Lecture#7

Data Structures

Dr. Abu Nowshed Chy

Department of Computer Science and Engineering

University of Chittagong

February 10, 2025

[Faculty Profile]

Stack

- A stack is a data structure in which items can be inserted only from one end and deleted from the same end.
- Stacks are a special form of collection with LIFO semantics
- Two methods
  - PUSH → add item to the top of the stack
  - POP → remove an item from the top of the stack
- It could be thought of just like a stack of plates placed on table, a person always takes off a plate from the top and the new plates are placed on to the stack at the top.

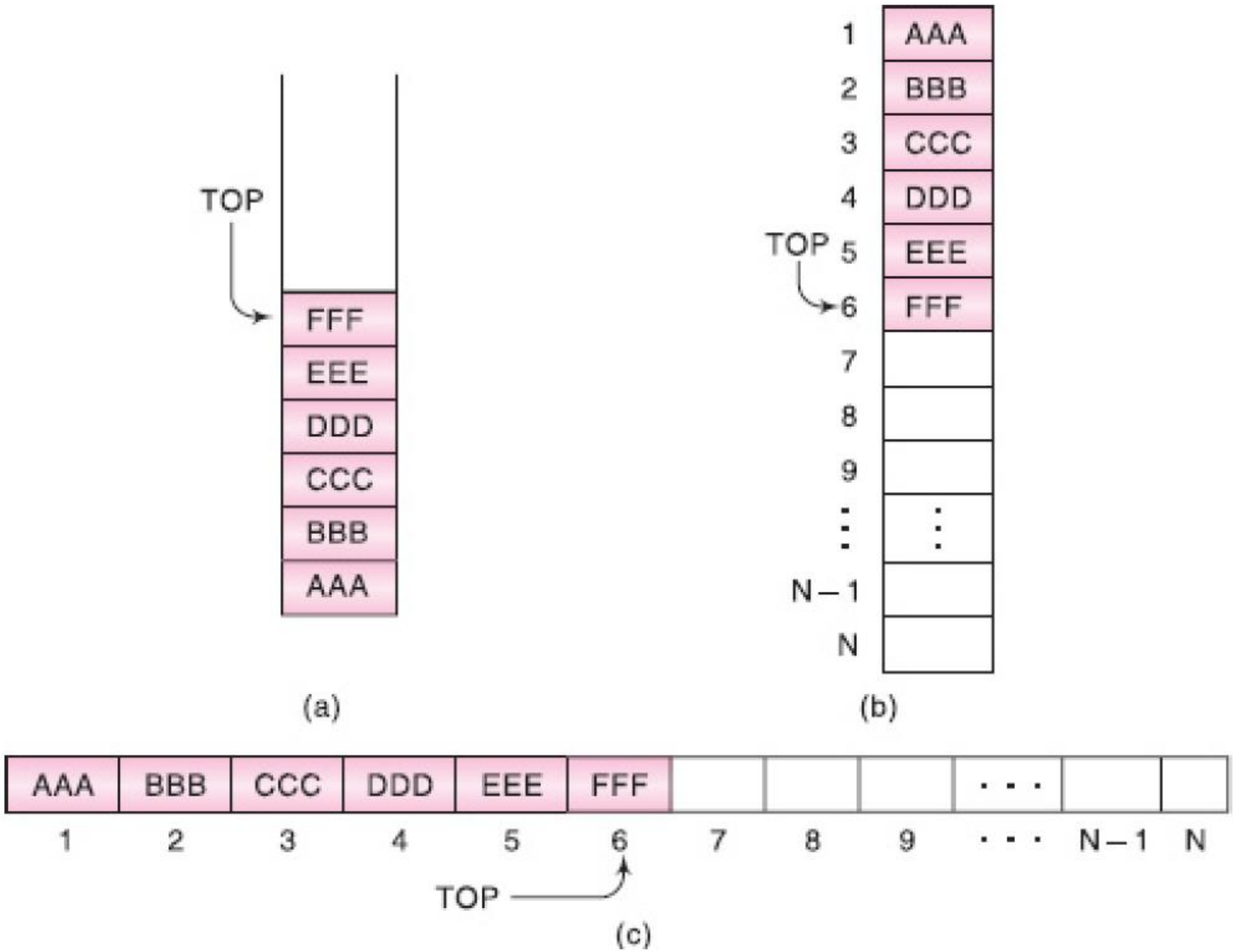
Examples

Implementing a Stack

Implementing a stack using an array is fairly easy.

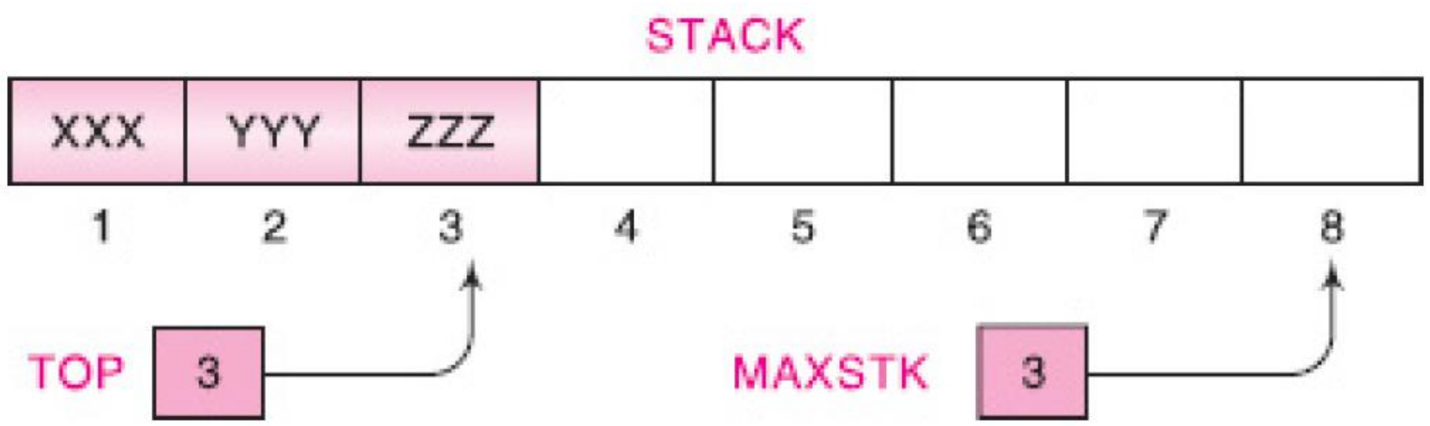
- The bottom of the stack is at data[0]
- The top of the stack is at data[numItems-1]
- *push* onto the stack at data[numItems]
- *pop* off of the stack at data[numItems-1]

