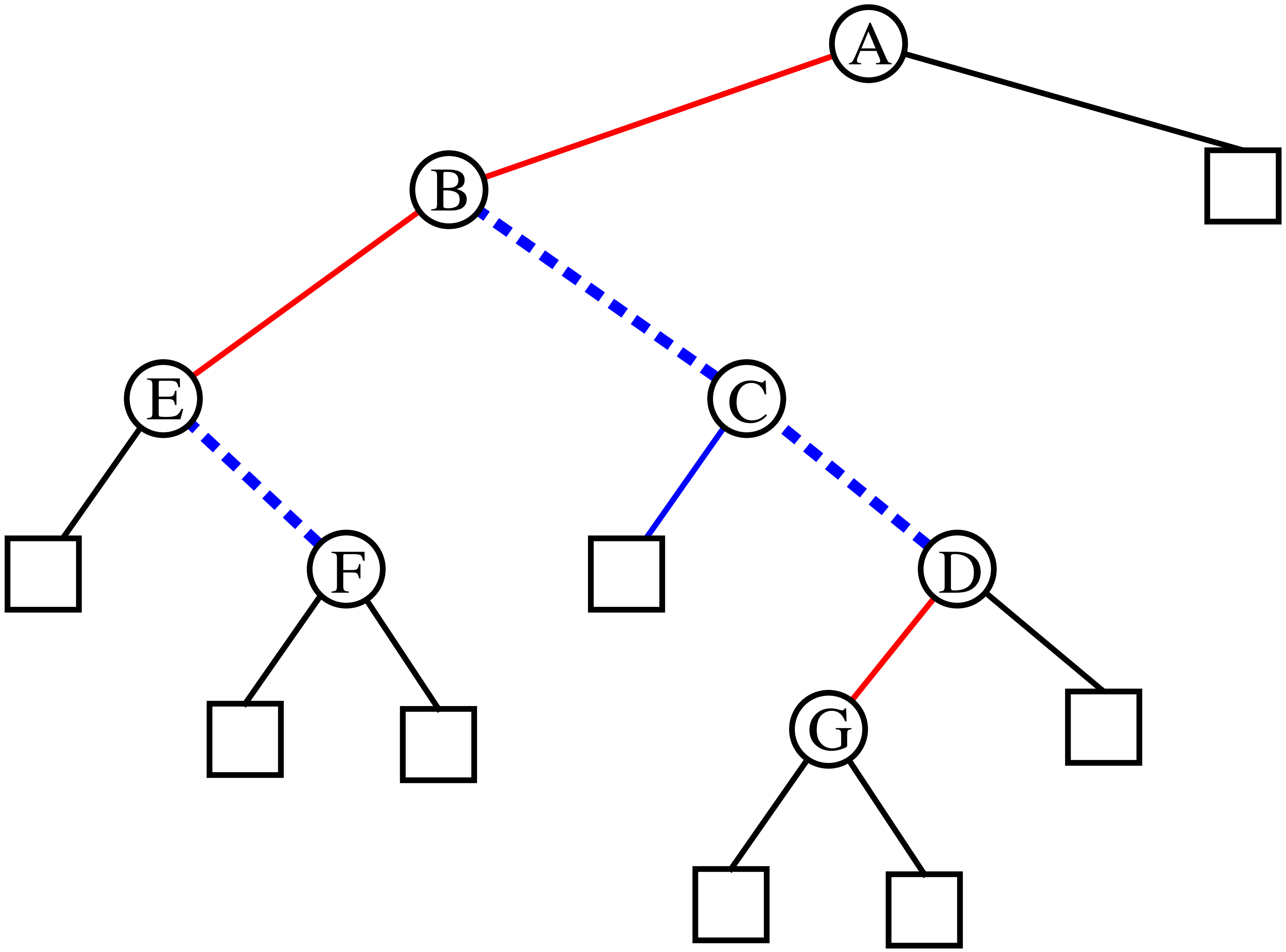
Representing General Trees

- tree T



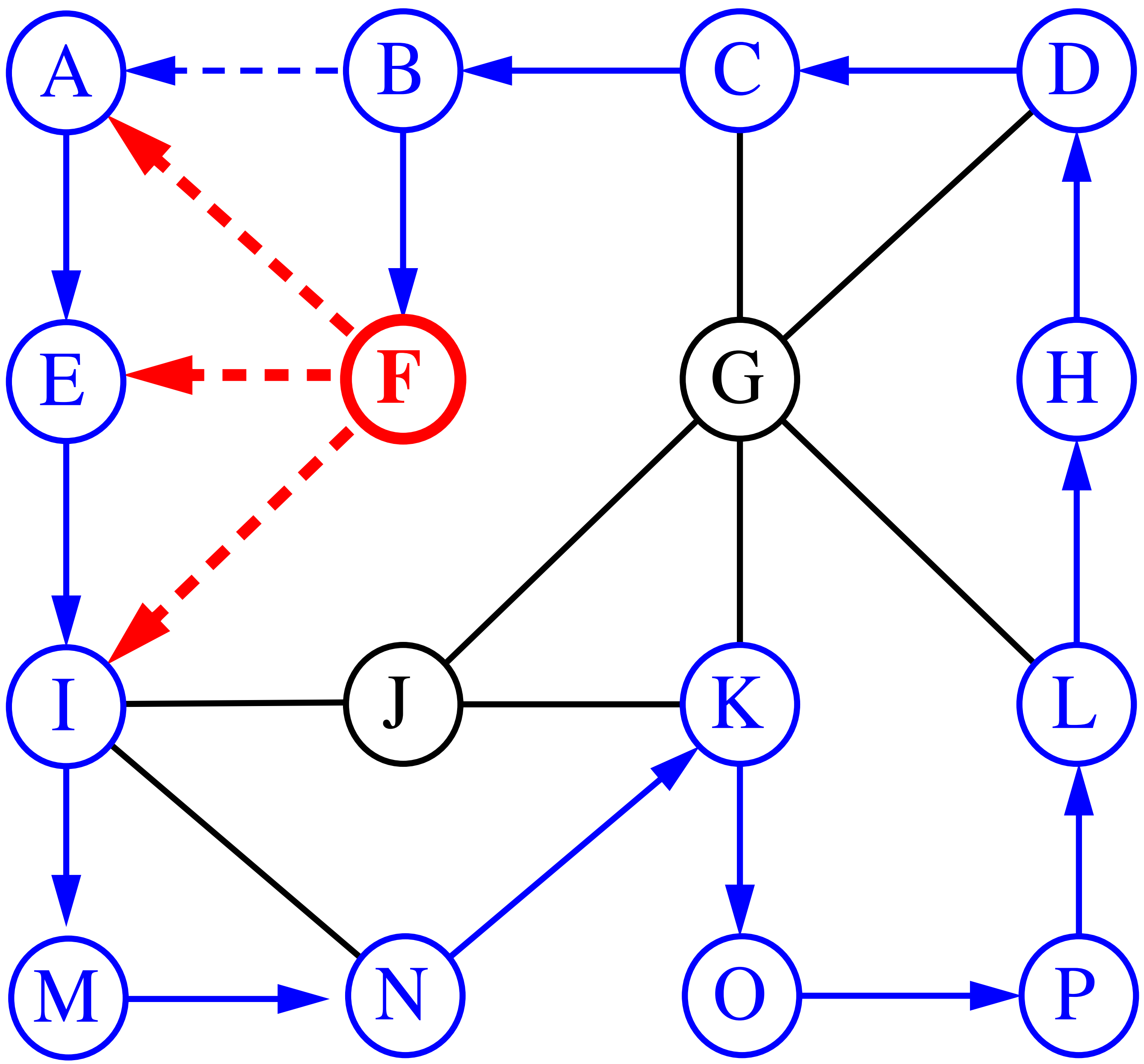
Representing General Trees

- binary tree T' representing T



Depth-First Search

- Graph Traversals
- Depth-First Search



Depth-First Search

Algorithm DFS(v);
  Input: A vertex v in a graph
  Output: A labeling of the edges as "discovery" edges and "backedges"
  for each edge e incident on v **do**
    if edge e is unexplored **then**
      let w be the other endpoint of e
      if vertex w is unexplored **then**
        label e as a discovery edge
        recursively call **DFS**(w)
      else
      label e as a backedge

DFS Properties

- Proposition 9.12 : Let G be an undirected graph on which a DFS traversal starting at a vertex s has been preformed. Then:
  1) The traversal visits all vertices in the connected component of s
  2) The discovery edges form a spanning tree of the connected component of s

- Justification of 1):
  - Let's use a contradiction argument: suppose there is at least on vertex v not visited and let w be the first unvisited vertex on some path from s to v.
  - Because w was the first unvisited vertex on the path, there is a neighbor u that has been visited.
  - But when we visited u we must have looked at edge(u, w). Therefore w must have been visited.
  - and justification

DFS Properties

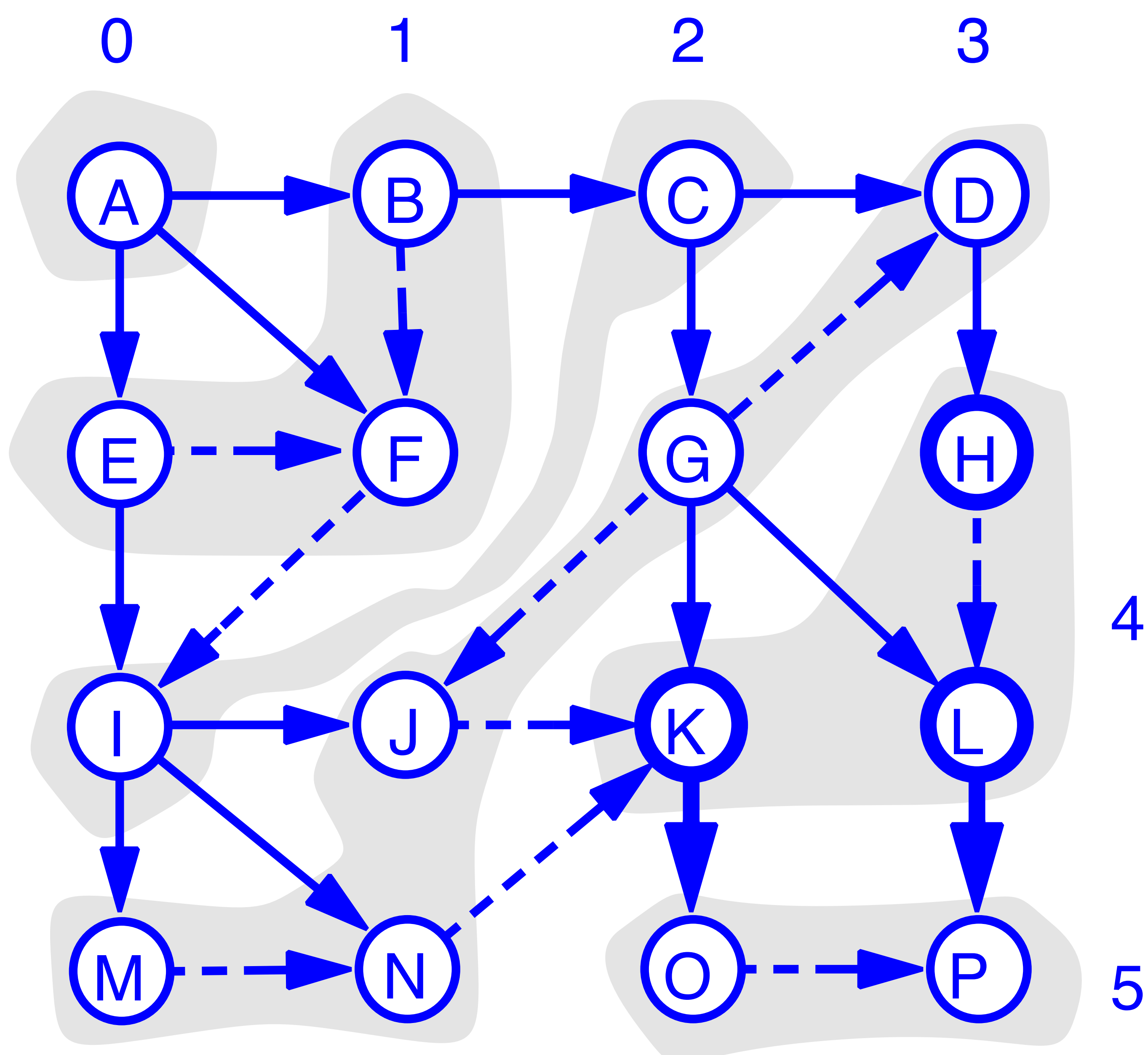
- Proposition 9.12 : Let G be an undirected graph on which a DFS traversal starting at a vertex *s* has been preformed. Then:
    1) The traversal visits all vertices in the connected component of *s*
    2) The discovery edges form a spanning tree of the connected component of *s*

- Justification of 2):
    - We only mark edges from when we go to unvisited vertices. So we never form a cycle of discovery edges, i.e. discovery edges form a tree.
    - This is a spanning tree because DFS visits each vertex in the connected component of *s*

Running Time Analysis

- Remember:
  - DFS is called on each vertex exactly once.
  - Every edge is examined exactly twice, once from each of its vertices
- For n_s vertices and m_s edges in the connected component of the vertex s, a DFS starting at s runs in $O(n_s + m_s)$ time if:
  - The graph is represented in a data structure, like the adjacency list, where vertex and edge methods take constant time
  - Marking a vertex as explored and testing to see if a vertex has been explored takes O(degree)
  - By marking visited nodes, we can systematically consider the edges incident on the current vertex so we do not examine the same edge more than once.

Breadth-First Search



Breadth-First Search

- Like DFS, a Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so definesaspanningtreewithseveralusefulproperties

  - The starting vertex s has level 0, and, as in DFS, defines that point as an “anchor.”
  - In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited.
  - These edges are placed into level 1
  - In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
  - This continues until every vertex has been assigned a level.
  - The label of any vertex v corresponds to the length of the shortest path from s to v.

BFS Pseudo-Code

Algorithm BFS(s):

Input: A vertex s in a graph

Output:Alabelingoftheedgesas"discovery"edges and "cross edges"

```
initialize container L0 to contain vertex s
i ← 0
while Li is not empty do
  create container Li+1 to initially be empty
  for each vertex v in Li do
    for eachedge  e incident on v do
      if edge e is unexplored then
        let w be the other endpoint of e
        if vertex w is unexplored then
          label e as a discovery edge
          insert w into Li+1
        else
          label e as a cross edge
  i ← i + 1
```

Properties of BFS

- Proposition: Let G be an undirected graph on which a BFS traversal starting at vertex s has been performed. Then

  -The traversal visits all vertices in the connected component of s.

  -The discovery-edges form a spanning tree T, which we call the BFS tree, of the connected component of s

  -For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G between s and v has at least i edges.

  - If (u, v) is an edge that is not in the BFS tree, then the level numbers of u and v differ by at most one.

Properties of BFS

- Proposition: Let G be a graph with n vertices and m edges. A BFS traversal of G takes time $O(n + m)$. Also, there exist $O(n + m)$ time algorithms based on BFS for the following problems:

-Testing whether G is connected.

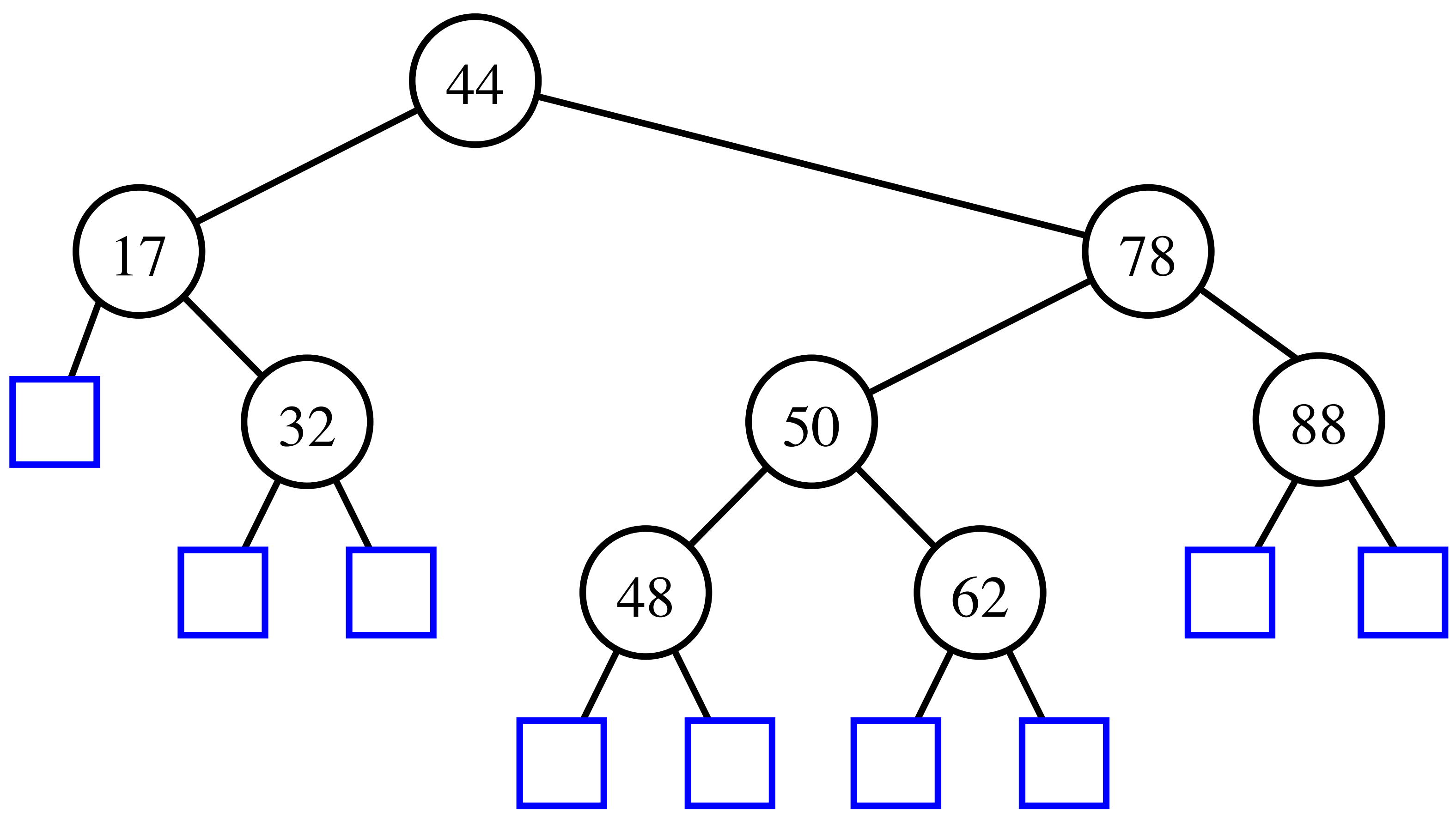
-Computing a spanning tree of G

-Computing the connected components of G

-Computing, for every vertex v of G, the minimum number of edges of any path between s and v.