Array Representation of a Stack



Array Representation of a Stack

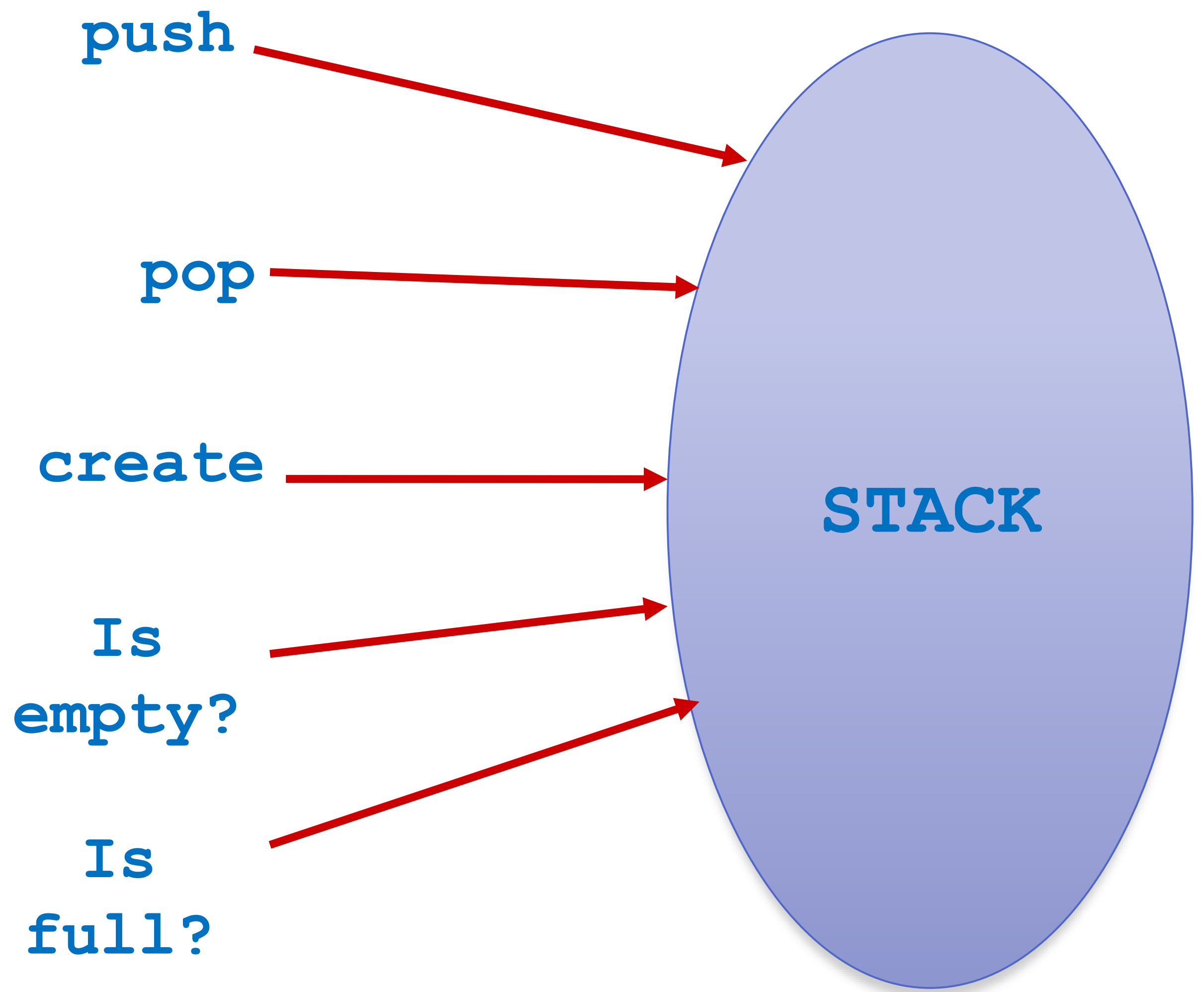
Since TOP = 3, the stack has three elements, XXX, YYY, and ZZZ; and since MAXSTK = 8, there is room for 5 more items in the stack.



Stack Operation

- Create an empty stack
- Destroy a stack
- Determine whether a stack is empty
- Add a new item
- Remove the item that was added most recently
- Retrieve the item that was added most recently

Stack Operation



PUSH Operation

Procedure 6.1: PUSH(STACK, TOP, MAXSTK, ITEM)

This procedure pushes an ITEM onto a stack.

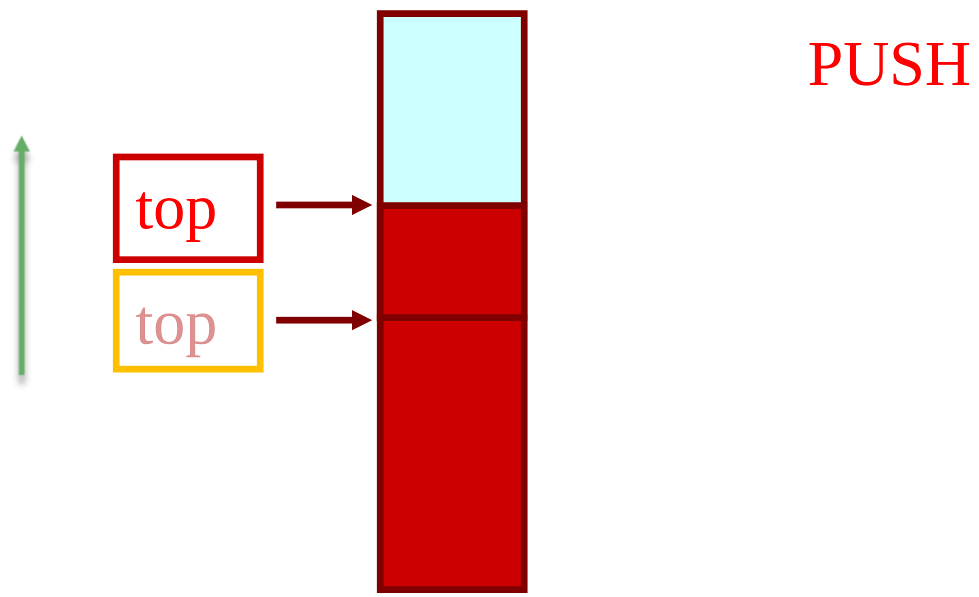
1. [Stack already filled?]
If TOP = MAXSTK, then: Print: OVERFLOW, and Return.

2. Set TOP := TOP + 1. [Increases TOP by 1.]

3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]

4. Return.

Stack Operation



POP Operation

Procedure 6.2: POP(STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

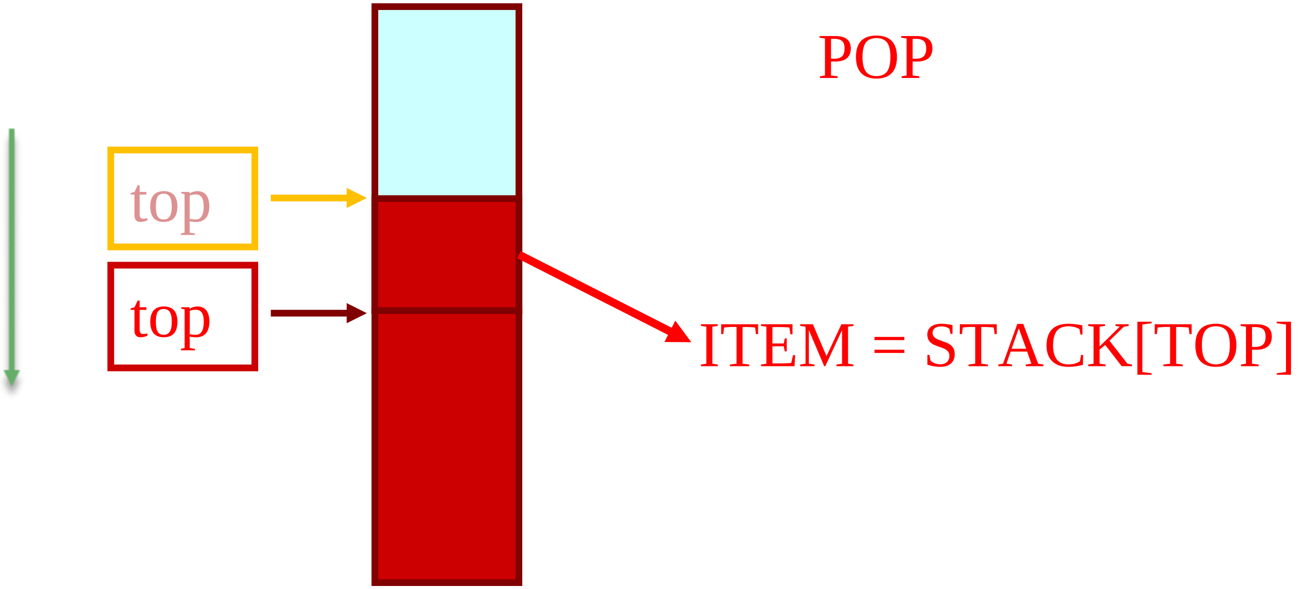
1. [Stack has an item to be removed?]
If TOP = 0, then: Print: UNDERFLOW, and Return.