SEARCHING

- the dictionary ADT
- binary search trees



The Dictionary ADT

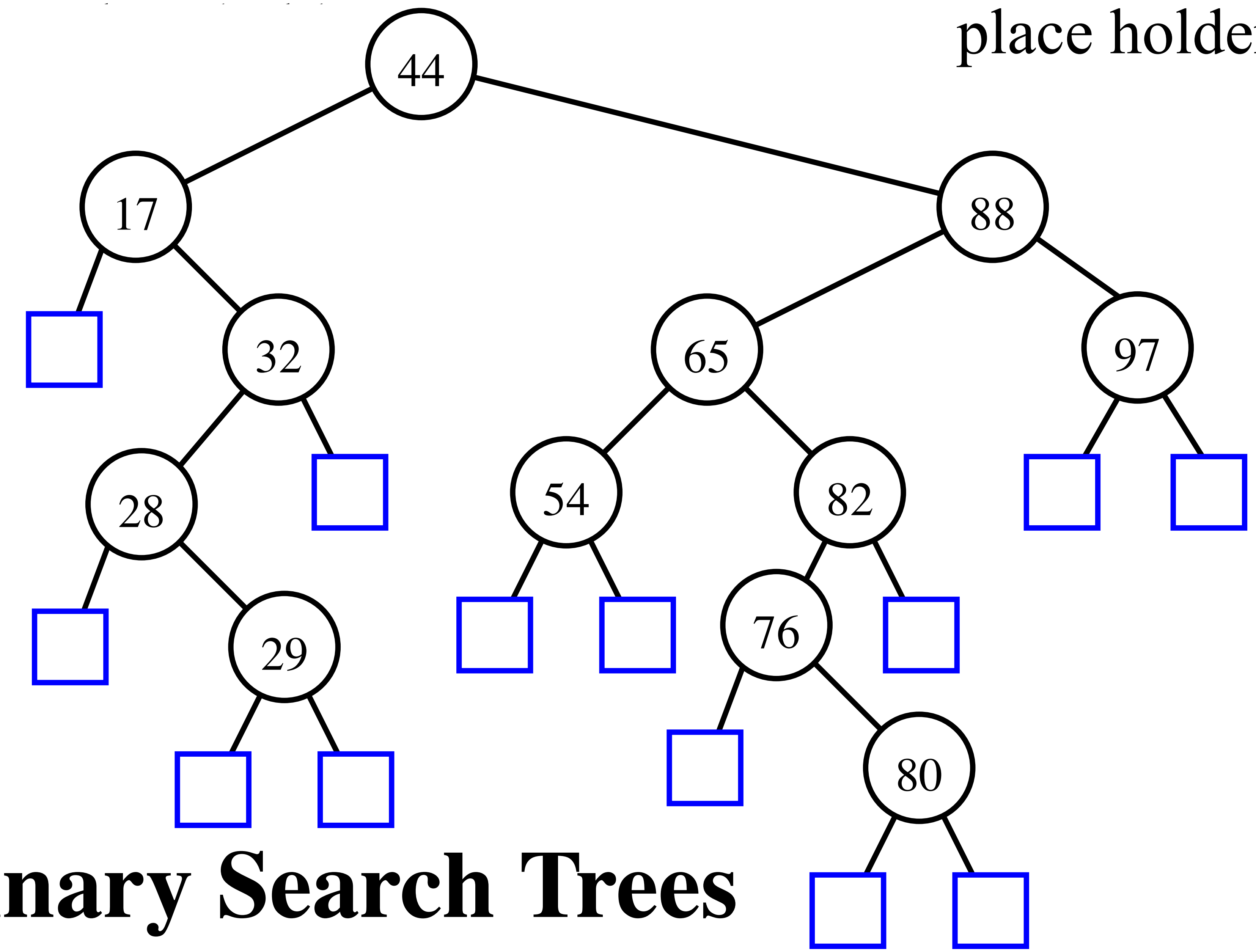
- a dictionary is an abstract model of a database
- like a priority queue, a dictionary stores key-element pairs
- the main operation supported by a dictionary is searching by key
- simple container methods:
  - size()
  - isEmpty()
  - elements()

The Dictionary ADT

- query methods:
  - findElement(k)
  - findAllElements(k)
- update methods:
  - insertItem(k, e)
  - removeElement(k)
  - removeAllElements(k)
- special element
  - NO_SUCH_KEY, returned by an unsuccessful search

[redacted]

- each internal node stores an item (k, e) of a dictionary.
- keys stored at nodes in the left subtree of v are less than or equal to k.
- keys stored at nodes in the right subtree of v are greater than or equal to k.
- external nodes do not hold elements but serve as place holders.



Binary Search Trees

Search

- A binary search tree *T* is a ***decision tree***, where the question asked at an internal node *v* is whether the search key *k* is less than, equal to, or greater than the key stored at *v*.

Algorithm TreeSearch(*k*, *v*):

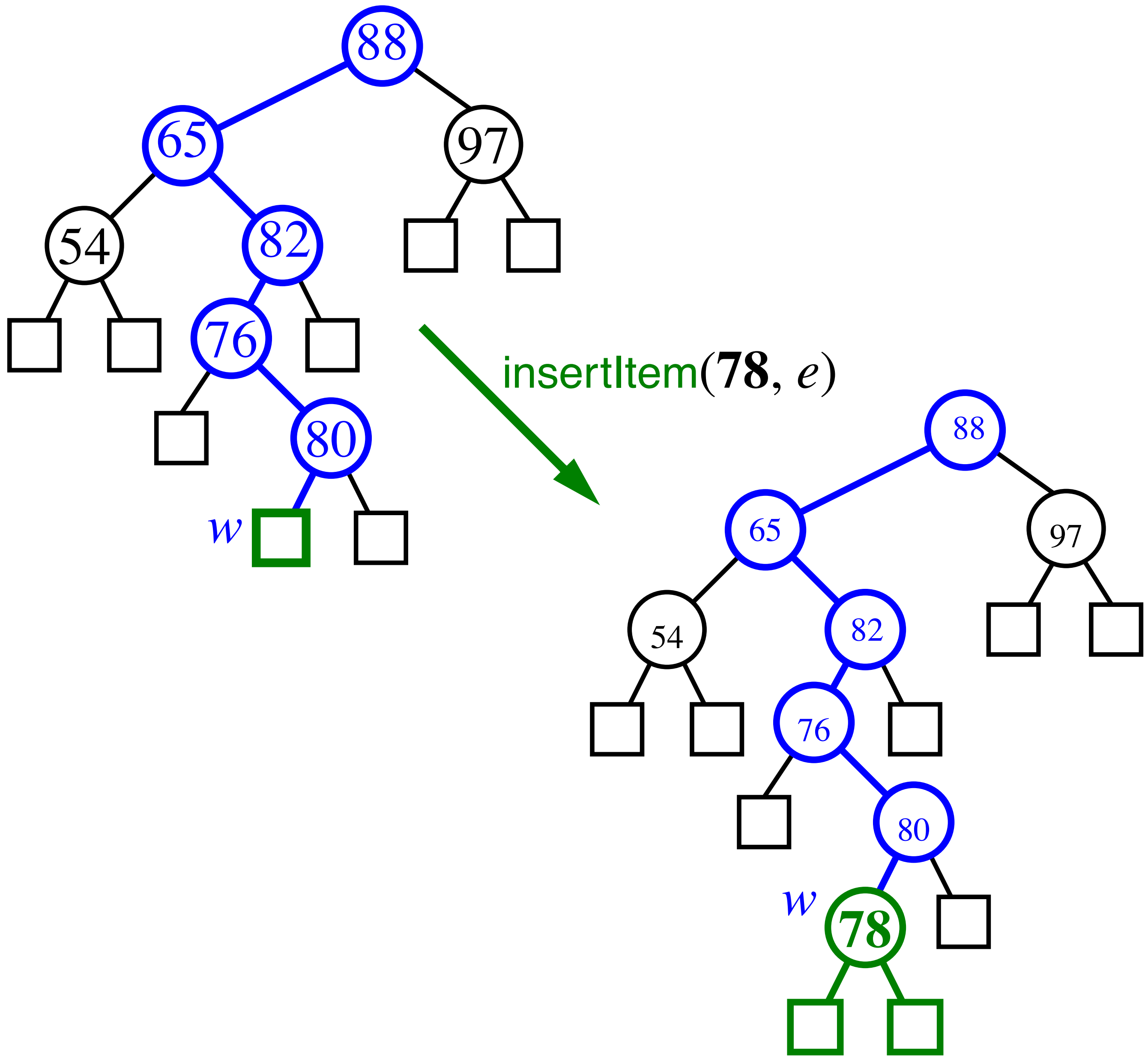
Input: A search key *k* and a node *v* of a binary search tree *T*.

Ouput: A node w of the subtree *T*(*v*) of *T* rooted at *v*,

```
if v is an external node then
    return v
if k = key(v) then
    return v
else if k < key(v) then
    return TreeSearch(k, T.leftChild(v))
else
    { k > key(v) }
    return TreeSearch(k, T.rightChild(v))
```

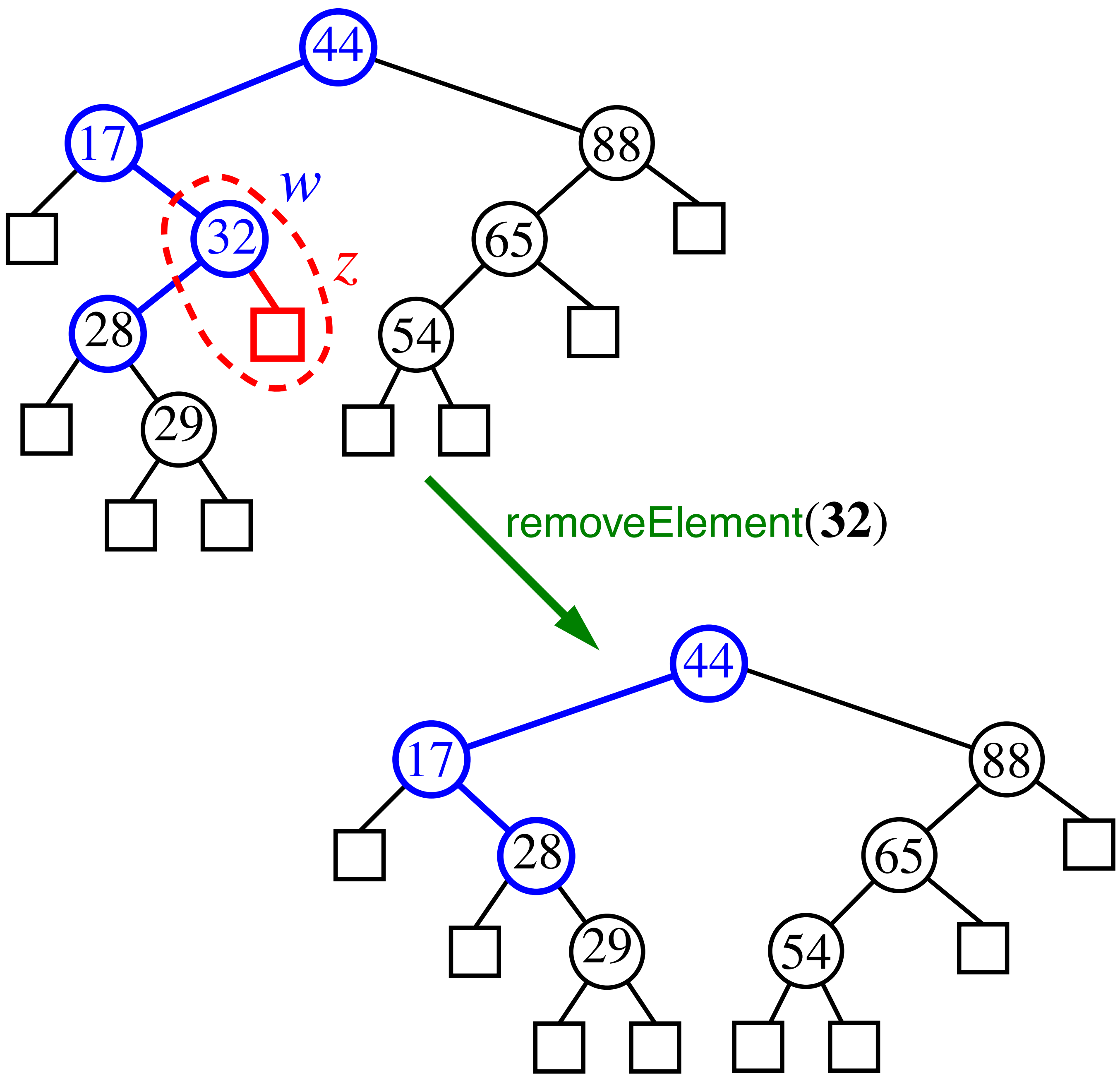
Insertion

- To perform insertItem(k, e), let w be the node returned by TreeSearch(k, T.root())
- If w is external, we know that k is not stored in T. We call expandExternal(w) on T and store (k, e) in w



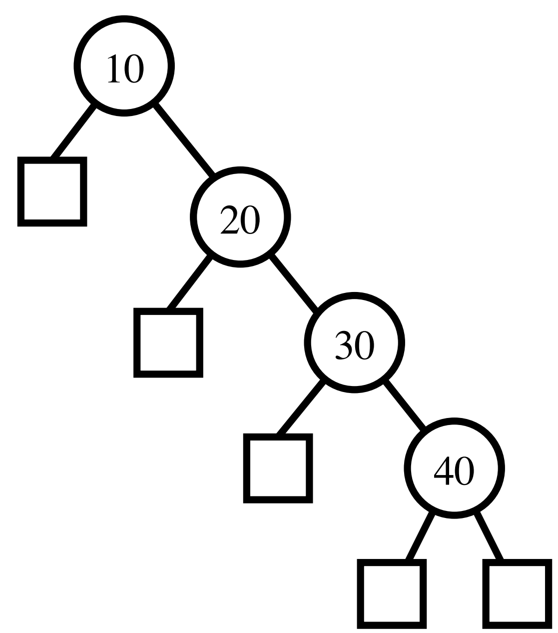
Removal I

- We locate the node w where the key is stored with algorithm TreeSearch
- If w ***has an external child*** z, we remove w and z with removeAboveExternal(z)



Time Complexity

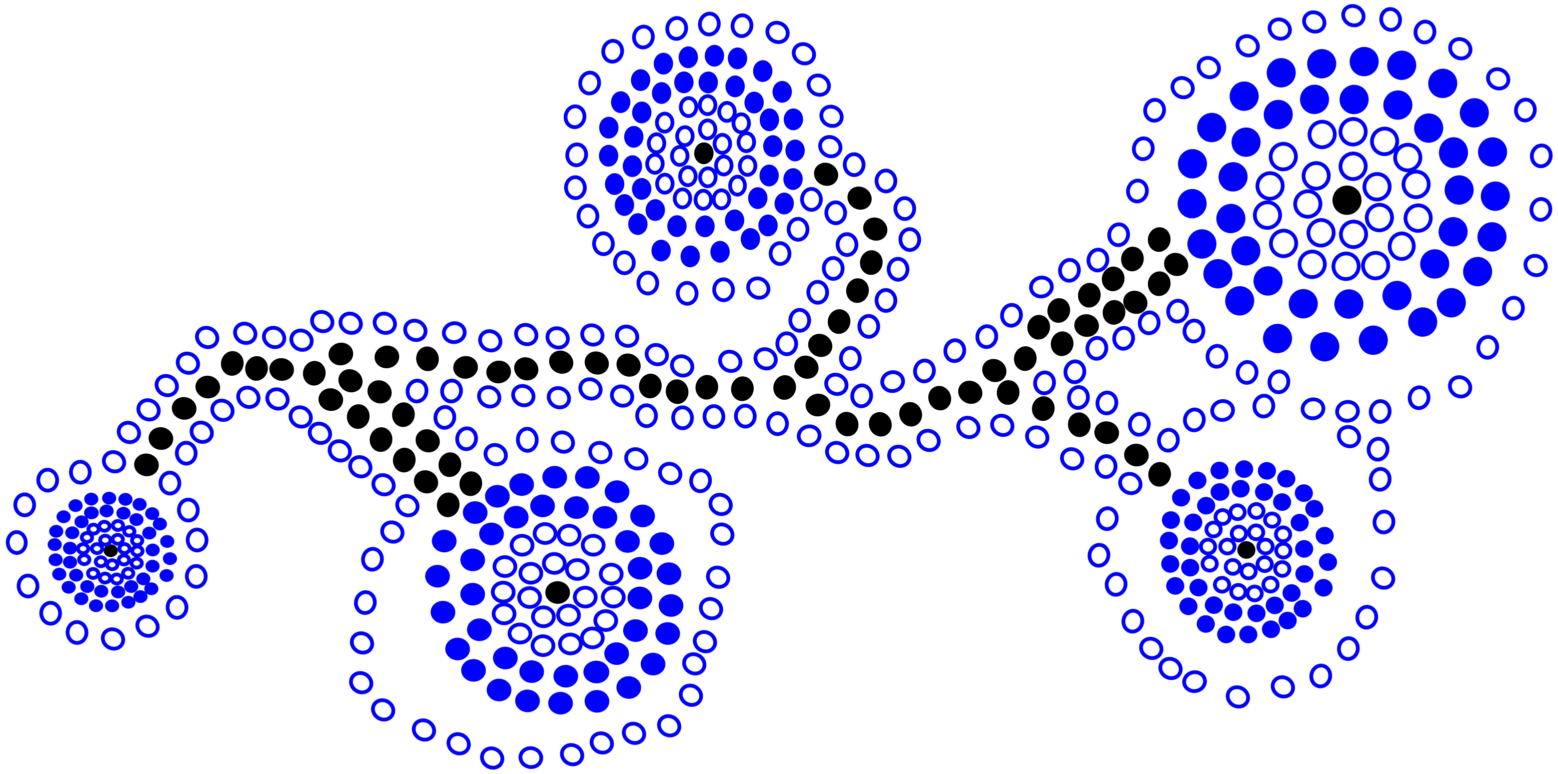
- A search, insertion, or removal, visits the nodes along a ***root-to leaf path***, plus possibly the ***siblings*** of such nodes
- Time O(1) is spent at each node
- The running time of each operation is O(h), where h is the height of the tree
- The height of binary serch tree is in n in the worst case, where a binary search tree looks like a sorted sequence