2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]

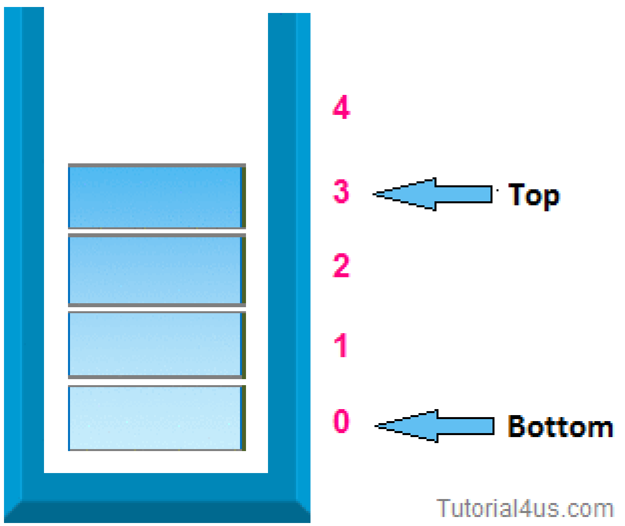
3. Set TOP := TOP – 1. [Decreases TOP by 1.]

4. Return.

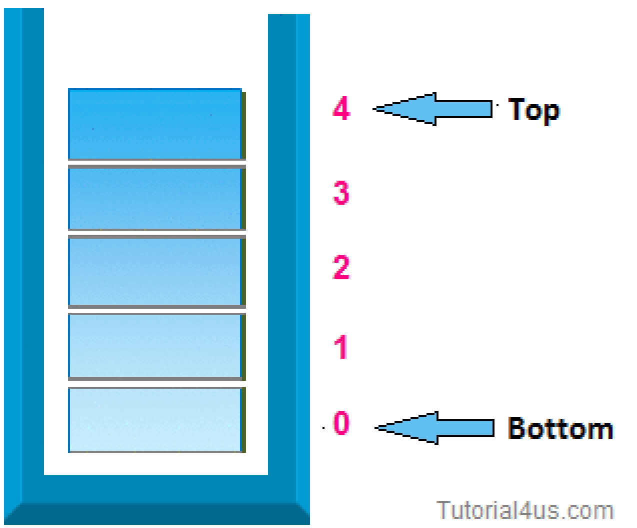
Stack Operation



Stack Operation

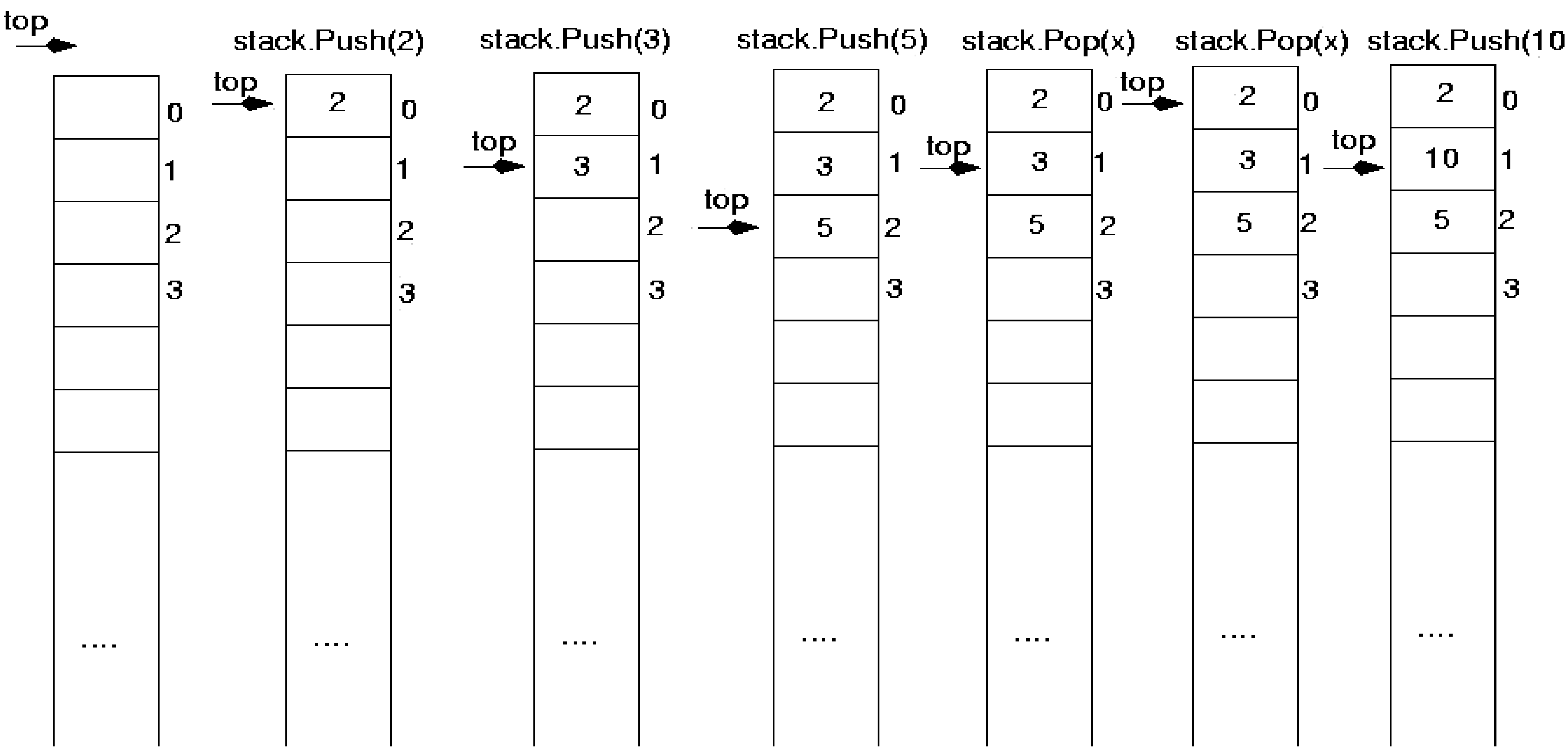


PUSH Operation

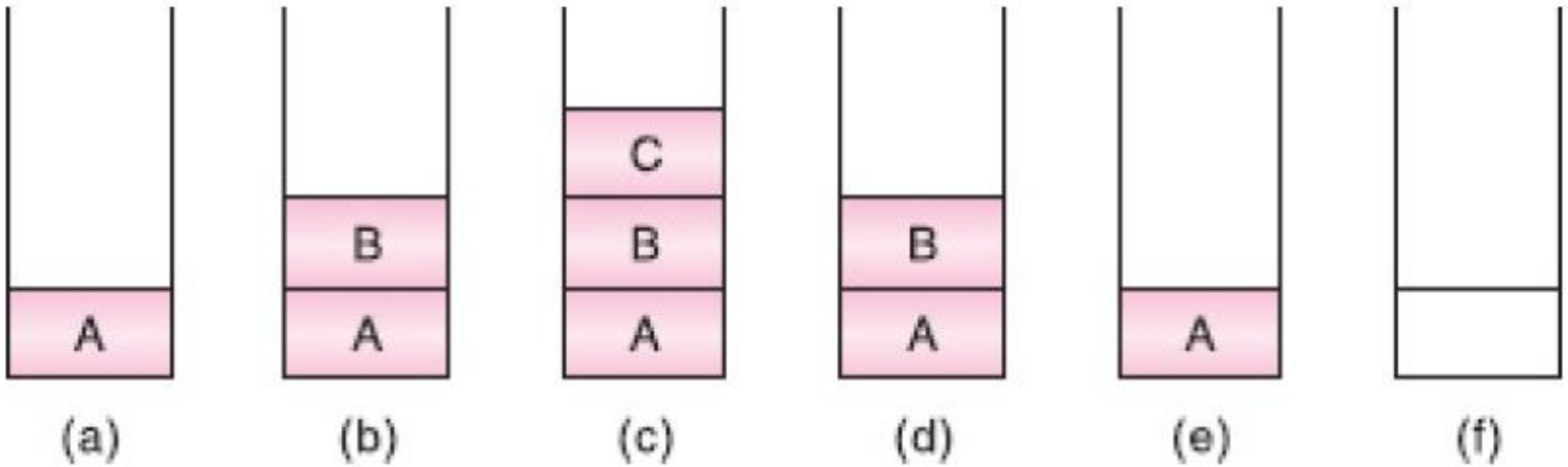


POP Operation

Stack Operation



Stack Operation



Miscellaneous

What happens if we try to pop an item off the stack when the stack is empty?

This is called a stack underflow. The pop method needs some way of telling us that this has happened.

Applications of Stacks

Direct applications:

- ✓ Page-visited history in a Web browser
- ✓ Undo sequence in a text editor
- ✓ Chain of method calls
- ✓ Validate XML
- ✓ Expression conversion and evaluation

Indirect applications:

- ✓ Auxiliary data structure for algorithms
- ✓ Component of other data structures

Stack - a Very Simple Application

We can use a stack to reverse the letters in a word.

How?

- Read each letter in the word and push it onto the stack
- When you reach the end of the word, pop the letters off the stack and print them out.

Sample Problems on Stack

6.1 Consider the following stack of characters, where STACK is allocated N = 8 memory cells:

STACK: A, C, D, F, K, ___, ___, ___,

(For notational convenience, we use "___" to denote an empty memory cell.) Describe the stack as the following operations take place: **(a)** POP(STACK, ITEM)

(b) POP(STACK, ITEM)
(c) PUSH(STACK, L)
(d) PUSH(STACK, P)
(e) POP(STACK, ITEM)
(f) PUSH(STACK, R)
(g) PUSH(STACK, S)
(h) POP(STACK, ITEM)

Sample Problems on Stack

The POP procedure always deletes the top element from the stack, and the PUSH procedure always adds the new element to the top of the stack. Accordingly: **(a)** STACK: A, C, D, F, ___, ___, ___, ___

(b) STACK: A, C, D, ___, ___, ___, ___, ___
(c) STACK: A, C, D, L, ___, ___, ___, ___
(d) STACK: A, C, D, L, P, ___, ___, ___
(e) STACK: A, C, D, L, ___, ___, ___, ___
(f) STACK: A, C, D, L, R, ___, ___, ___
(g) STACK: A, C, D, L, R, S, ___, ___
(h) STACK: A, C, D, L, R, ___, ___, ___

Sample Problems on Stack

6.2 Consider the data in Problem 6.1. (a) When will overflow occur? (b) When will C be deleted before D?

(a) Since STACK has been allocated N = 8 memory cells, overflow will occur when STACK contains 8 elements and there is a PUSH operation to add another element to STACK.

(b) Since STACK is implemented as a stack, C will never be deleted before D.

Sample Problems on Stack

6.3 Consider the following stack, where STACK is allocated N = 6 memory cells: ______

STACK: AAA, DDD, EEE, FFF, GGG, ______

Describe the stack as the following operations take place: (a) PUSH(STACK, KKK), (b) POP(STACK, ITEM), (c) PUSH(STACK, LLL), (d) PUSH(STACK, SSS), (e) POP(STACK, ITEM) and (f) PUSH(STACK, TTT).

Sample Problems on Stack

(a) KKK is added to the top of STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, KKK

(b) The top element is removed from STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, ______

(c) LLL is added to the top of STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, LLL

(d) Overflow occurs, since STACK is full and another element SSS is to be added to STACK.

No further operations can take place until the overflow is resolved—by adding additional space for STACK, for example.

Infix, Postfix and Prefix

Infix notation:
Operators are written in-between their operands.
This is the usual way we write expressions.

X + Y

Postfix notation (also known as "Reverse Polish notation"):
Operators are written after their operands.

X Y +

Prefix notation (also known as "Polish notation"):
Operators are written before their operands.

+ X Y

Infix, Postfix and Prefix

Infix notation:
Operators are written in-between their operands.
This is the usual way we write expressions.

A * (B + C) / D

Postfix notation (also known as "Reverse Polish notation"):
Operators are written after their operands.

A B C + * D /

Prefix notation (also known as "Polish notation"):
Operators are written before their operands.

/ * A + B C D

Why Infix, Postfix and Prefix?

Why do we need 3 different expressions ?

- Infix expressions are human readable but not efficient for machine reading
- Prefix and Postfix do not need the concept of precedence and associativity hence it becomes highly efficient for machines to parse expressions in prefix or postfix formats.

Infix to Postfix Conversion using Stack

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Impl

<table>
<tr><th>Operator</th><th>Description</th><th>Associativity</th><th>Rank</th></tr>
<tr><td>( )
[ ]</td><td>Function call
Aray element reference</td><td>Left to right</td><td>1</td></tr>
<tr><td>+
–
++
– –
!
~
*
&
sizeof
(type)</td><td>Unary plus
Unary minus
Increment
Decrement
Logical negation
Ones complement
Pointer reference (indirection)
Address
Size of an object
Type cast (conversion)</td><td>Right to left</td><td>2</td></tr>
<tr><td>*
/
%</td><td>Multiplication
Division
Modulus</td><td>Left to right</td><td>3</td></tr>
<tr><td>+
–</td><td>Addition
Subtraction</td><td>Left to right</td><td>4</td></tr>
<tr><td><<
>></td><td>Left shift
Right shift</td><td>Left to right</td><td>5</td></tr>
<tr><td><
<=
>
>=</td><td>Less than
Less than or equal to
Greater than
Greater than or equal to</td><td>Left to right</td><td>6</td></tr>
<tr><td>==
!=</td><td>Equality
Inequality</td><td>Left to right</td><td>7</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>Left to right</td><td>8</td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>Left to right</td><td>9</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>Left to right</td><td>10</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>Left to right</td><td>11</td></tr>
<tr><td>||</td><td>Logical OR</td><td>Left to right</td><td>12</td></tr>
<tr><td>?:</td><td>Conditional expression</td><td>Right to left</td><td>13</td></tr>
<tr><td>=
* = /= %=
+= –= &=
^= |=
<<= >>=</td><td>Assignment operators</td><td>Right to left</td><td>14</td></tr>
<tr><td>,</td><td>Comma operator</td><td>Left to right</td><td>15</td></tr>
</table>

Infix to Postfix

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Infix to Postfix

6.10 Consider the following infix expression Q:

$$Q: ((A + B) * D) \uparrow (E - F)$$

Use Algorithm 6.6 to translate Q into its equivalent postfix expression P.