- To achive good running time, we need to keep the tree ***balanced***, i.e., with O(log n) height

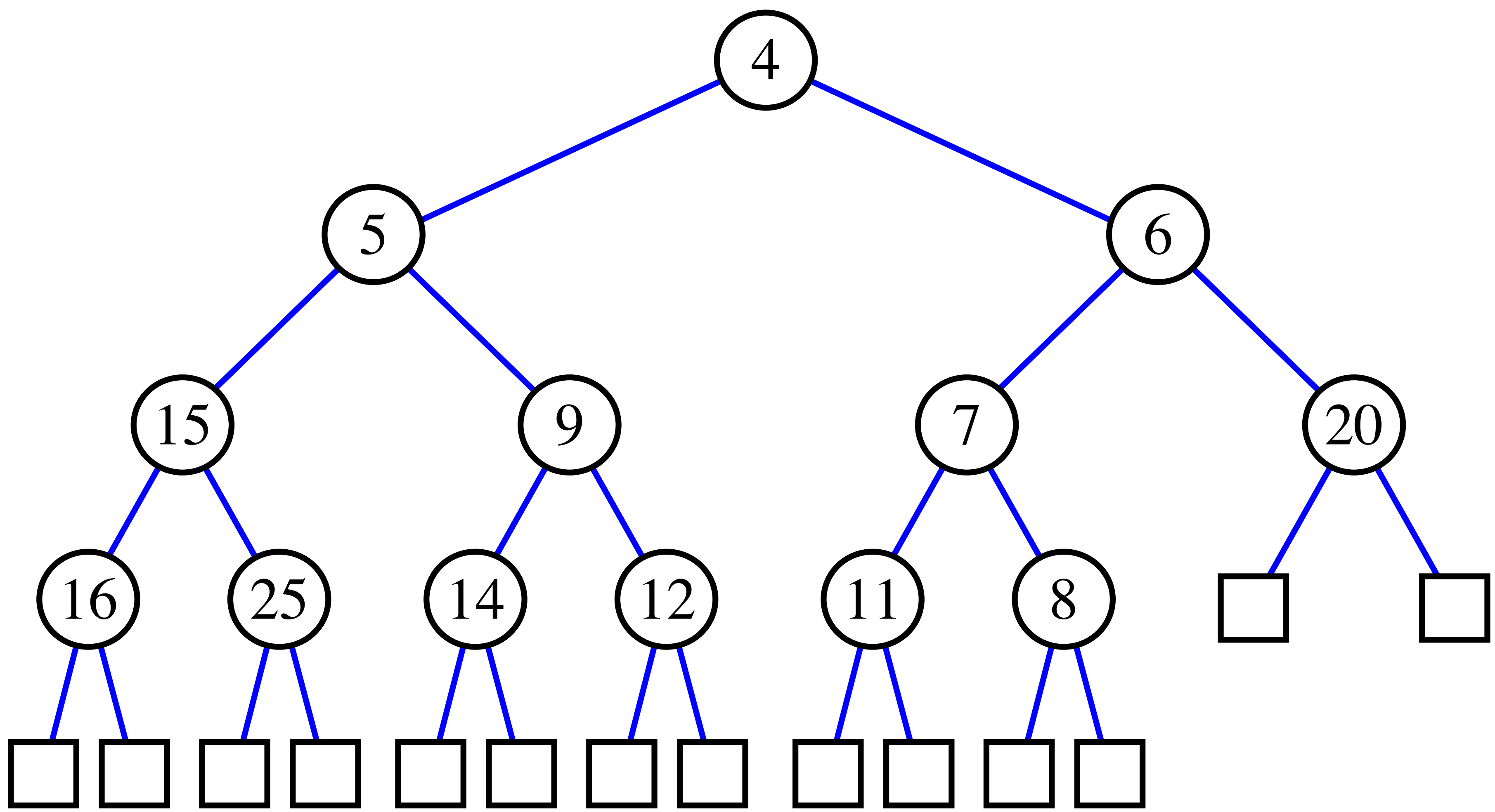
HEAPS I

- Heaps
- Properties
- Insertion and Deletion



Heaps

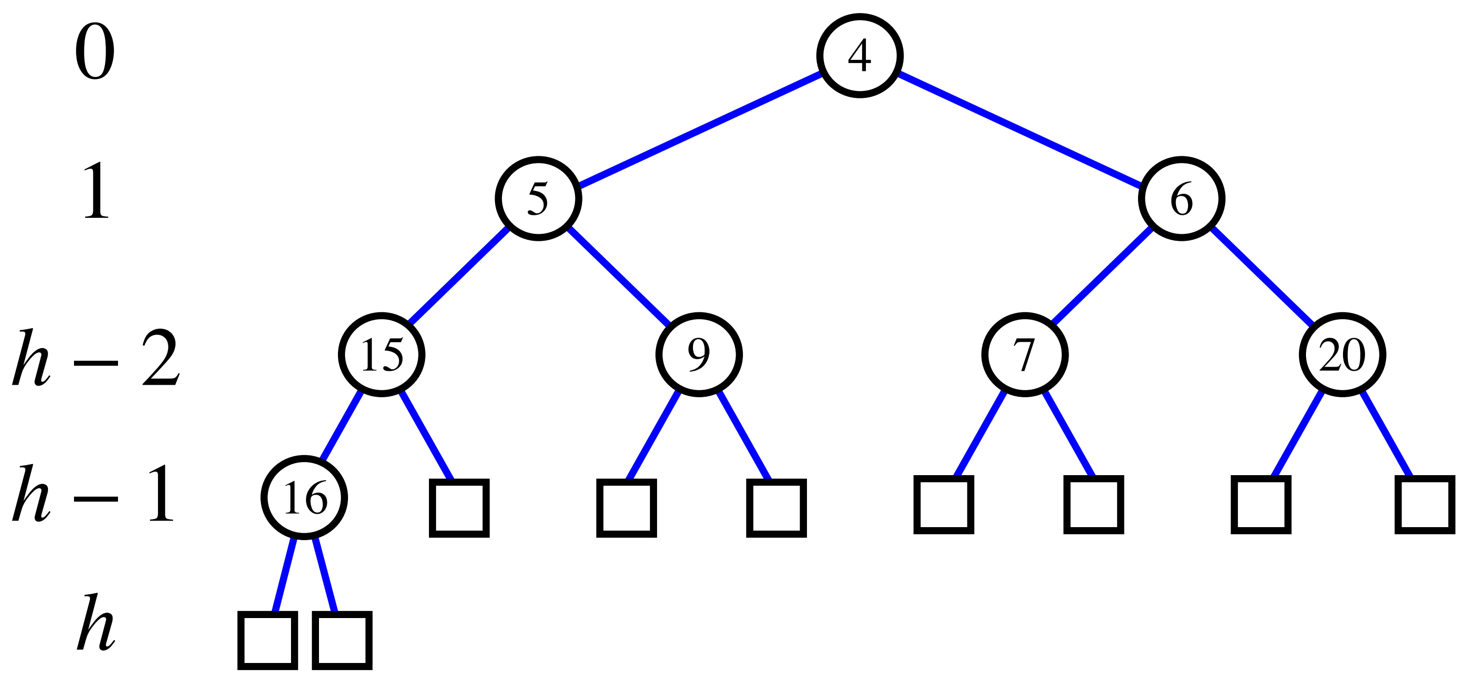
- A ***heap*** is a binary tree *T* that stores a collection of keys (or key-element pairs) at its internal nodes and that satisfies two additional properties:
  - ***Order Property:*** key(parent) ≤ key(child)
  - ***Structural Property***: all levels are full, except the last one, which is left-filled (***complete binary tree***)



Height of a Heap

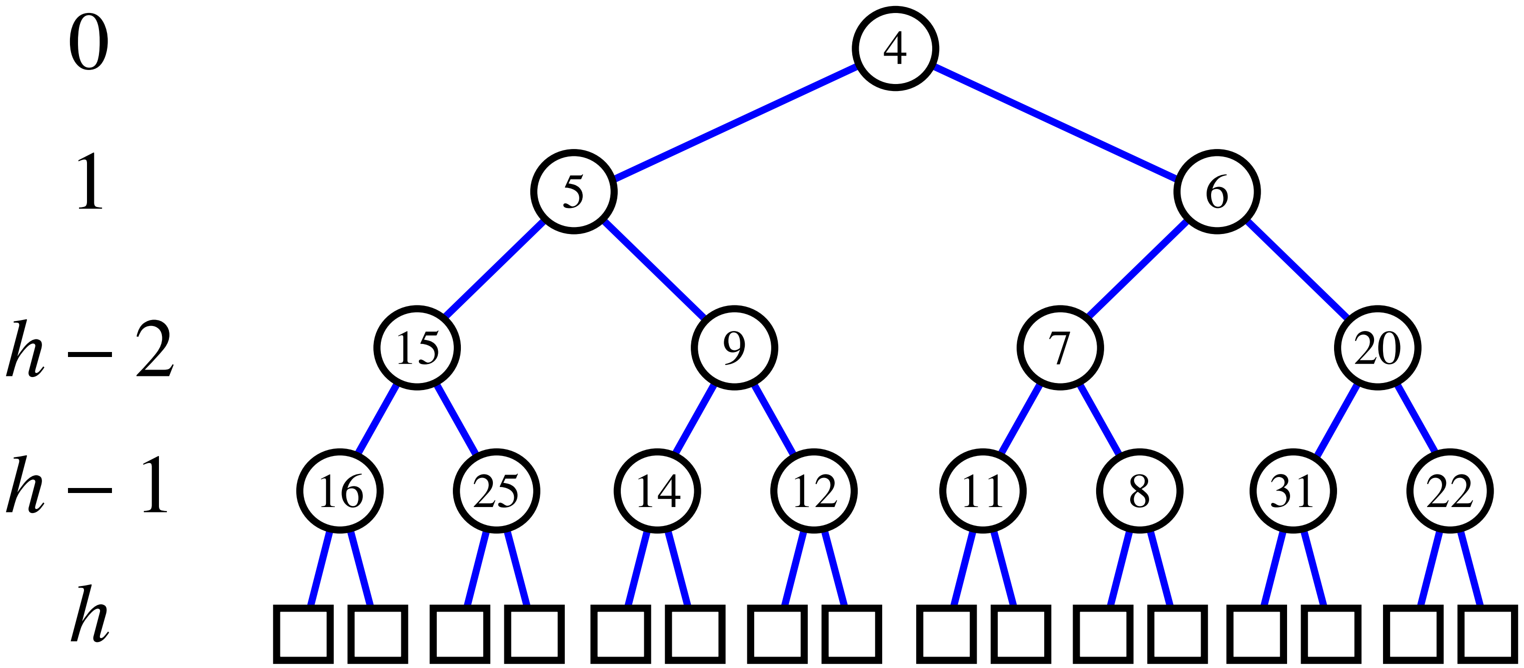
A heap T storing n keys has height $h = \lceil \log(n + 1) \rceil$, which is O(log n)

- $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$



Height of a Heap

- $n \leq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$

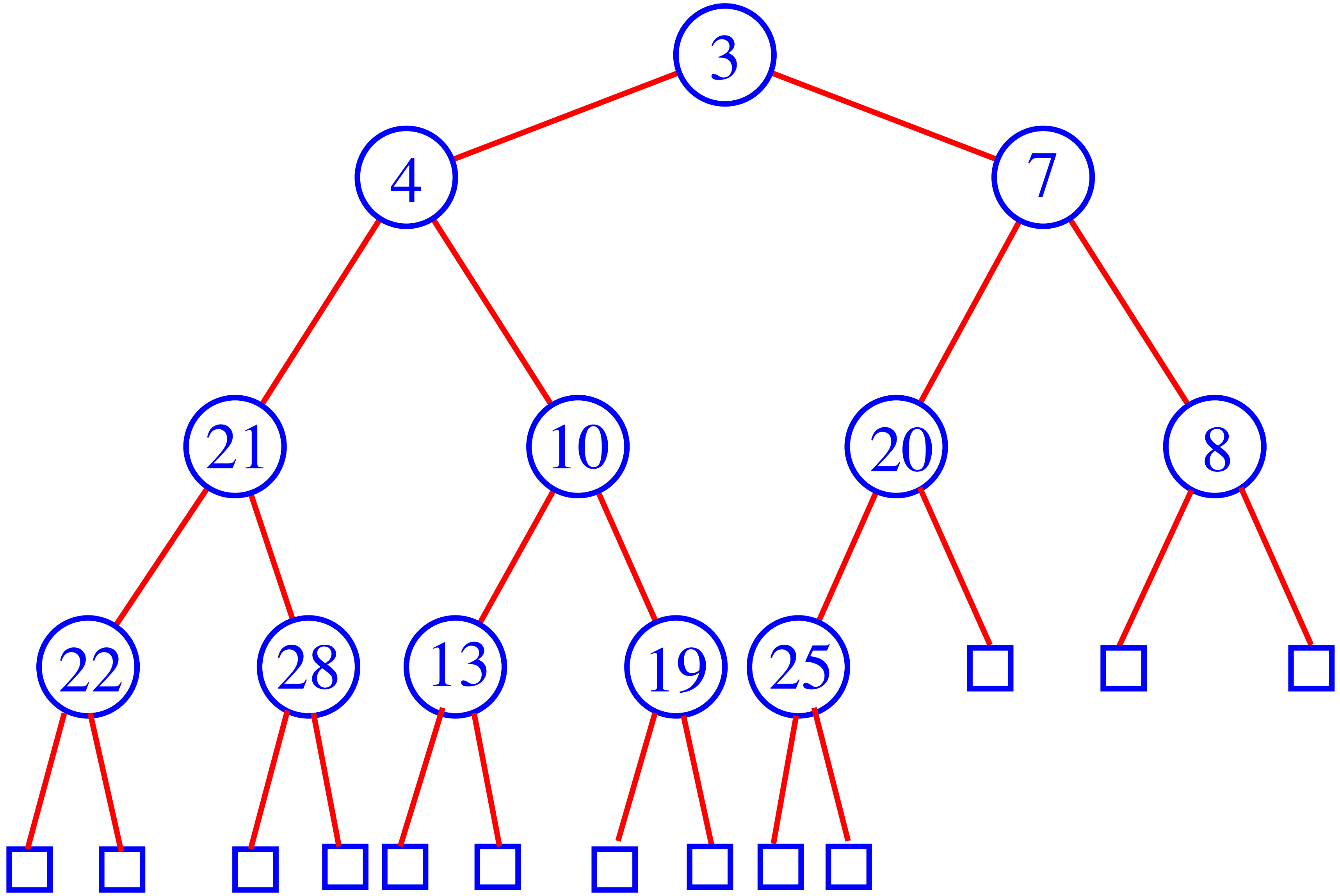


- Therefore $2^{h-1} \leq n \leq 2^h - 1$
- Taking logs, we get $\log(n + 1) \leq h \leq \log n + 1$
- Which implies $\boldsymbol{h = \lceil \log(n+1) \rceil}$

Heap Insertion

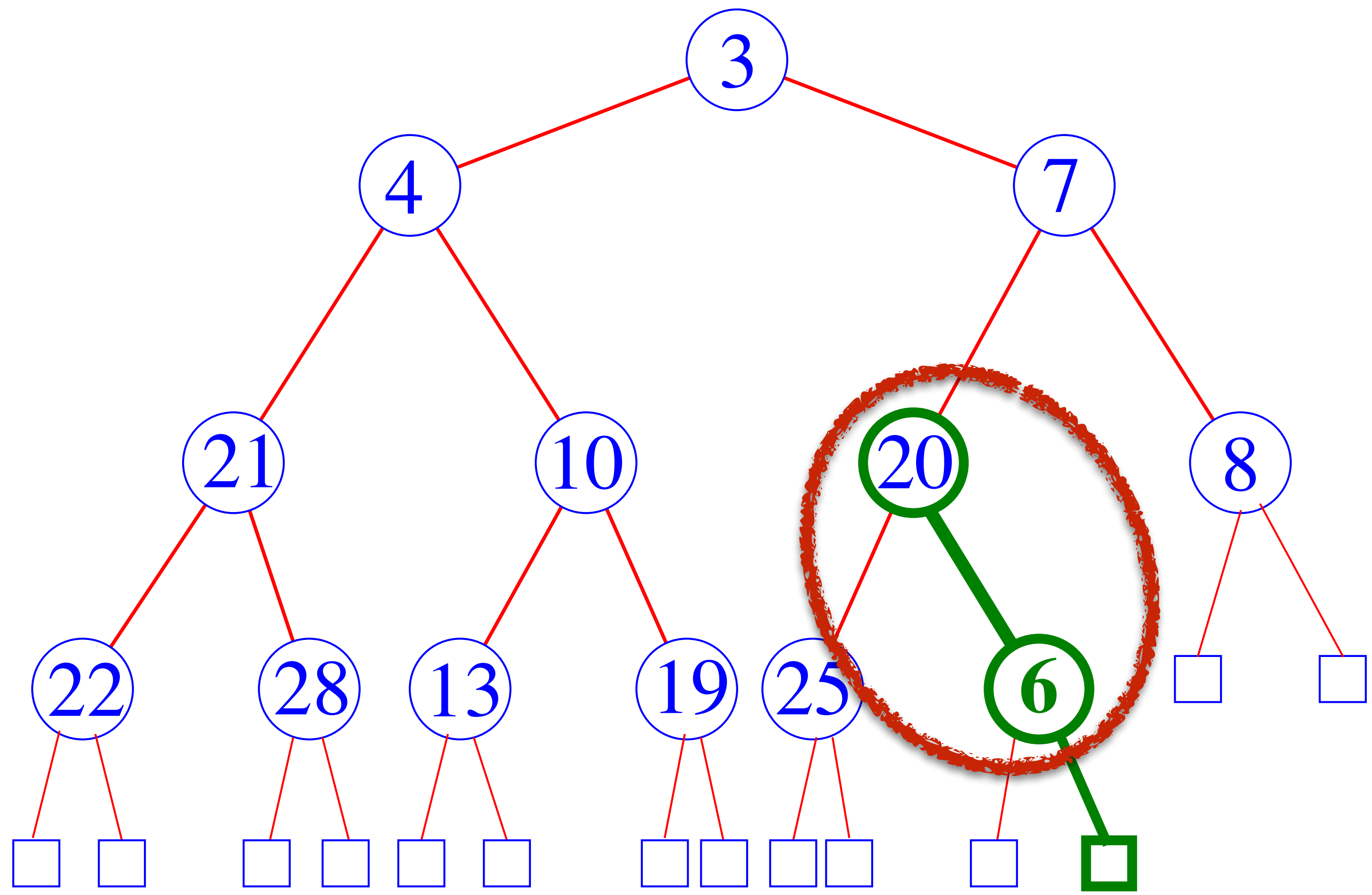
So here we go ...