Infix to Postfix

Q: ((A + B) * D) ↑ (E – F)

Symbol	STACK	Expression P
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	A B
)	((	A B +
*	((*	A B +
D	((*	A B + D
)	(	A B + D *
↑	(↑	A B + D *
(	(↑ (	A B + D *
E	(↑ (	A B + D * E
–	(↑ (–	A B + D * E
F	(↑ (–	A B + D * E F
)	(↑	A B + D * E F –
)		A B + D * E F – ↑

Infix to Postfix

Consider the following arithmetic infix expression Q:

Q: A + (B * C – (D / E ↑ F) * G) * H

We simulate Algorithm 6.6 to transform Q into its equivalent postfix expression P.

First we push “(“ onto STACK, and then we add “)” to the end of Q to obtain:

Q:	A	+	(	B	*	C	–	(	D	/	E	↑	F	)	*
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
											G	)	*	H	)
											(16)	(17)	(18)	(19)	(20)

Q: A + (B * C – (D / E ↑ F) * G) * H

Infix to Postfix

Symbol Scanned		STACK	Expression P
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+ (	A
(4)	B	(+ (	A B
(5)	*	(+ (*	A B
(6)	C	(+ (*	A B C
(7)	–	(+ (–	A B C *
(8)	(	(+ (– (	A B C *
(9)	D	(+ (– (	A B C * D
(10)	/	(+ (– (/	A B C * D
(11)	E	(+ (– (/	A B C * D E
(12)	↑	(+ (– (/ ↑	A B C * D E
(13)	F	(+ (– (/ ↑	A B C * D E F
(14)	)	(+ (–	A B C * D E F ↑ /
(15)	*	(+ (– *	A B C * D E F ↑ /
(16)	G	(+ (– *	A B C * D E F ↑ / G
(17)	)	(+	A B C * D E F ↑ / G * –
(18)	*	(+ *	A B C * D E F ↑ / G * –
(19)	H	(+ *	A B C * D E F ↑ / G * – H
(20)	)		A B C * D E F ↑ / G * – H * +

Postfix Expression Evaluation

6.9 Consider the postfix expression P in Problem 6.8. Evaluate P using Algorithm 6.5.

First add a sentinel right parenthesis at the end of P to obtain:

P: 12, 7, 3, –, /, 2, 1, 5, +, *, +,)

Postfix Expression Evaluation

Symbol	STACK
12	12
7	12, 7
3	12, 7, 3
–	12, 4
/	3
2	3, 2
1	3, 2, 1
5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15
)	15

Postfix Expression Evaluation

Example:

- Consider **P: 5, 6, 2, +, *, 12, 4, /, –**

Symbol Scanned	STACK
(1) 5	5
(2) 6	5, 6
(3) 2	5, 6, 2
(4) +	5, 8
(5) *	40
(6) 12	40, 12
(7) 4	40, 12, 4
(8) /	40, 3
(9) -	37
(10))	

- Postfix Expression: 6 2 3 + - 3 8 2 / + * 2 $ 3 +
- Steps for evaluation are as follows:

6 **<u>2 3 +</u>** - 3 8 2 / + * 2 $ 3 +

<u>6 5 -</u> 3 8 2 / + * 2 $ 3 +

1 3 **<u>8 2 /</u>** + * 2 $ 3 +

1 **<u>3 4 +</u>** * 2 $ 3 +

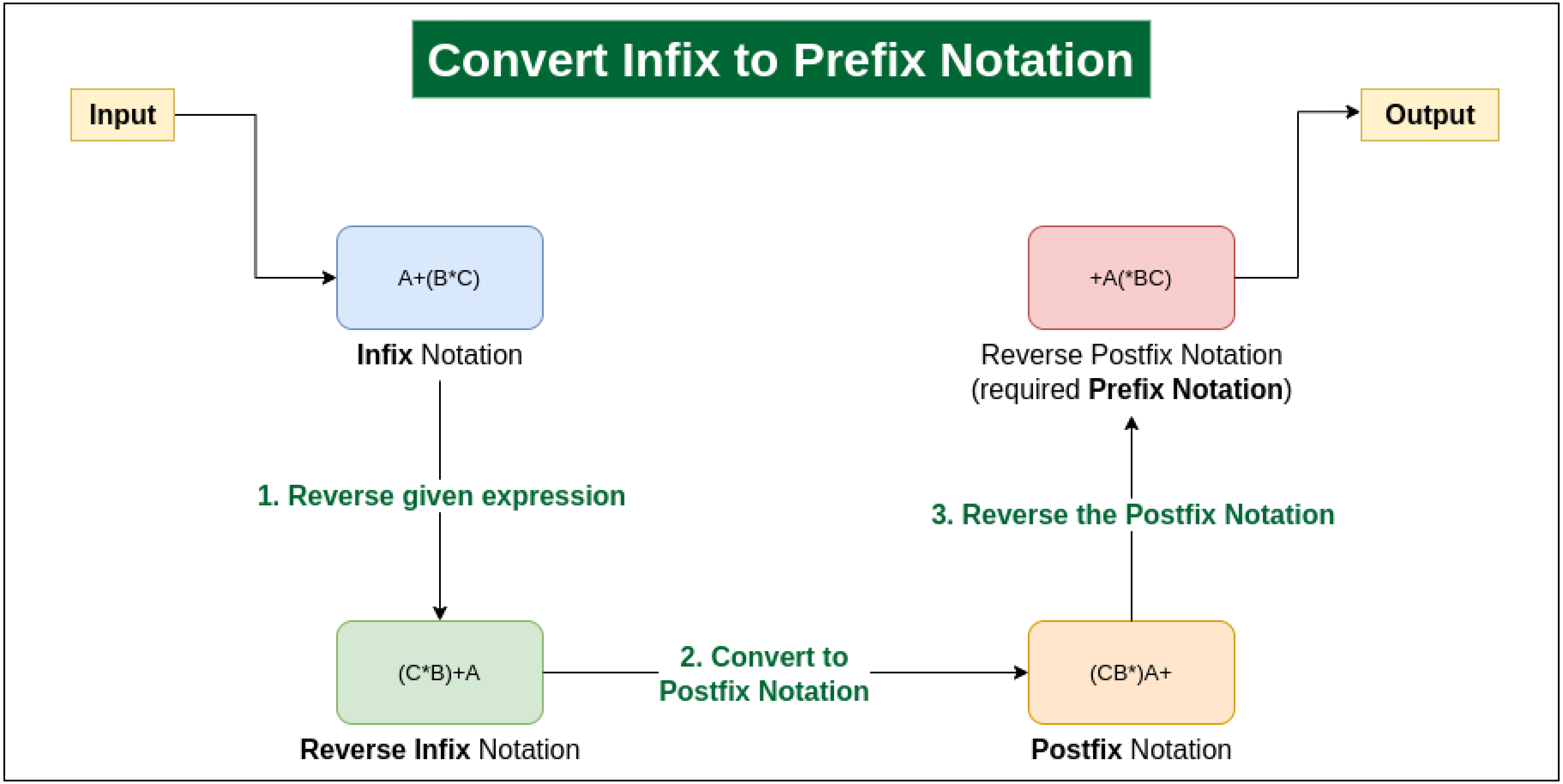
<u>1 7 *</u> 2 $ 3 +

<u>7 2 $</u> 3 +

<u>49 3 +</u>

52

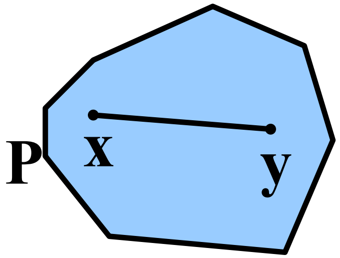
Infix to Prefix Expression



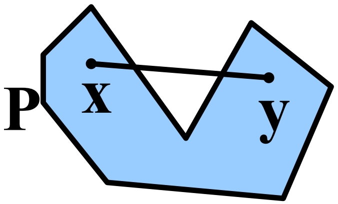
Graham Scan
–
Convex Hull

Convex vs. Concave

- A polygon P is convex if for every pair of points x and y in P, the line xy is also in P; otherwise, it is called concave.



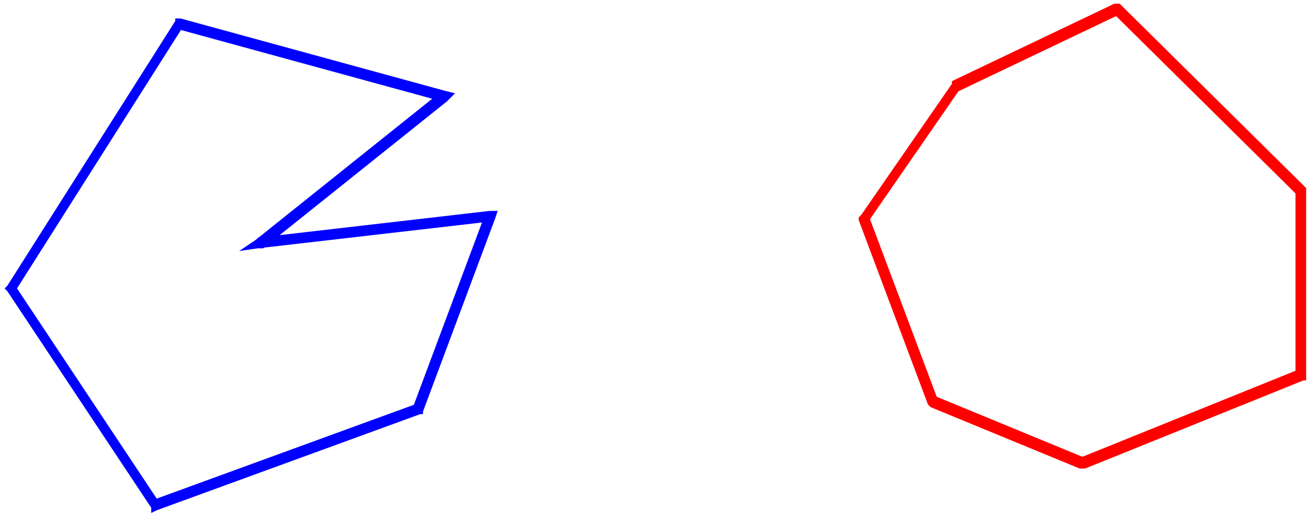
Convex



Concave

Convex vs. Concave

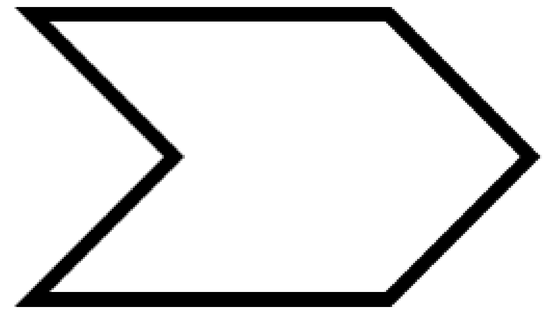
- A polygon P is convex if for every pair of points x and y in P, the line xy is also in P; otherwise, it is called concave.



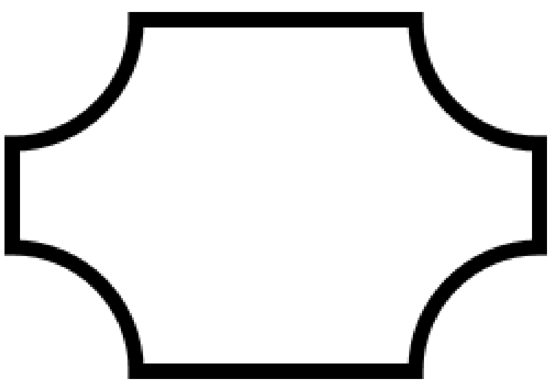
Convex vs. Concave

a)

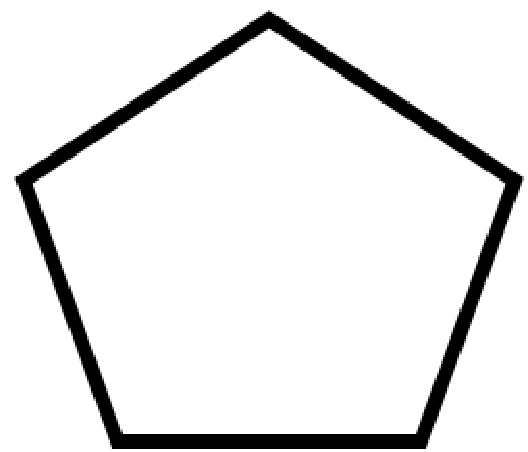
b)



c)



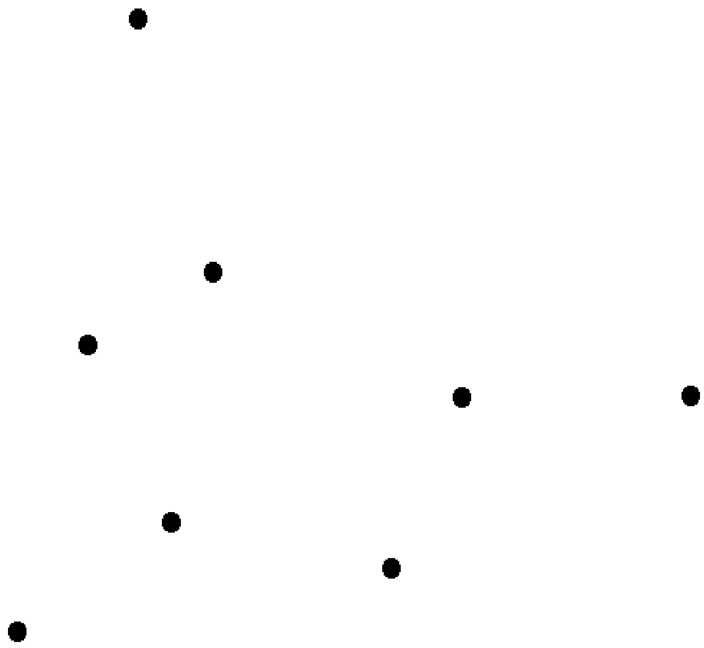
d)



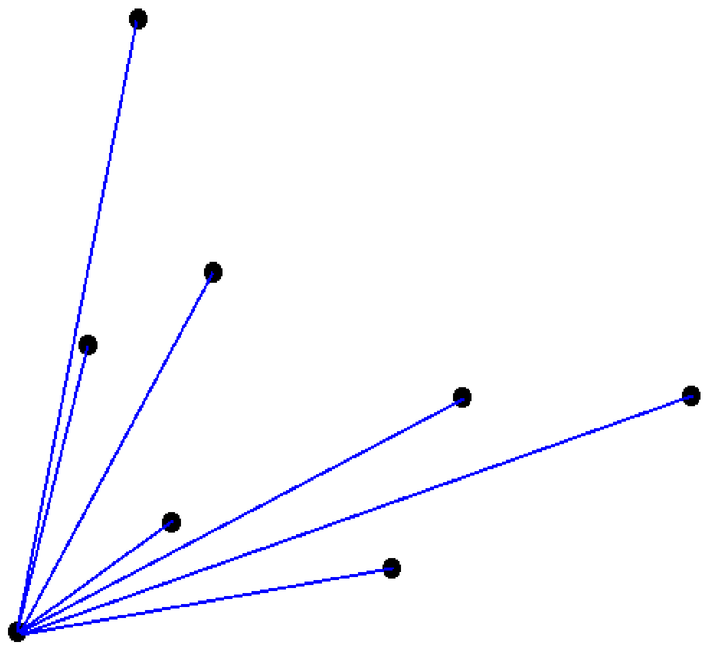
Graham's Scan for Convex Hull

- Start at point guaranteed to be on the hull. (the point with the minimum y value)
- Sort remaining points by polar angles of vertices relative to the first point.
- Go through sorted points, keeping vertices of points that have left turns and dropping points that have right turns.