The key to insert is **6**



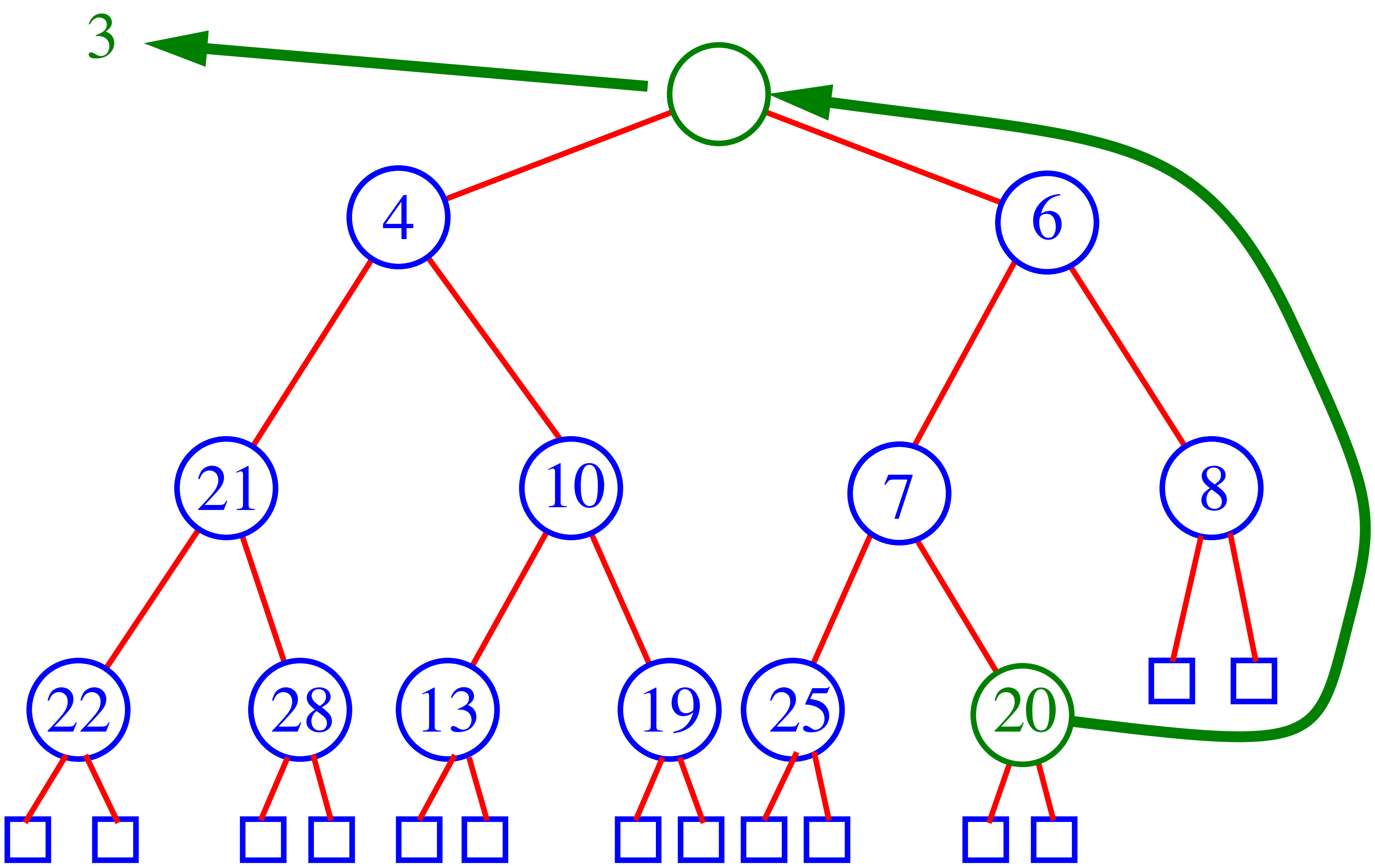
Upheap

- ***Swap parent-child keys out of order***

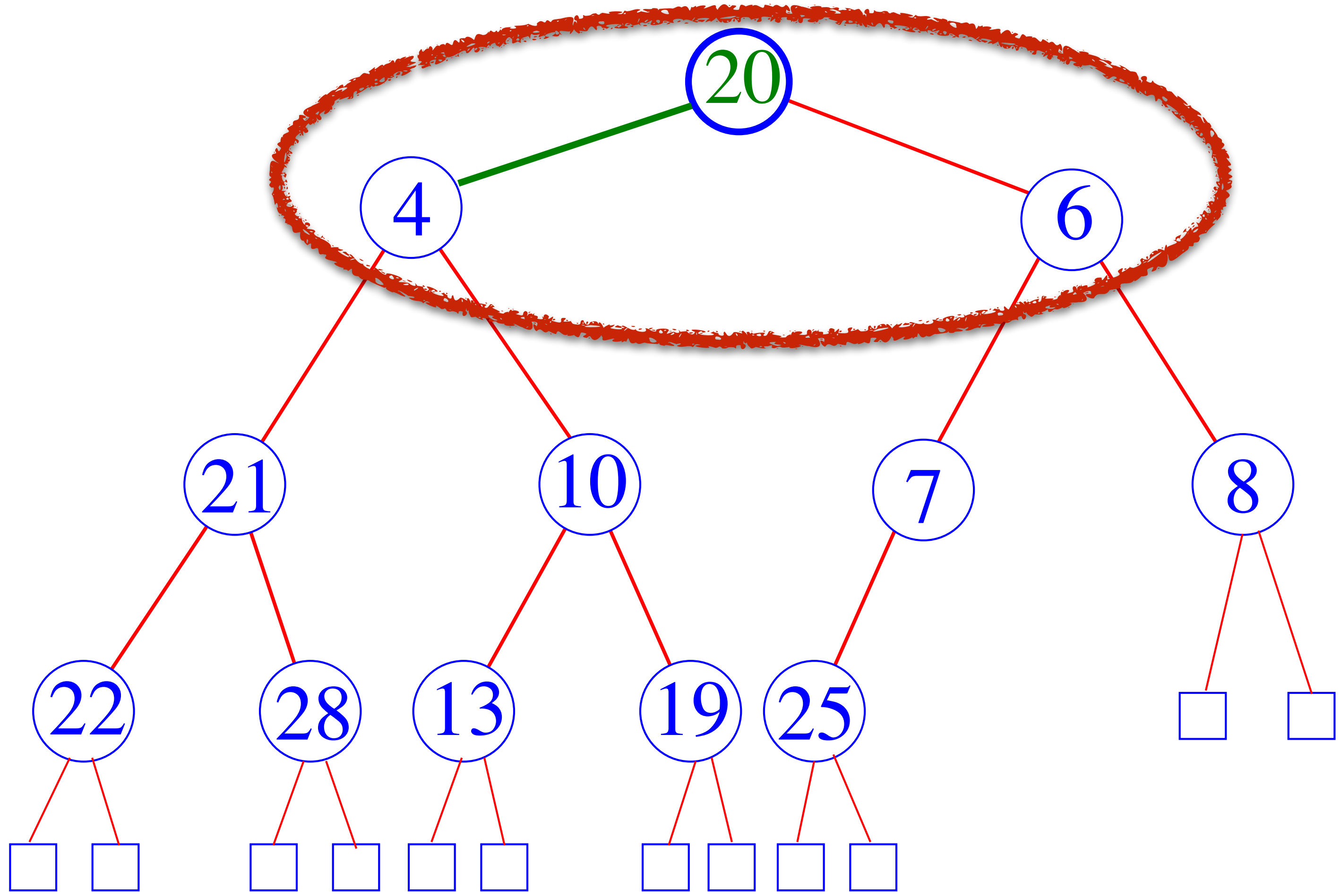


Removal From a Heap

RemoveMin()

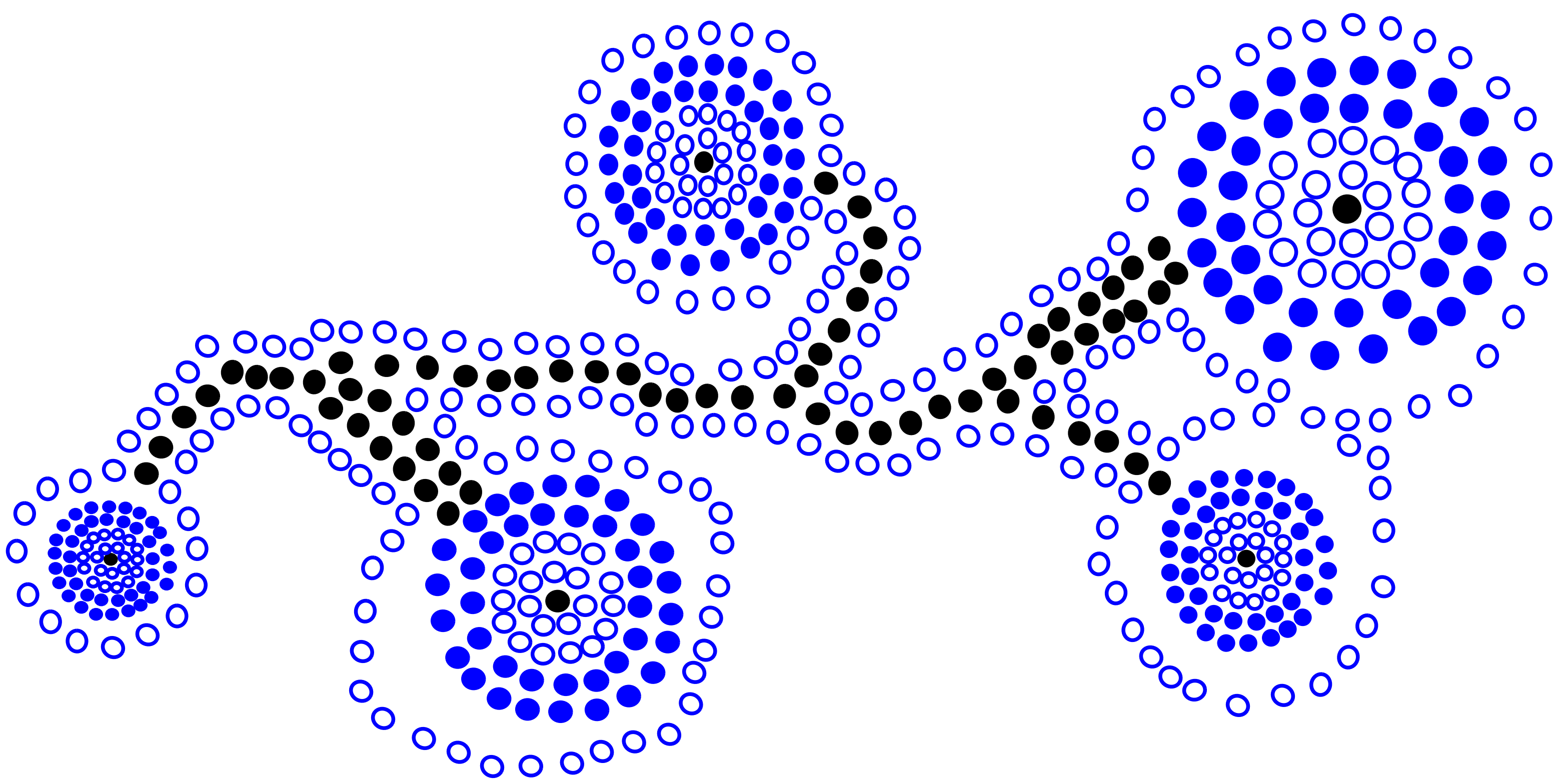


Downheap



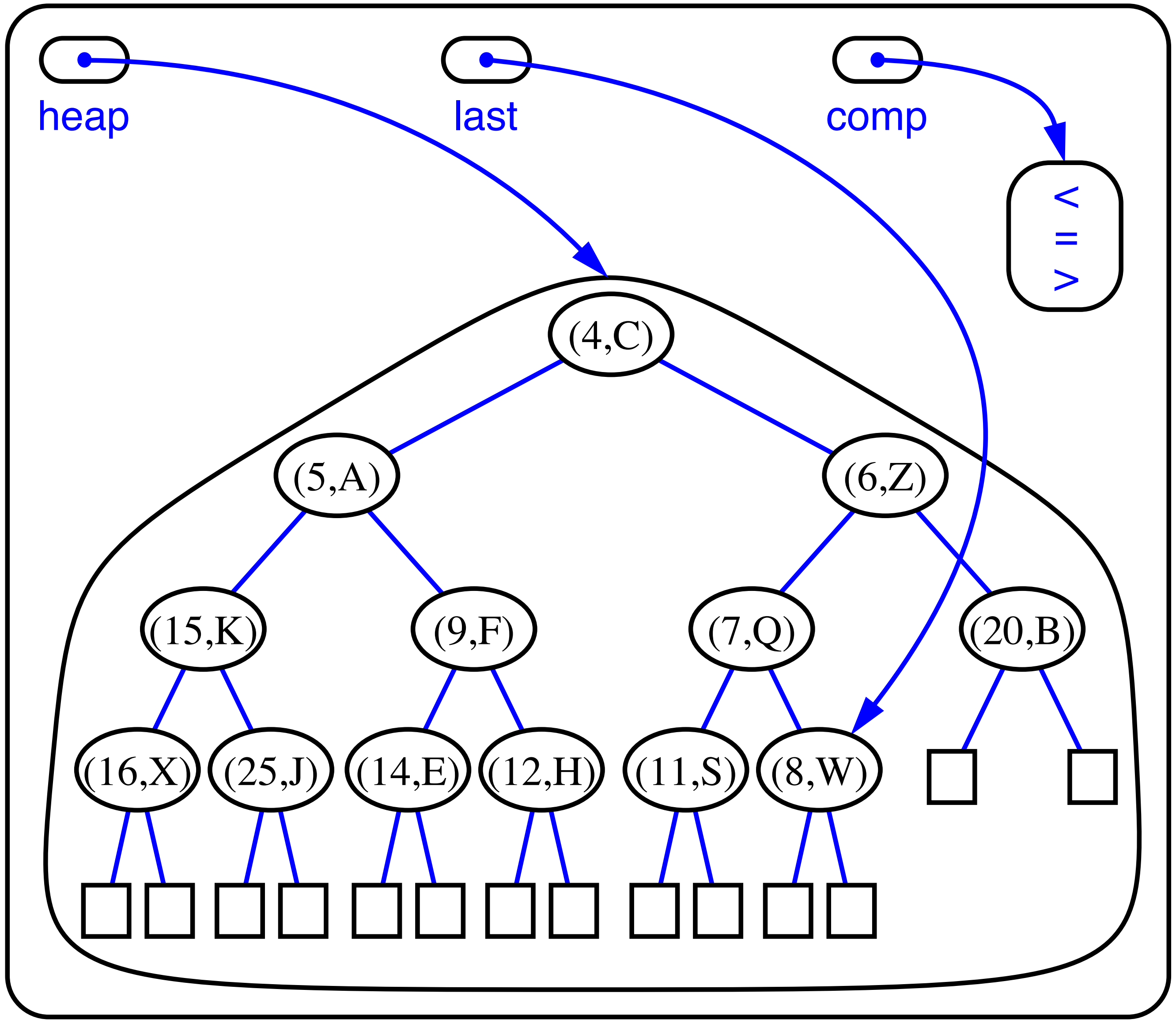
HEAPS II

- Implementation
- HeapSort
- Bottom-Up Heap Construction
- Locators



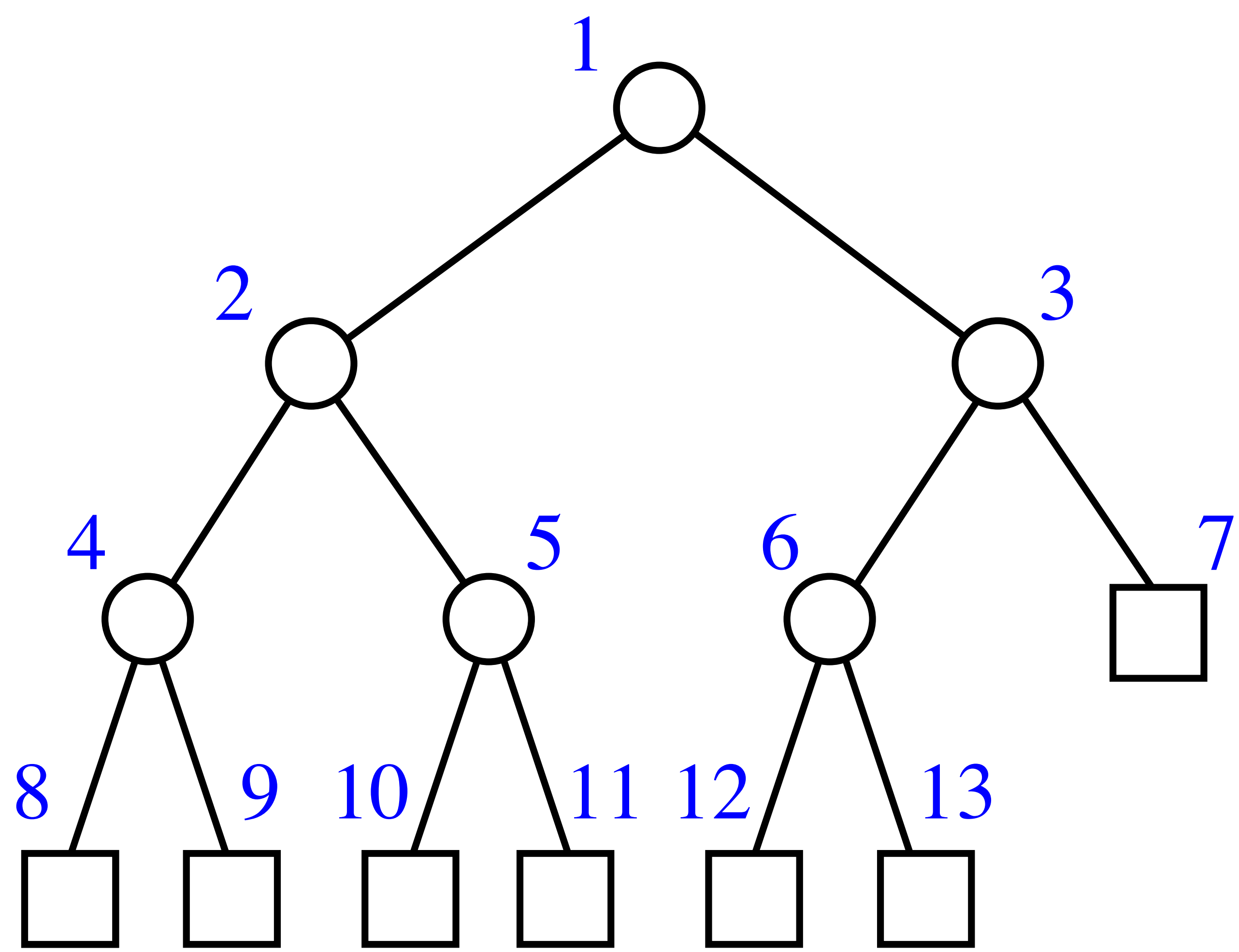
Implementation of a Heap

```
public class HeapPriorityQueue implements PriorityQueue
{
   BinaryTree T;
   Position last;
   Comparator comparator;
   ...
}
```



Vector Based Implementation

- Updates in the underlying tree occur only at the "last element"
- A heap can be represented by a vector, where the node at rank i has
  - left child at rank $2i$ and
  - right child at rank $2i + 1$



Heap Sort

- All heap methods run in logarithmic time or better
- If we implement PriorityQueueSort using a heap for our priority queue, insertItem and removeMin each take O(log k), k being the number of elements in the heap at a given time.
- We always have at most n elements in the heap, so the worst case time complexity of these methods is O(log n).
- Thus each phase takes O(n log n) time, so the algorithm runs in O(n log n) time also.
- This sort is known as ***heap-sort***.
- The O(n log n) run time of heap-sort is much better than the O(n^2) run time of selection and insertion sort.

In-Place Heap-Sort

- Do not use an external heap
- Embed the heap into the sequence, using the vector representation

Bottom-Up Heap Construction

- build $(n + 1)/2$ trivial one-element heaps

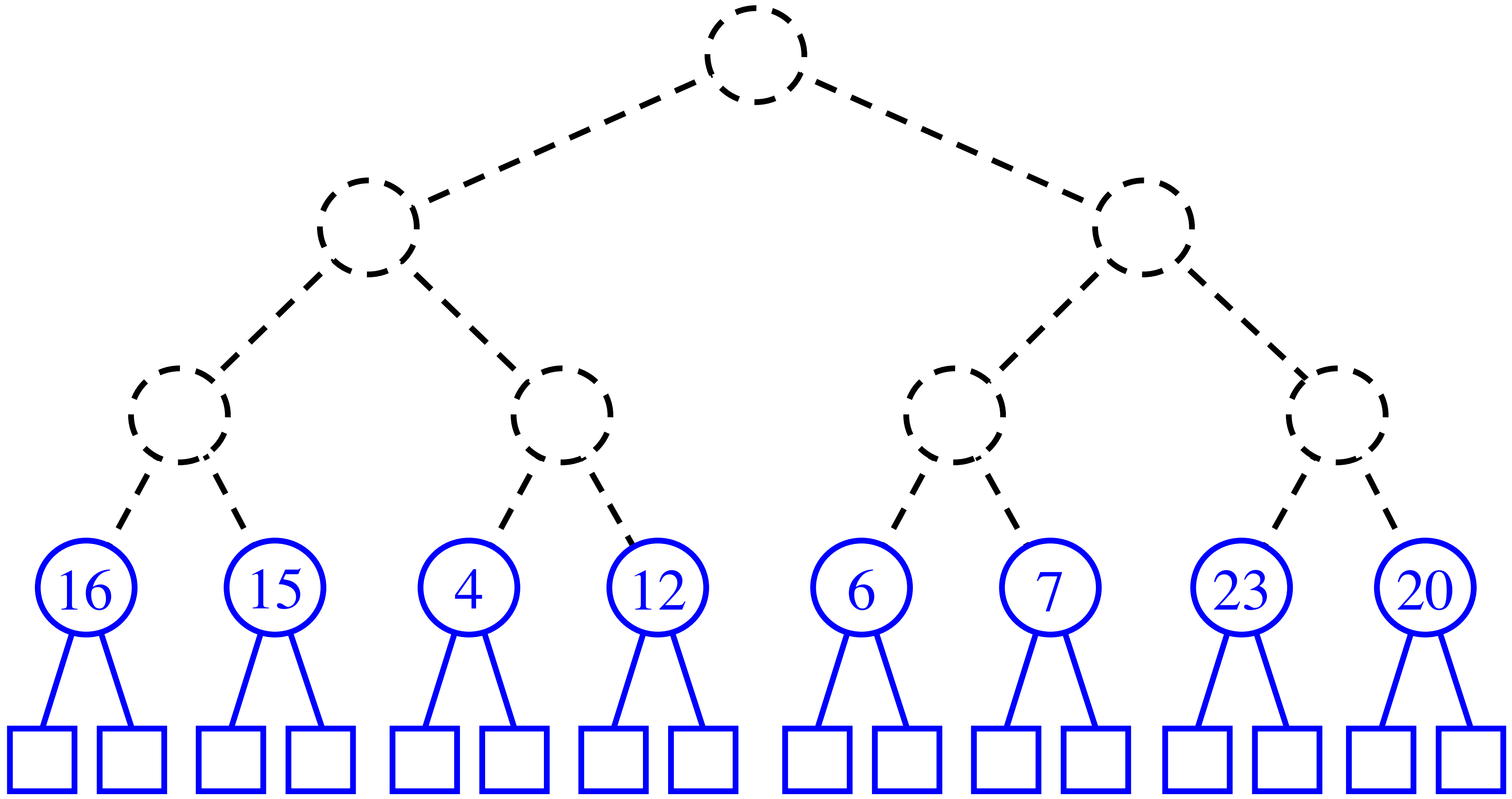
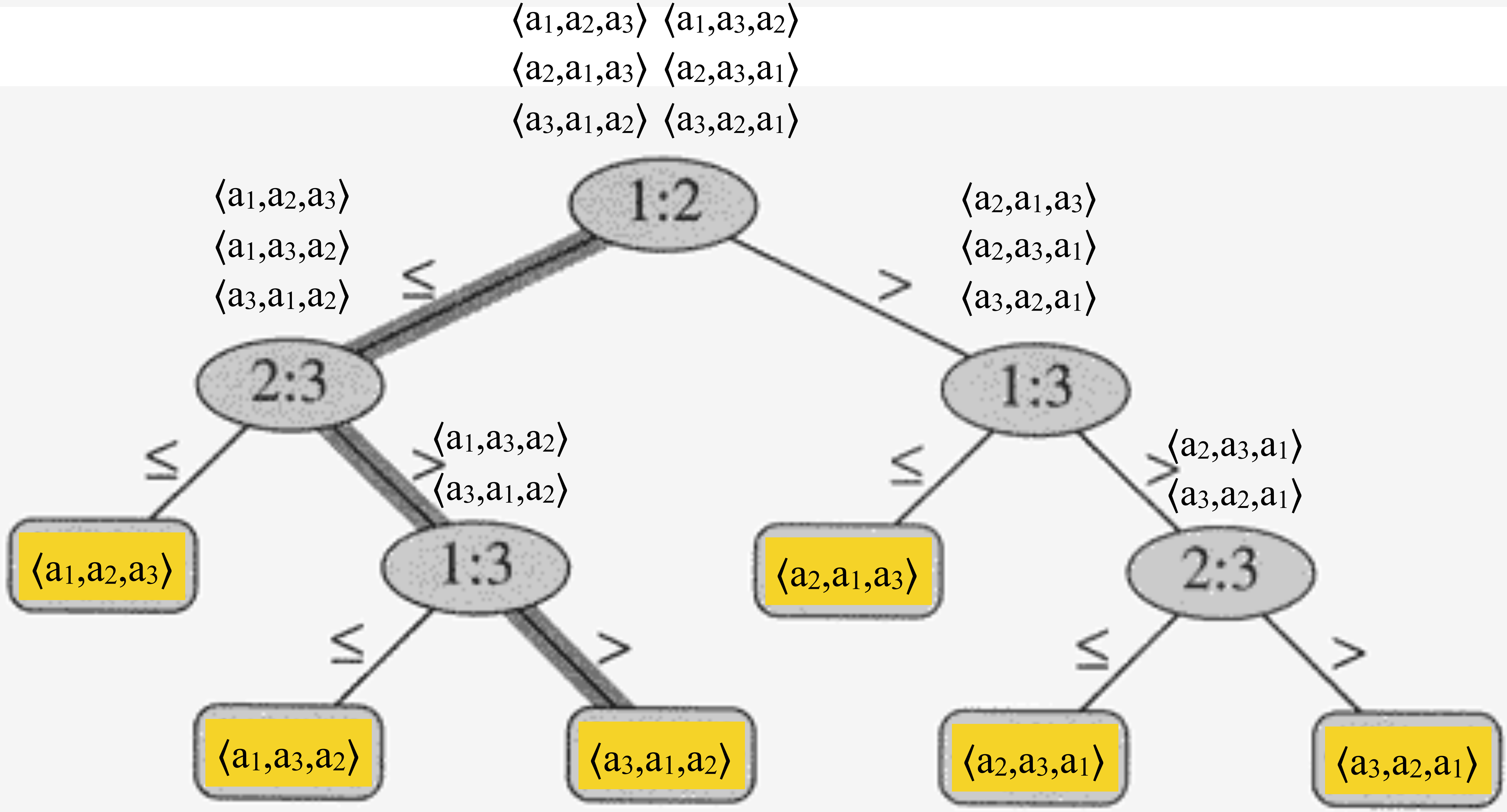
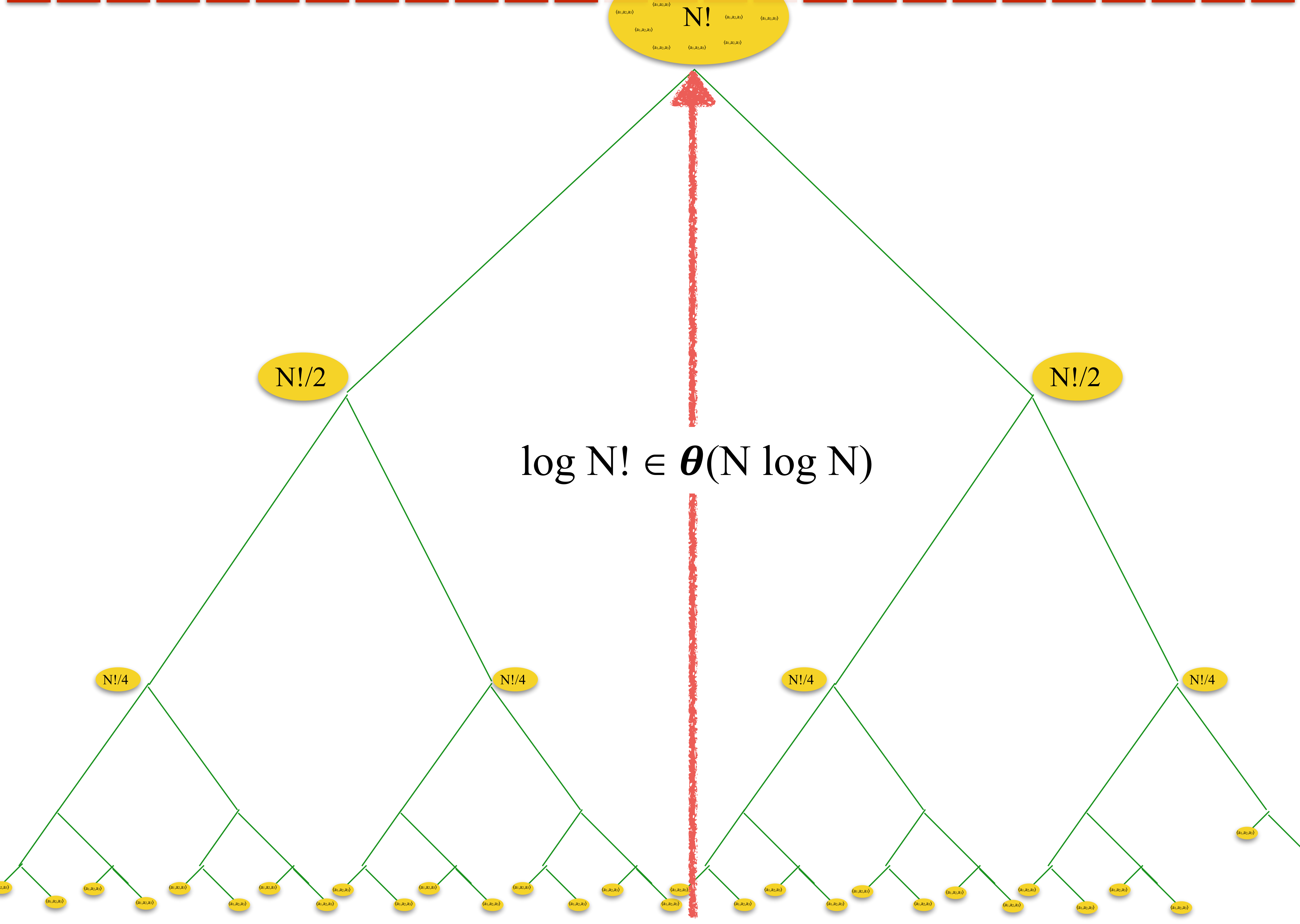
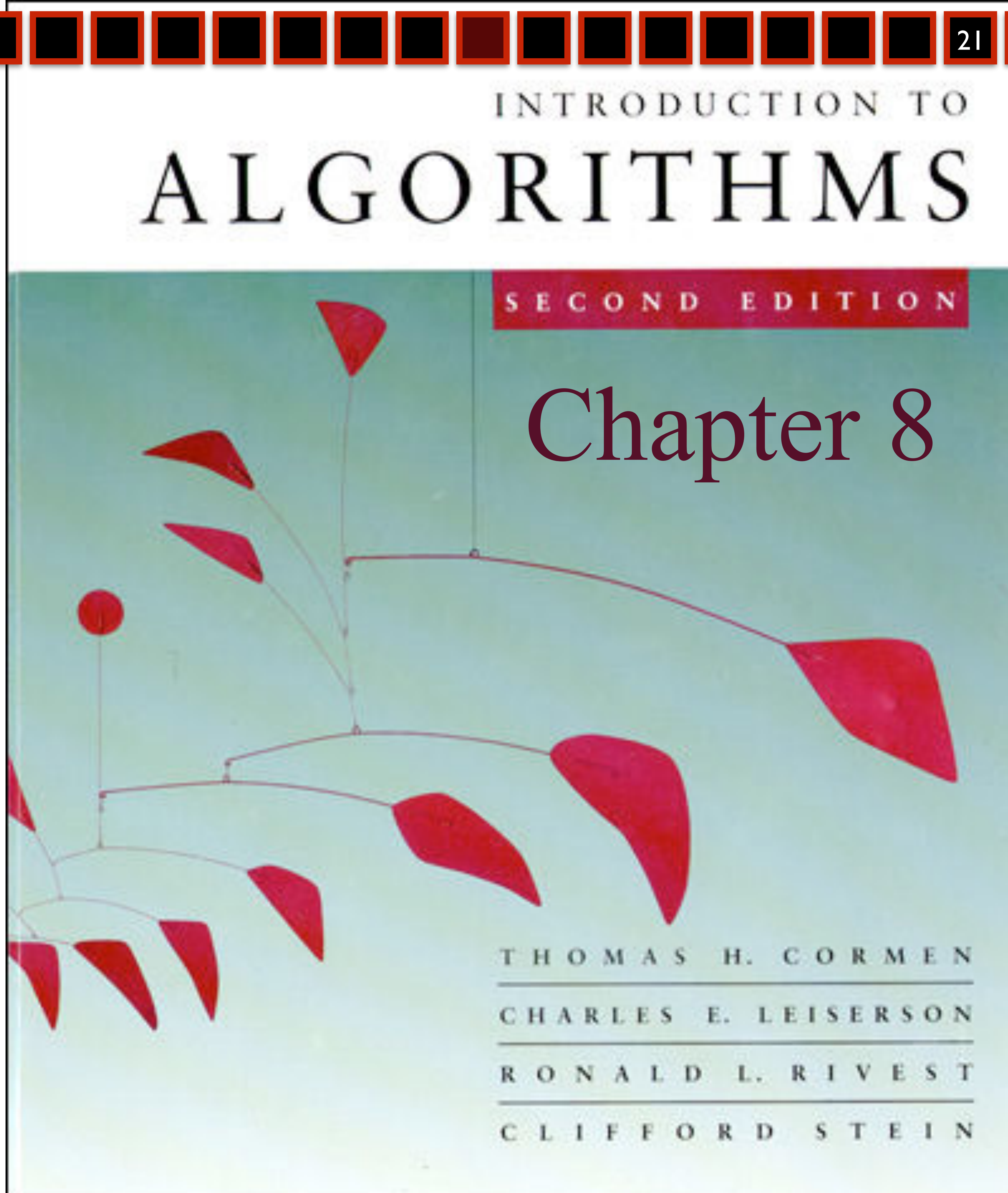


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.