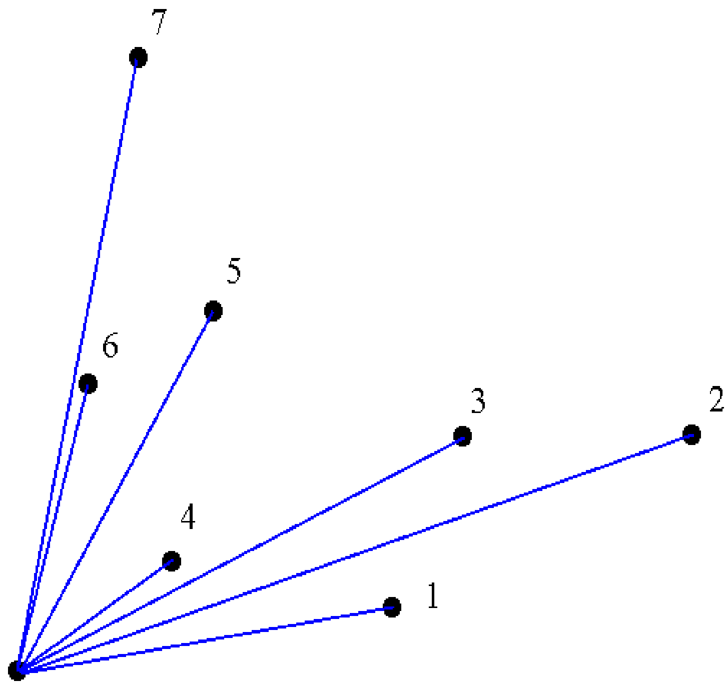
Graham's Scan for Convex Hull



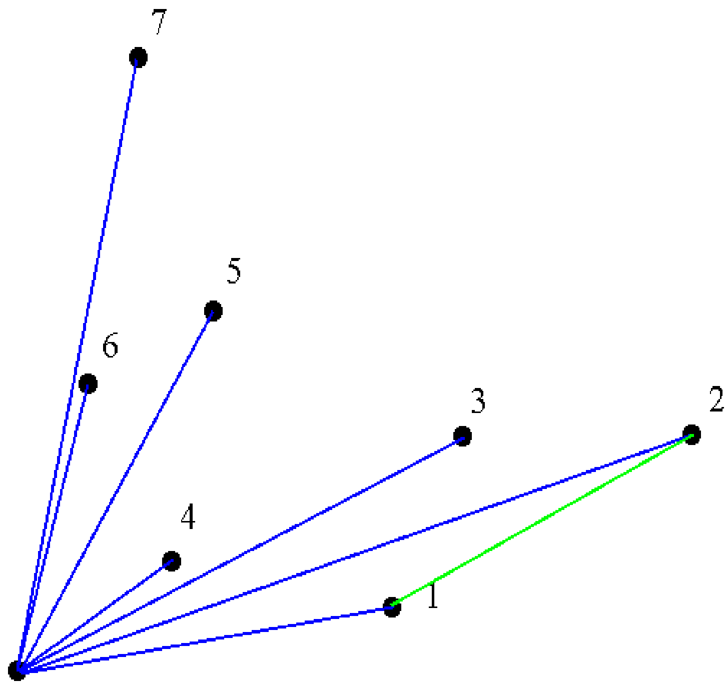
Graham’s Scan for Convex Hull



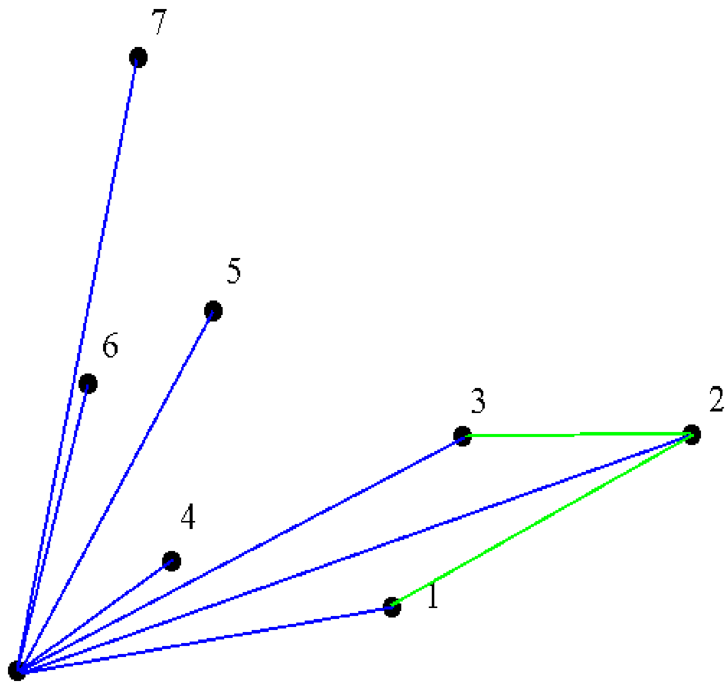
Graham’s Scan for Convex Hull



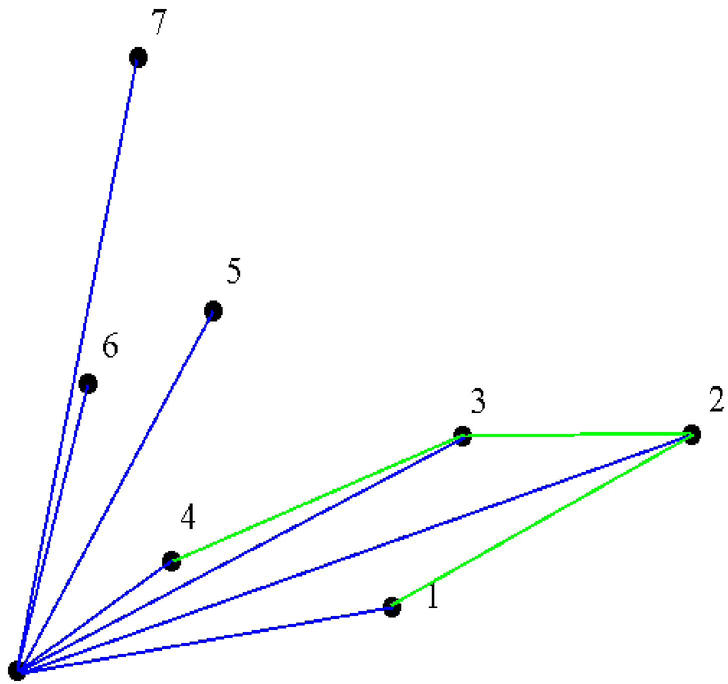
Graham's Scan for Convex Hull



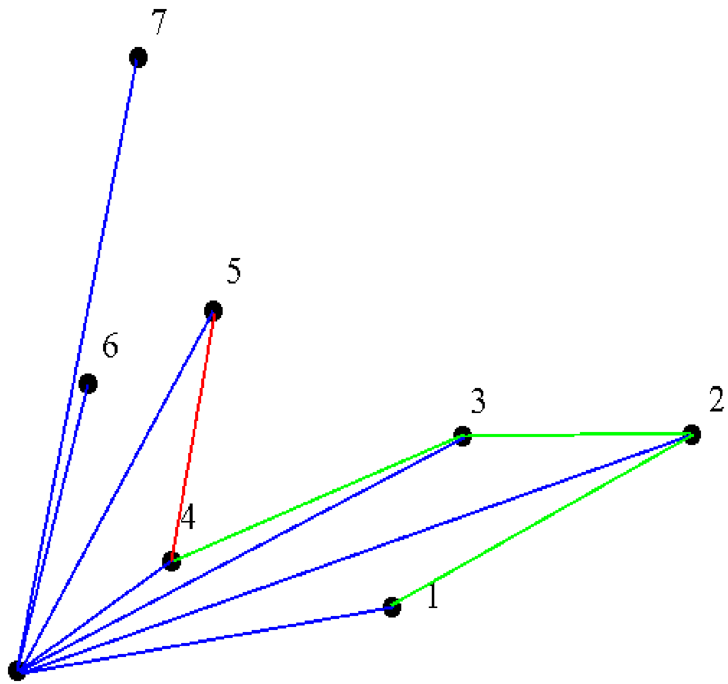