N!
N!/2
N!/2
N!/4
N!/4
N!/4
N!/4
log N! ∈ θ(N log N)

Chapter 8


INTRODUCTION TO
ALGORITHMS
SECOND EDITION
THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

```
COUNTING-SORT(A, B, k)
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i ← 1 to k
        do C[i] ← C[i] + C[i − 1]
▷ C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
           C[A[j]] ← C[A[j]] − 1
```

Radix sort

How IBM made its money. Punch card readers for census tabulation in early 1900's. Card sorters, worked on one column at a time. It's the algorithm for using the machine that extends the technique to multi-column sorting. The human operator was part of the algorithm!

Key idea: Sort *least* significant digits first.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

```
RADIX-SORT(A, d)
    for i ← 1 to d
        do use a stable sort to sort array A on digit i
```

Correctness:

- Induction on number of passes (i in pseudocode).
- Assume digits $1, 2, \ldots, i-1$ are sorted.
- Show that a stable sort on digit i leaves digits $1, \ldots, i$ sorted:
  - If 2 digits in position i are different, ordering by position i is correct, and positions $1, \ldots, i-1$ are irrelevant.
  - If 2 digits in position i are equal, numbers are already in the right order (by inductive hypothesis). The stable sort on digit i leaves them in the right order.

This argument shows why it's so important to use a stable sort for intermediate sort.

```
RADIX-SORT(A, d)
for i ← 1 to d
        do use a stable sort to sort array A on digit i
```

Analysis: Assume that we use counting sort as the intermediate sort.

- $\Theta(n + k)$ per pass (digits in range $0, \ldots, k$)
- d passes
- $\Theta(d(n + k))$ total
- If $k = O(n)$, time $= \Theta(dn)$.

QuickSort

- Yet another sorting algorithm!
- Usually faster than other algorithms on average, although worst-case is $O(n^2)$
- Divide-and-conquer:
  - **Divide**: Choose an element of the array for *pivot*. Divide the elements into three groups: those smaller than the pivot, those equal, and those larger.
  - **Conquer**: Recursively sort each group.
  - **Combine**: Concatenate the three sorted groups.

QuickSort running time

- Worse case:
  - Already sorted array (either increasing or decreasing)
  - $T(n) = T(n-1) + c\ n + d$
  - $T(n)$ is $O(n^2)$
- Average case: If the array is in random order, the pivot splits the array in roughly equal parts, so the average running time is $O(n \log n)$
- Advantage over mergeSort:
  - constant hidden in $O(n \log n)$ are smaller for quickSort. Thus it is faster by a constant factor
  - QuickSort is easy to do "in-place"

In-place algorithms

- An algorithm is *in-place* if it uses only a *constant* amount of memory in addition of that used to store the input
- Importance of in-place sorting algorithms:
  - If the data set to sort barely fits into memory, we don't want an algorithm that uses twice that amount to sort the numbers
- SelectionSort and InsertionSort are in-place: all we are doing is moving elements around the array
- MergeSort is not in-place, because of the merge procedure, which requires a temporary array
- QuickSort can easily be made in-place...

Partition

Algorithm partition(A, start, stop)

Input: An array A, indices start and stop.

Output: Returns an index j and rearranges the elements of A such that for all i<j, A[i] ≤ A[j] and for all k>j, A[k] ≥ A[j].

pivot ← A[stop]

left ← start

right ← stop - 1

while left ≤ right **do**

  while left ≤ right **and** A[left] ≤ pivot) **do** left ← left + 1

  while (left ≤ right **and** A[right] ≥ pivot) **do** right ← right -1

  if (left < right **) then** exchange A[left] ↔ A[right]

exchange A[stop] ↔ A[left]

return left

In-place quickSort

Algorithm quickSort(A, start, stop)

Input: An array A to sort, indices start and stop

Output: A[start...stop] is sorted

if (start < stop) **then**

pivot ← partition(A, start, stop)

quickSort(A, start, pivot-1)

quickSort(A, pivot+1, stop)

Randomized Quicksort

```
RandomizedQuicksort(A,start,stop) {
   if |A| = 0 return

   choose a pivot A[i] uniformly at random (start ≤ i ≤ stop)
   exchange A[i] ↔ A[stop]

  pivot ← partition(A,start,stop)

   RandomizedQuicksort(A, start, pivot-1)
   RandomizedQuicksort(A, pivot+1, stop)
}
```

Quicksort

Running time.

- [Best case.] Select the median element as the pivot: quicksort makes Θ(n log n) comparisons.
- [Worst case.] Select the smallest (or largest) element as the pivot: quicksort makes $\Theta(n^2)$ comparisons.

Randomize. Protect against worst case by choosing pivot at random.

Intuition. If we always select a pivot that is bigger than 25% of the elements and smaller than 25% of the elements, then quicksort makes Θ(n log n) comparisons.

Notation. Label elements so that $x_1 < x_2 < \ldots < x_n$.

Randomized Quicksort: Expected Number of Comparisons

Theorem. Expected # of comparisons is O(n log n).

Theorem. [Knuth 1973] Stddev of number of comparisons is ~ 0.65n.

Ex. If n = 1 million, the probability that randomized quicksort takes less than 4n ln n comparisons is at least 99.94%.

Chebyshev's inequality. $\Pr[|X - \mu| \geq k\delta] < 1 / k^2$.

Mean Stddev

Strings and Pattern Matching

- Brute Force,Rabin-Karp, Knuth-Morris-Pratt
- Regular Expressions

String Searching

- The object of string searching is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).
- As with most algorithms, the main considerations for string searching are speed and efficiency.
- There are a number of string searching algorithms in existence today, but the three we shall review are Brute Force,Rabin-Karp, and Knuth-Morris-Pratt.

Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.
- Perhaps an example will clarify some things...

Rabin-Karp Algorithm

pattern is M characters long

hash_p=hash value of pattern
hash_t=hash value of first M letters in body of text

```
do
  if (hash_p == hash_t)
    brute force comparison of pattern
    and selected section of text
  hash_t = hash value of next section of
           text, one character over
until (end of text or
       brute force comparison == true)
```

Rabin-Karp Complexity

- If a sufficiently large prime number is used for the *hash function*, the hashed values of two different patterns will usually be distinct.
- If this is the case, searching takes O(N) time, where N is the number of characters in the larger body of text.
- It is always possible to construct a scenario with a worst case complexity of O(MN). This, however, is likely to happen only if the prime number used for hashing is small.

Comment about input size…

2)
Write *any* algorithm that runs in time $\Theta(n^2 \log^2 n)$ in worse case.
Explain why this is its running time. I don't care what it does.
I only care about its running time…

```
WhatEver(int m)

FOR i=1 TO m
  FOR j=1 TO m
     x=m; WHILE x>1 DO { x=x/2; y=m;
                         WHILE y>1 DO y=y/2 }
```

n = |m| ~ log m. Therefore running time is $\Theta(m^2 \log^2 m) = \Theta(2^{2n} n^2)$

Comment about input size…

2)
Write *any* algorithm that runs in time $\Theta(n^2 \log^2 n)$ in worse case.
Explain why this is its running time. I don't care what it does.
I only care about its running time…

```
WhatEver(int[] A)

n = A.length;
FOR i=1 TO n
  FOR j=1 TO n
      x=n; WHILE x>1 DO { x=x/2; y=n;
                          WHILE y>1 DO y=y/2 }
```

Strings and Pattern Matching

- Brute Force,Rabin-Karp, Knuth-Morris-Pratt
- Regular Expressions

The Knuth-Morris-Pratt Algorithm

- The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.
- A failure function (*f*) is computed that indicates how much of the last comparison can be reused if it fails.
- Specifically, *f* is defined to be the longest prefix of the pattern P[0,..,j] that is also a suffix of P[1,..,j]
  - Note: **not** a suffix of P[0,..,j]

The KMP Algorithm (contd.)

- the KMP string matching algorithm: Pseudo-Code

Algorithm KMPMatch(T,P)
  Input: Strings T (text) with n characters and P (pattern) with m characters.
  Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

```
f ← KMPFailureFunction(P) {build failure function}
i ← 0
j ← 0
while i < n do
   if P[j] = T[i] then
      if j = m - 1 then
         return i - m - 1 {a match}
      i ← i + 1
      j ← j + 1
   else if j > 0 then {no match, but we have advanced}
      j ← f(j-1) {j indexes just after matching prefix in P}
   else
      i ← i + 1
return "There is no substring of T matching P"
```

The KMP Algorithm (contd.)

- The KMP failure function: Pseudo-Code

Algorithm KMPFailureFunction(P);
  Input: String P (pattern) with m characters
  Ouput: The faliure function f for P, which maps j to the length of the longest prefix of P that is a suffix of $P[1,..,j]$

```
i ← 1
j ← 0
while i ≤ m-1 do
   if P[j] = P[i] then
      {we have matched j + 1 characters}
      f(i) ← j + 1
      i ← i + 1
      j ← j + 1
   else if j > 0 then
      {j indexes just after a prefix of P that matches}
      j ← f(j-1)
   else
      {there is no match}
      f(i) ← 0
      i ← i + 1
```

The KMP Algorithm (contd.)

- Time Complexity Analysis
- define $k = i - j$
- In every iteration through the while loop, one of three things happens.
  - 1) if $T[i] = P[j]$, then i increases by 1, as does j
    k remains the same.
  - 2) if $T[i]$!= $P[j]$ and $j > 0$, then i does not change
    and k increases by at least 1, since k changes from $i - j$ to $i - f(j\text{-}1)$
  - 3) if $T[i]$!= $P[j]$ and $j = 0$, then i increases by 1 and
    k increases by 1 since j remains the same.

The KMP Algorithm (contd.)

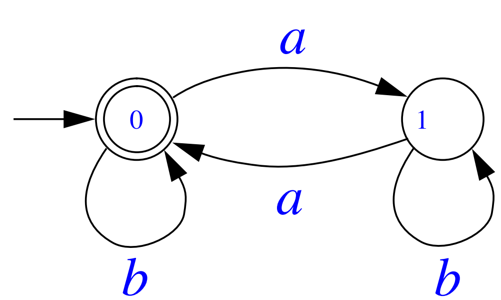
- Thus, each time through the loop, either i or k increases by at least 1, so the greatest possible number of loops is $2n$
- This of course assumes that f has already been computed.
- However, f is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is $\boldsymbol{O}(m)$
- Total Time Complexity: $\boldsymbol{O}(n + m)$

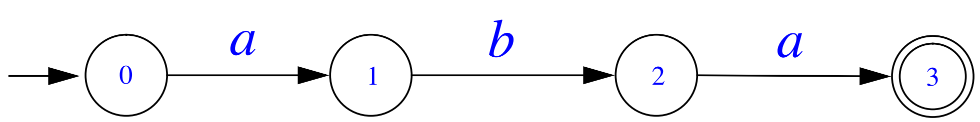
Regular Expressions

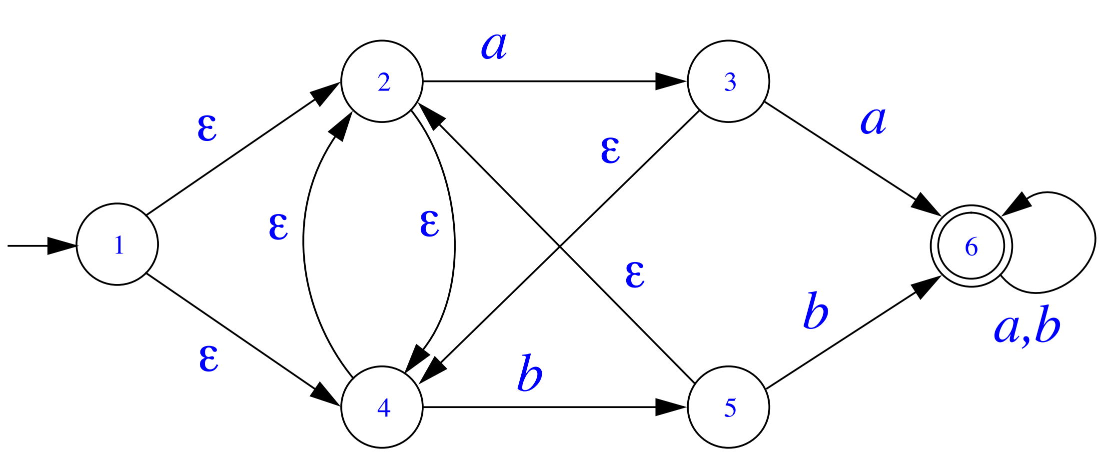
- notation for describing a set of strings, possibly of infinite size
- ε denotes the empty string
- **ab + c** denotes the set {ab, c}
- **a*** denotes the set {ε, a, aa, aaa, ...}
- Examples
  - **(a+b)*** all the strings from the alphabet {a,b}
  - **b*(ab*a)*b*** strings with an even number of a's
  - **(a+b)*sun(a+b)*** strings containing the pattern "sun"
  - **(a+b)(a+b)(a+b)a** 4-letter strings ending in a

Finite State Automaton

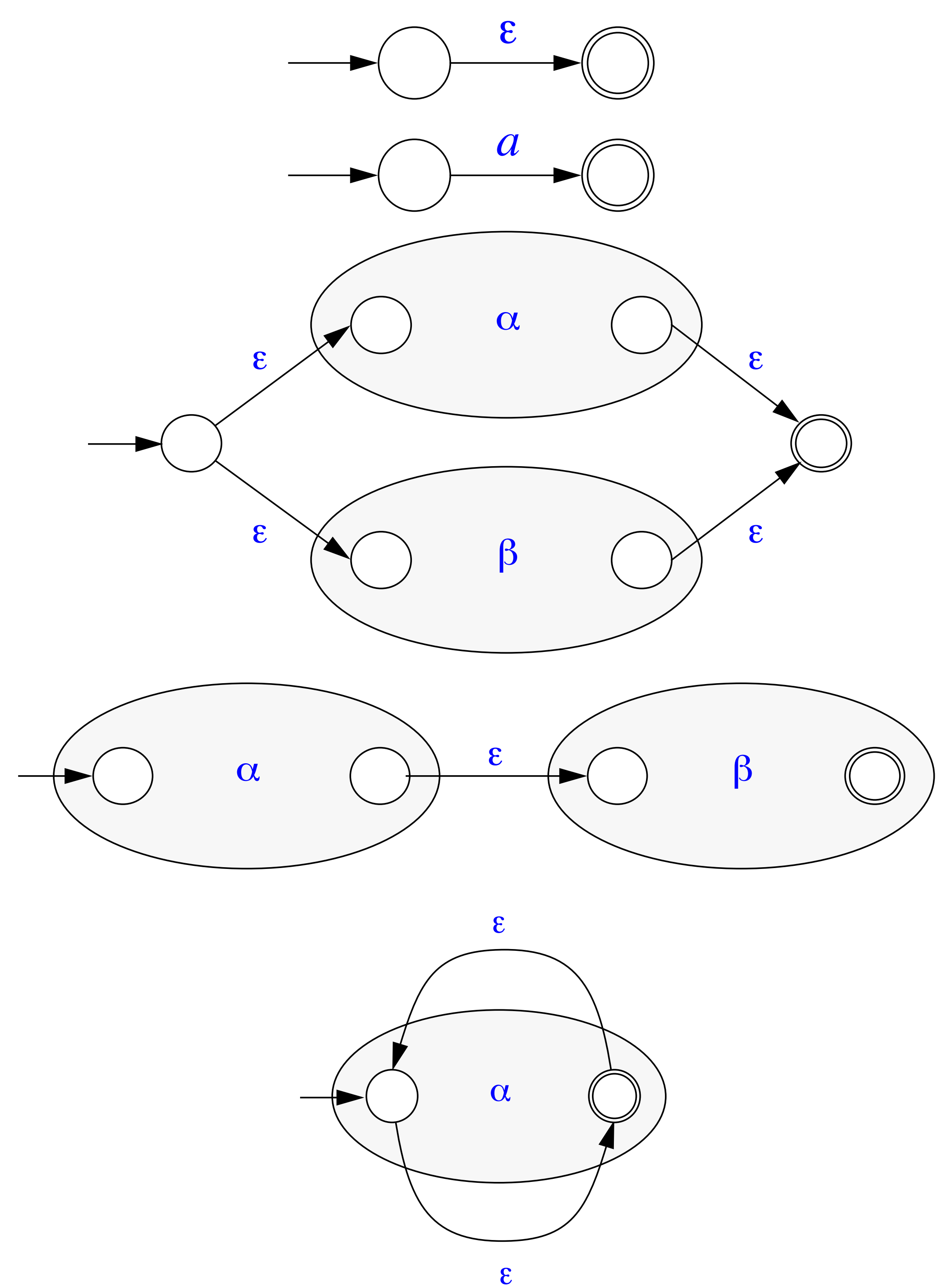
- “machine” for processing strings



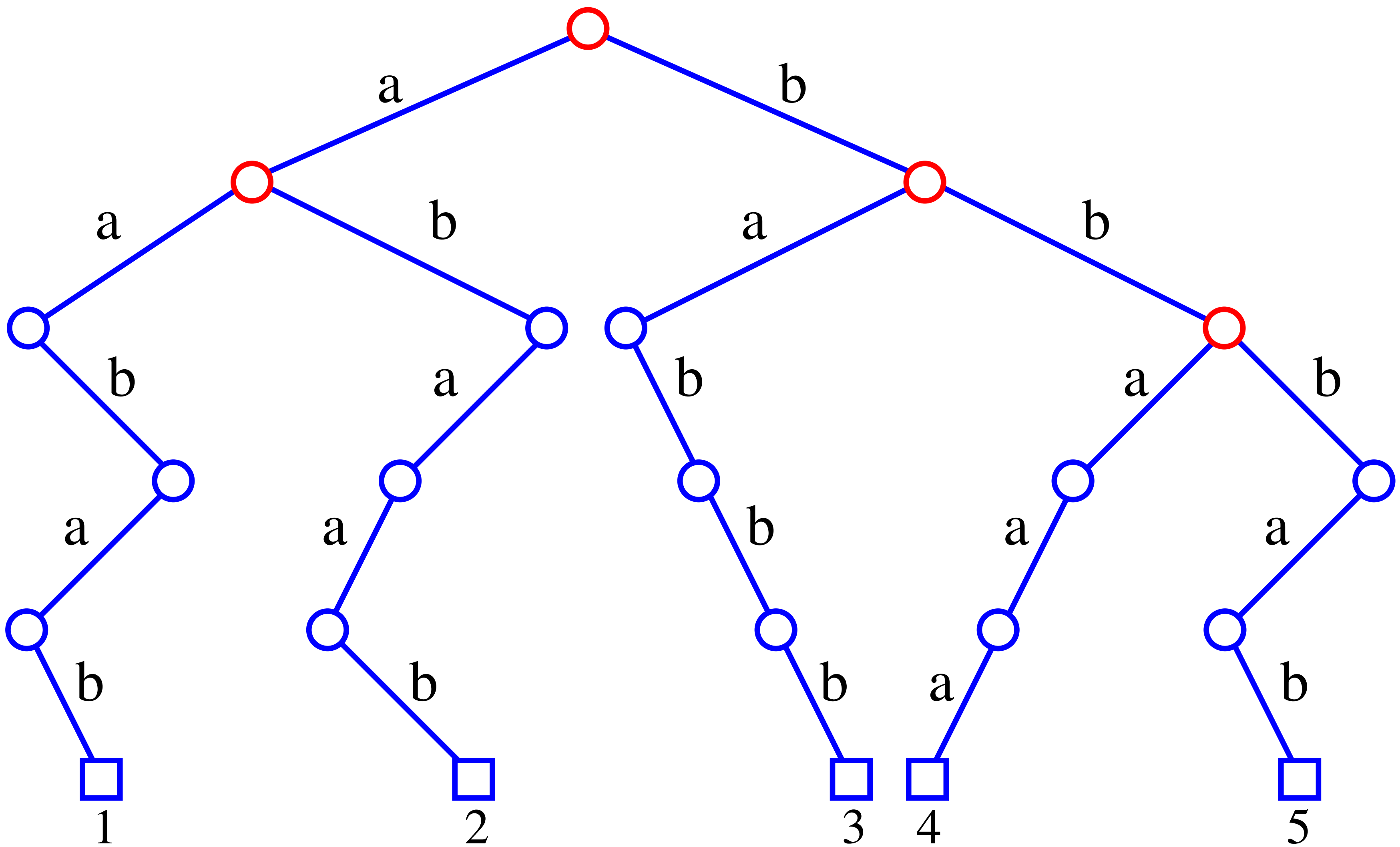




Composition of FSA’s



Tries



Tries

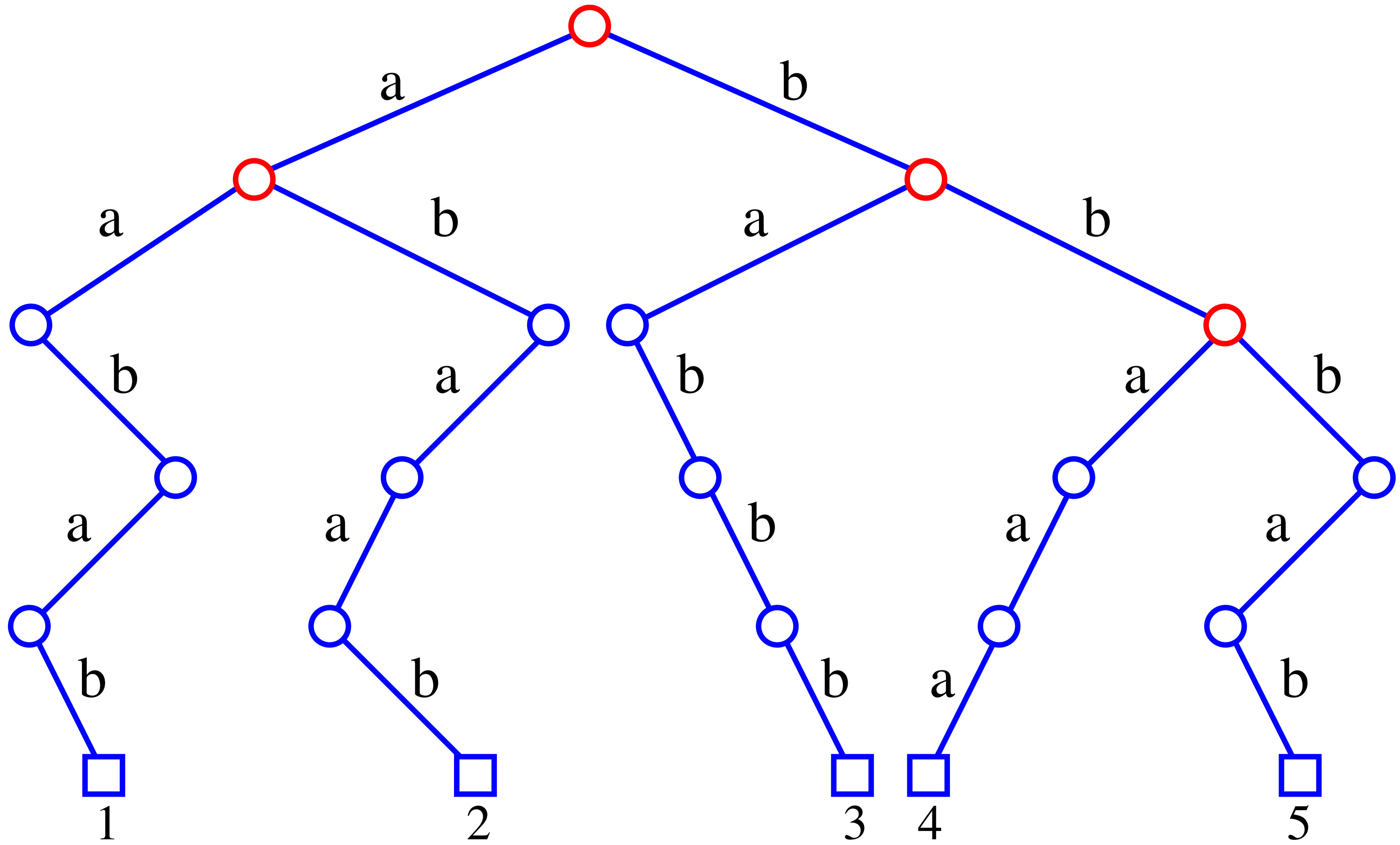
- A trie is a tree-based data structure for storing strings in order to make pattern matching faster.
- Tries can be used to perform **prefix queries** for information retrieval. Prefix queries search for the longest prefix of a given string X that matches a prefix of some string in the trie.
- A trie supports the following operations on a set S of strings:

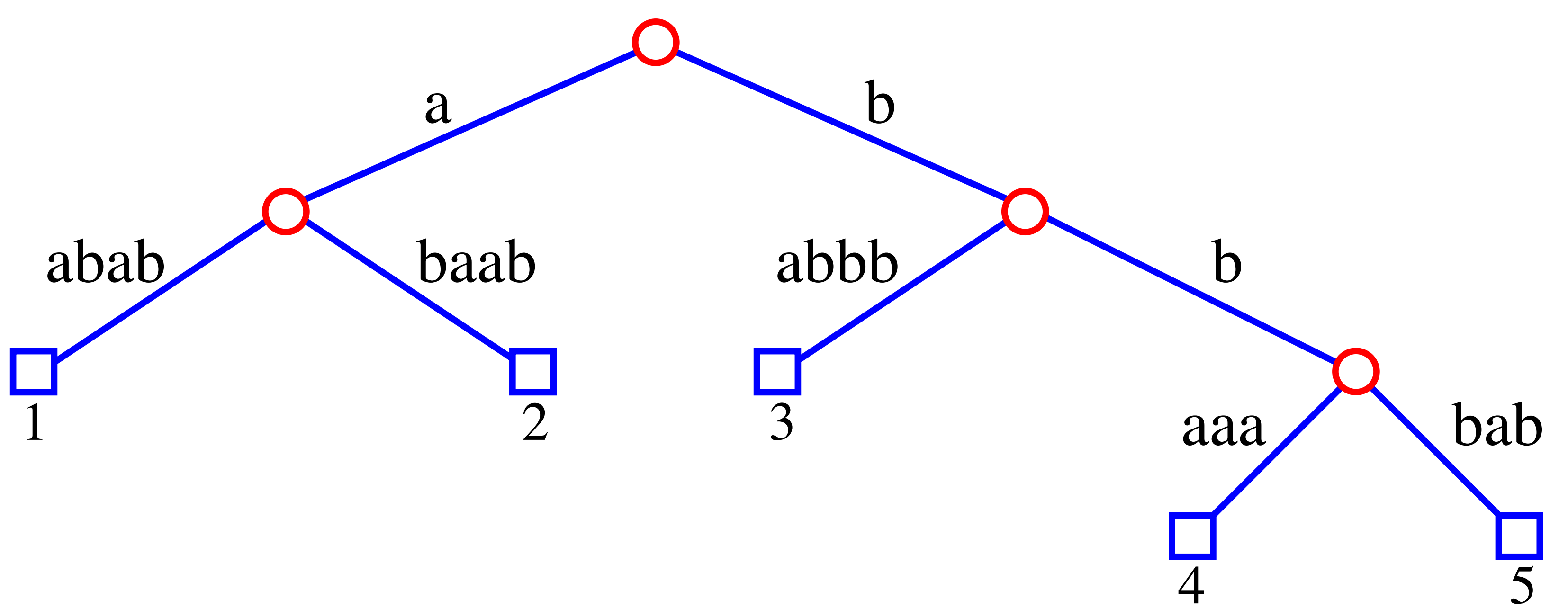
  insert(X): Insert the string X into S
  Input: String **Ouput**: None

  remove(X): Remove string X from S
  Input: String **Output**: None

  prefixes(X): Return all the strings in S that have a longest prefix of X
  Input: String **Output**: Enumeration of strings

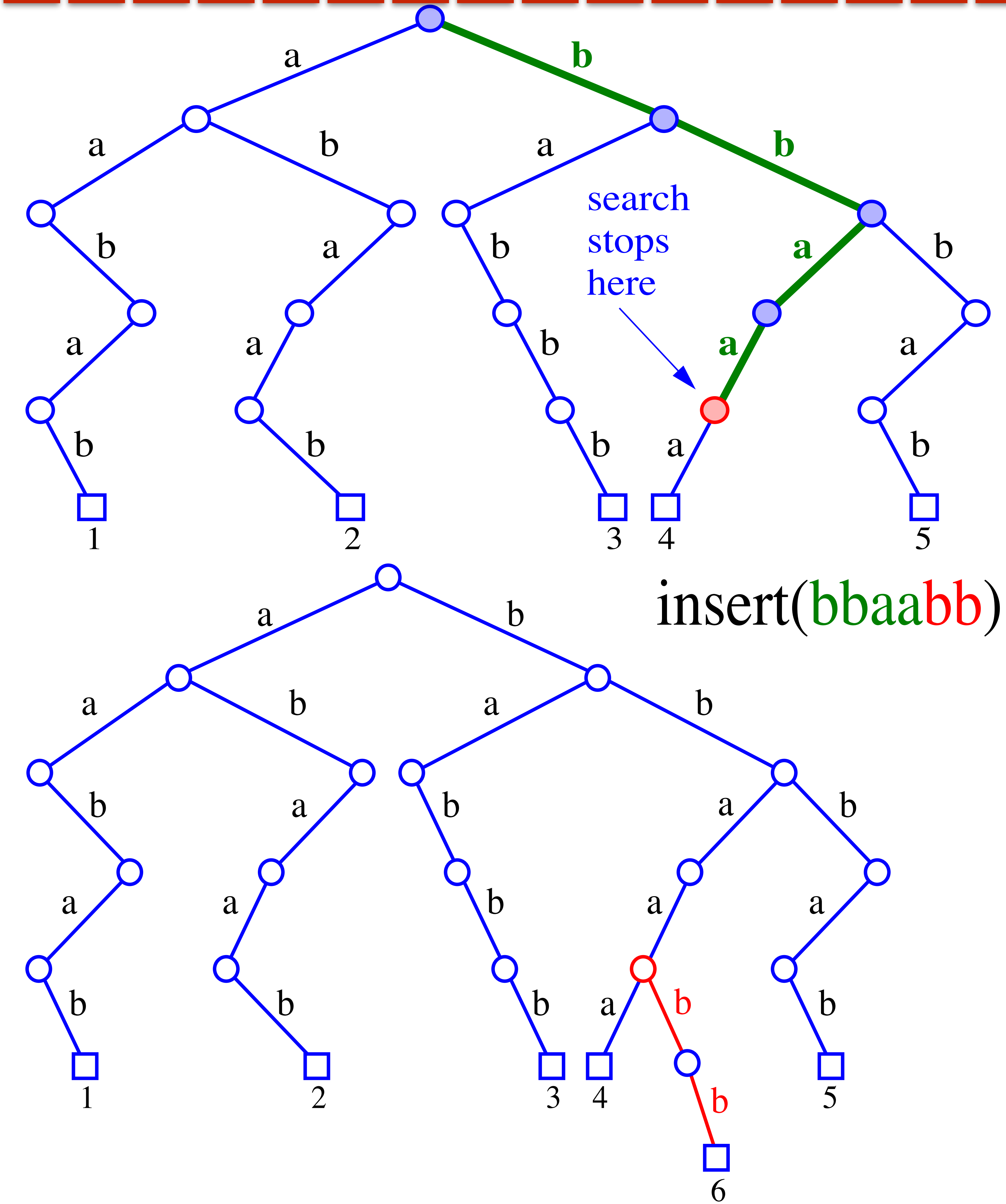
- An example:


a
b
a
b
a
b
b
a
b
a
b
a
a
a
b
b
a
b
b
b
a
b
1
2
3
4
5

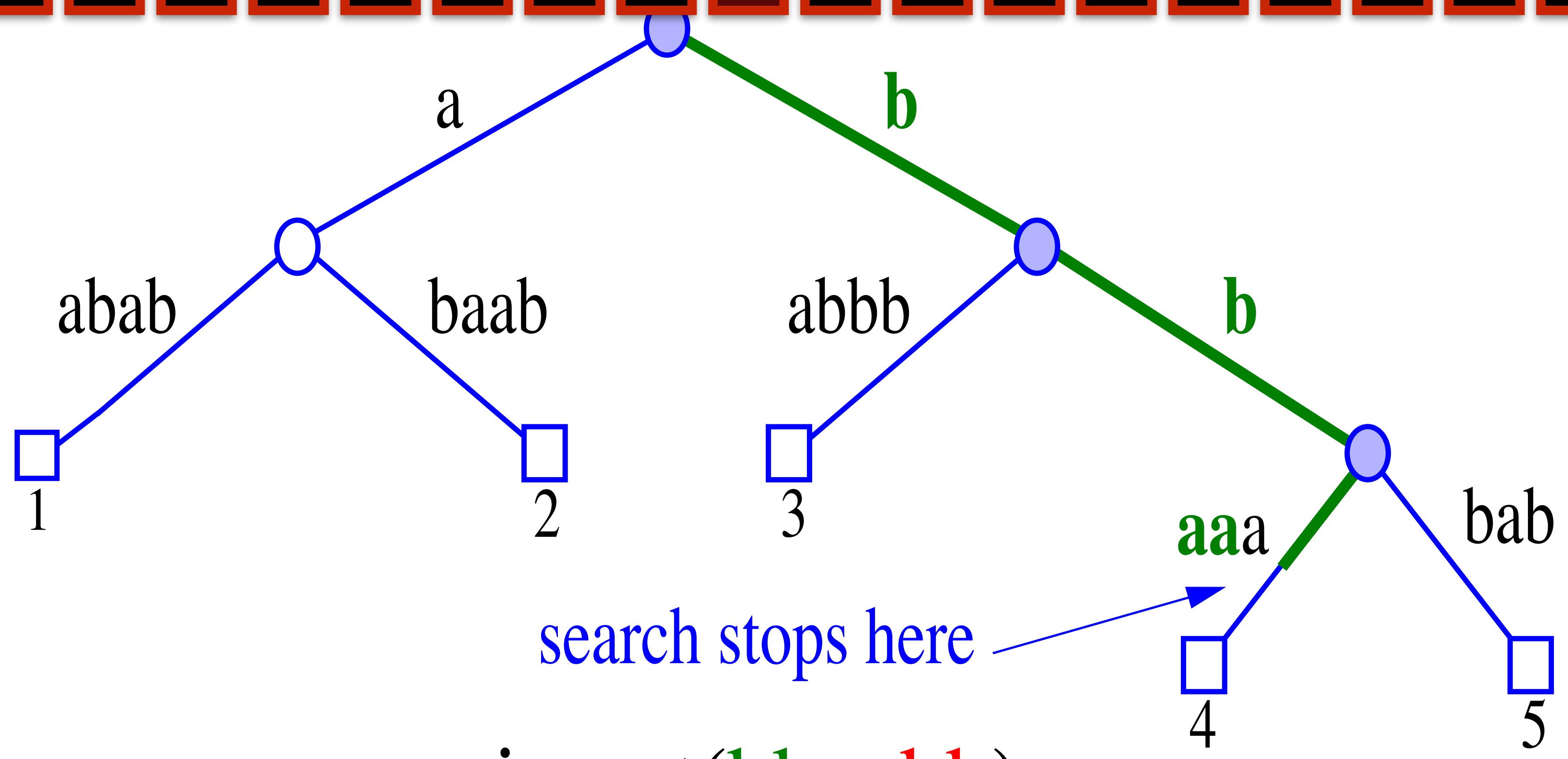

a
b
abab
baab
abbb
b
aaa
bab
1
2
3
4
5

Prefix Queries on a Trie

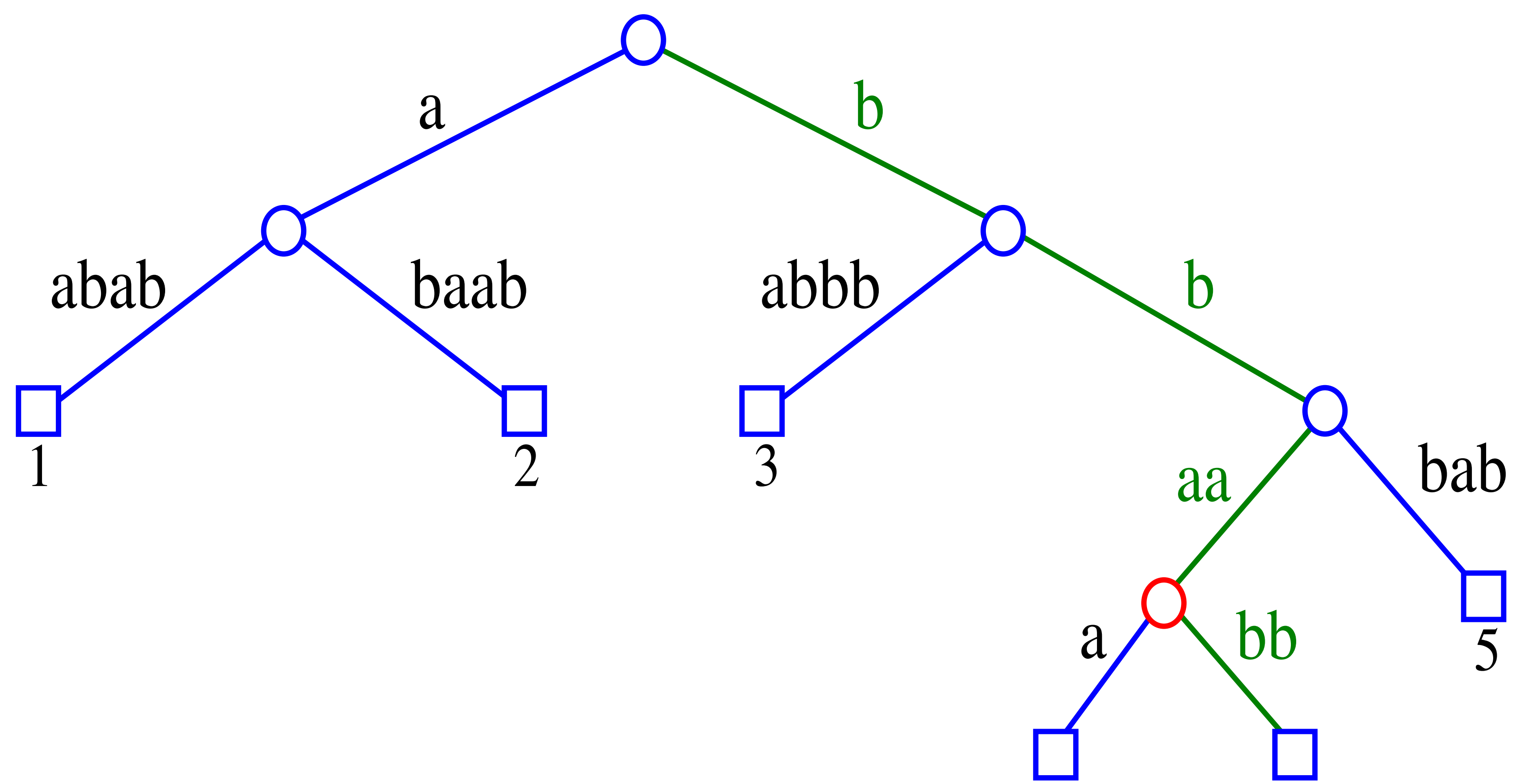
Algorithm prefixQuery(T, X):
  Input: Trie T for a set S of strings and a query string X
  Output: The node v of T such that the labeled nodes of the subtree of T rooted at v store the strings of S with a longest prefix in common with X
  $v \leftarrow T$.root()
  $i \leftarrow 0$ {i is an index into the string X}
  repeat
    for each child w of v **do**
    let e be the edge (v,w)
    $Y \leftarrow$string(e) {Y is the substring associated with e}
    $l \leftarrow Y$.length() {l=1 if T is a standard trie}
  Z$\leftarrow X$.substring(i, i+l-1) {Z holds the next l characters of X}
    if Z = Y **then**
      $v \leftarrow$w
      $i \leftarrow i$+1{move to W, incrementing i past Z}
      break out of the **for** loop
    else if a proper prefix of Z matched a proper prefix of Y **then**
      $v \leftarrow$w
      break out ot the **repeat** loop
until v is external **or** $v \neq$w
return v



a
b
search
stops
here
1
2
3
4
5
insert(bbaabb)
6



insert(bbaabb)



Lempel Ziv Encoding

- Constructing the trie:
  - Let phrase 0 be the null string.
  - Scan through the text
  - If you come across a letter you haven't seen before, add it to the top level of the trie.
  - If you come across a letter you've already seen, scan down the trie until you can't match any more characters, add a node to the trie representing the new string.
  - Insert the pair (nodeIndex, lastChar) into the compressed string.
- Reconstructing the string:
  - Every time you see a '0' in the compressed string add the next character in the compressed string directly to the new string.
  - For each non-zero nodeIndex, put the substring corresponding to that node into the new string, followed by the next character in the compressed string.

Lempel Ziv Encoding (contd.)

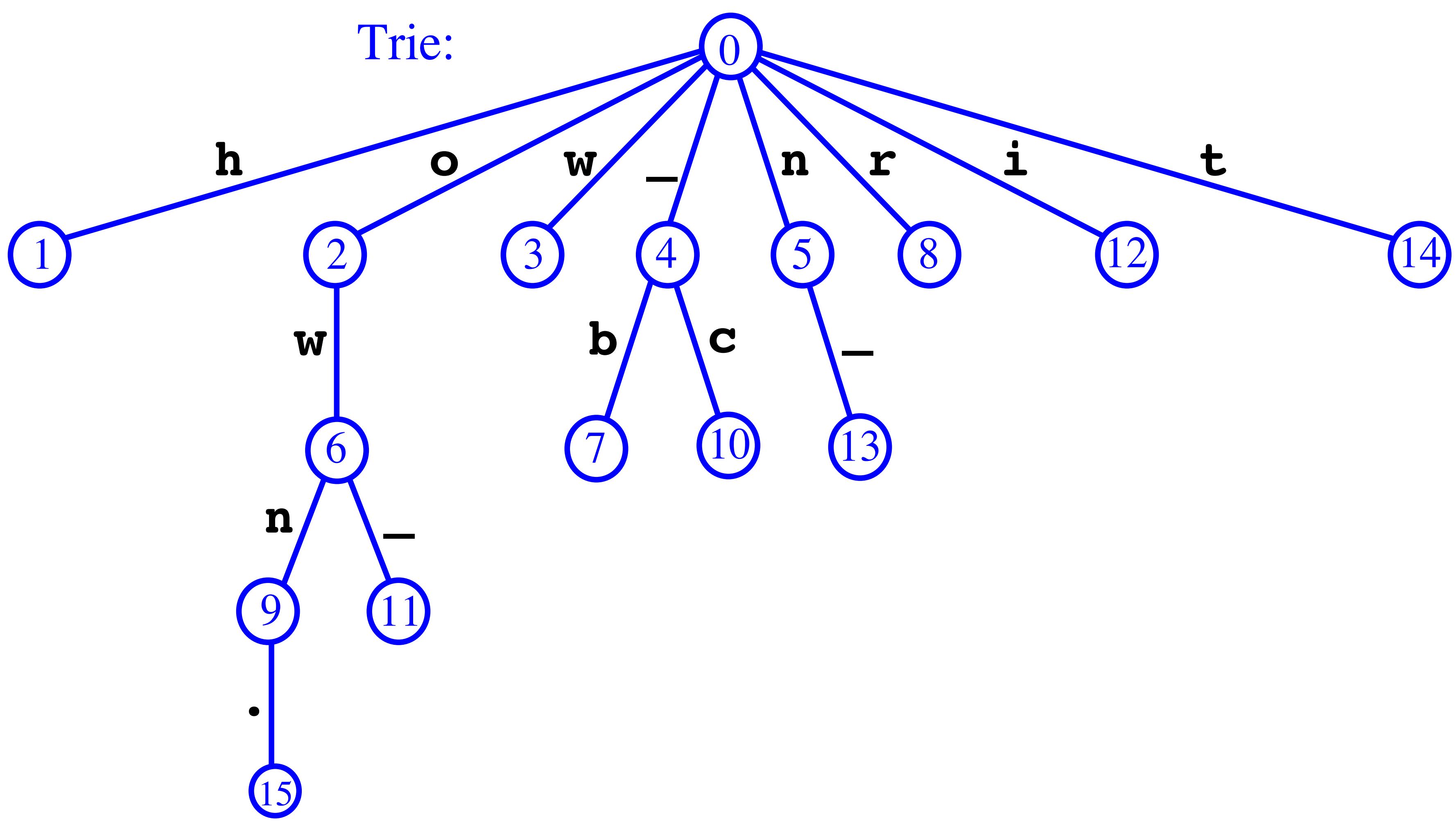
- A graphical example:

Uncompressed text: **how_now_brown_cow_in_town.**

phrases: (nil) 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Compressed text: 0**h**0**o**0**w**0_0**n**2**w**4**b**0**r**6**n**4**c**6_0**i**5_0**t**9**.**

Trie:

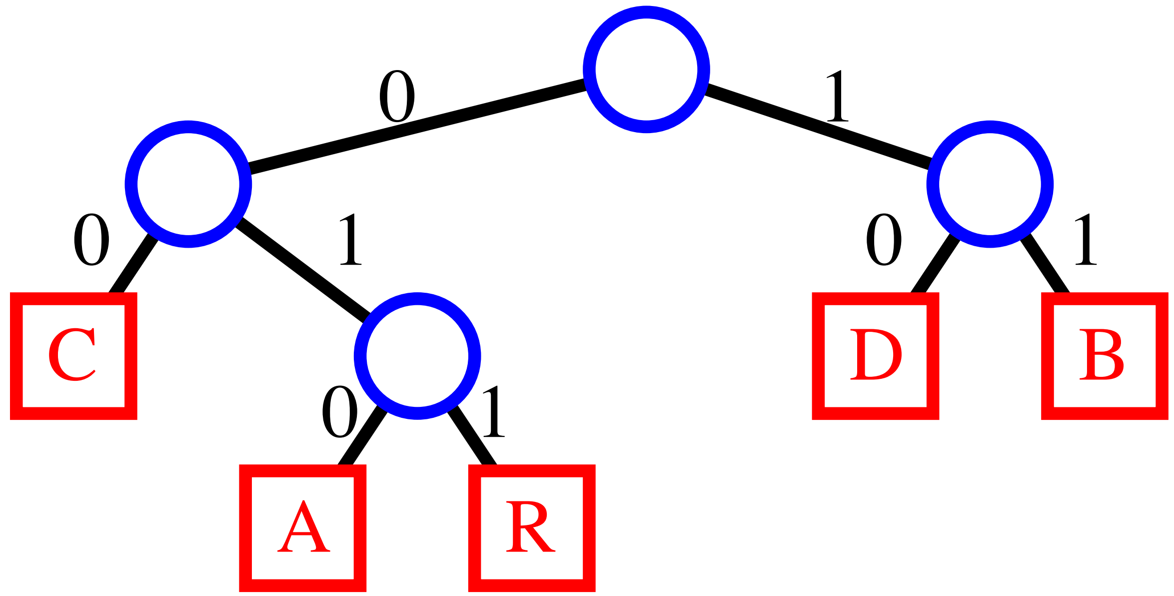


File Compression

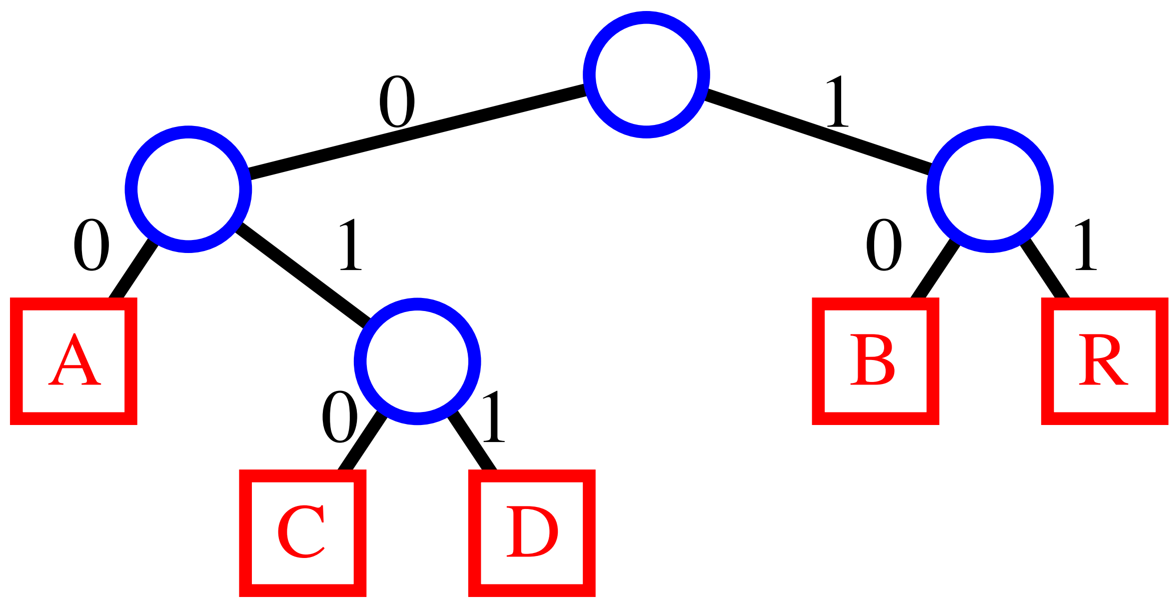
- text files are usually stored by representing each character with an 8-bit ASCII code (type man ascii in a Unix shell to see the ASCII encoding)
- the ASCII encoding is an example of **fixed-length encoding**, where each character is represented with the same number of bits
- in order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others
- **variable-length encoding** uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters.
- Example:
  - text: java
  - encoding: a = "0", j = "11", v = "10"
  - encoded text: 110100 (6 bits)
- How to decode?
  - a = "0", j = "01", v = "00"
  - encoded text: 010000 (6 bits)
  - is this java, jvv, jaaaa ...

Optimal Compression

- An issue with encoding tries is to insure that the encoded text is as short as possible:



ABRACADABRA
01011011010000101001011011010
29 bits



ABRACADABRA
001011000100001100101100
24 bits

Construction Algorithm

- with a Huffman encoding trie, the encoded text has minimal length

Algorithm Huffman(X):
Input: String X of length n
Output: Encoding trie for X

Compute the frequency $f(c)$ of each character c of X.
Initialize a priority queue Q.

for each character c in X **do**
  Create a single-node tree T storing c
  Q.insertItem($f(c)$, T)
while Q.size() > 1 **do**
  $f_1 \leftarrow Q$.minKey()
  $T_1 \leftarrow Q$.removeMinElement()
  $f_2 \leftarrow Q$.minKey()
  $T_2 \leftarrow Q$.removeMinElement()
  Create a new tree T with left subtree T_1 and right subtree T_2.
  Q.insertItem($f_1 + f_2$)
return tree Q.removeMinElement()

- runing time for a text of length n with k distinct characters: O(n + k log k)

Winter 2016
COMP-250: Introduction to Computer Science

Lecture 26, April 14, 2016

REVIEW SESSION