Graham’s Scan for Convex Hull



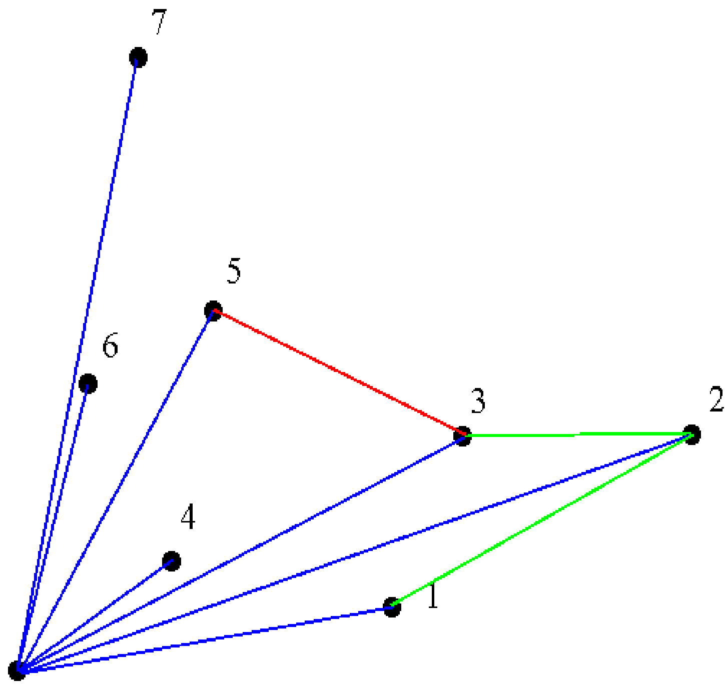
Graham’s Scan for Convex Hull



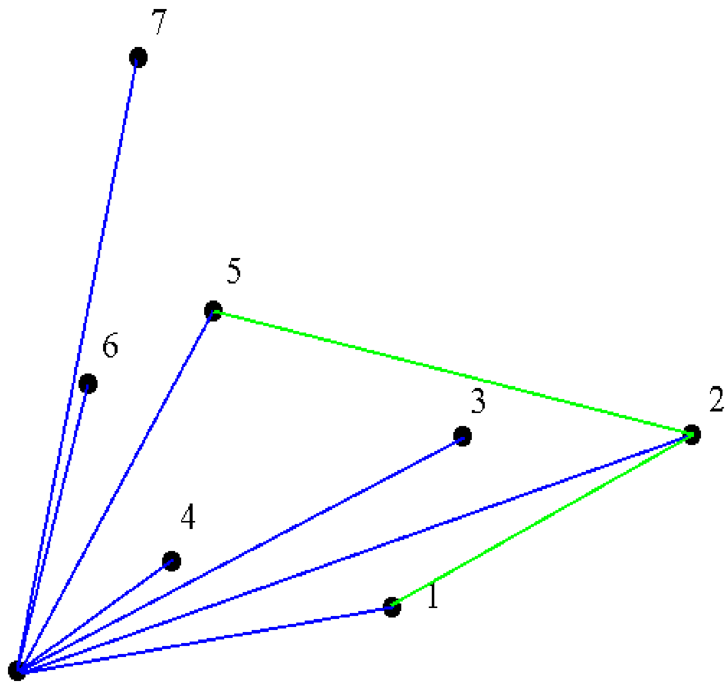
Graham’s Scan for Convex Hull



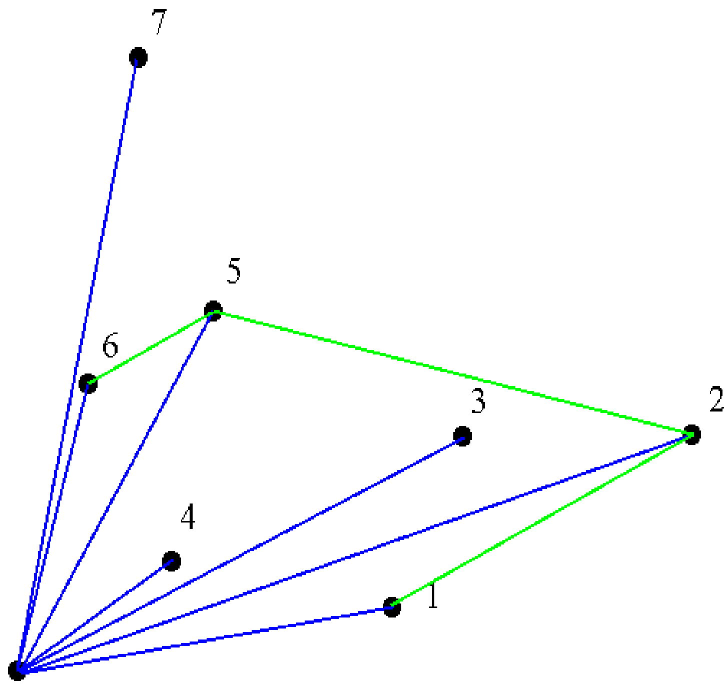
Graham’s Scan for Convex Hull



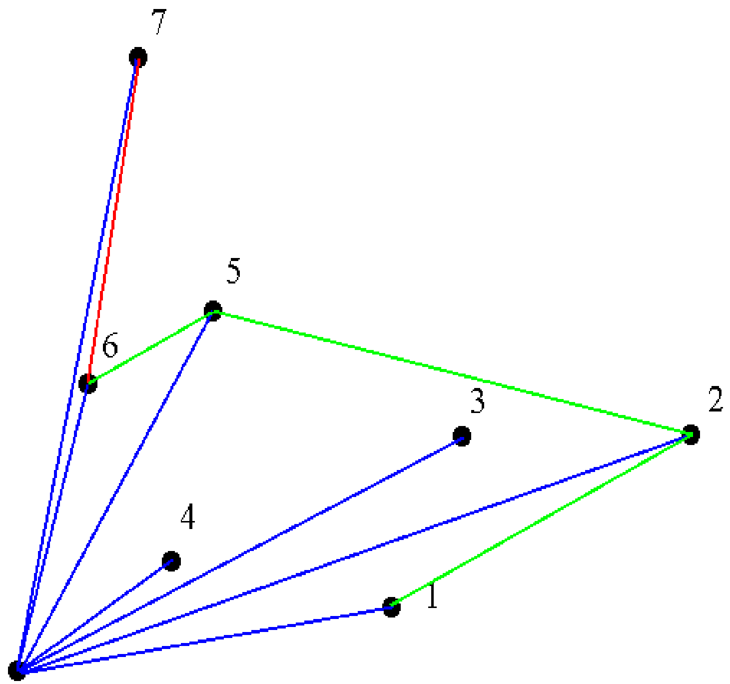
Graham’s Scan for Convex Hull



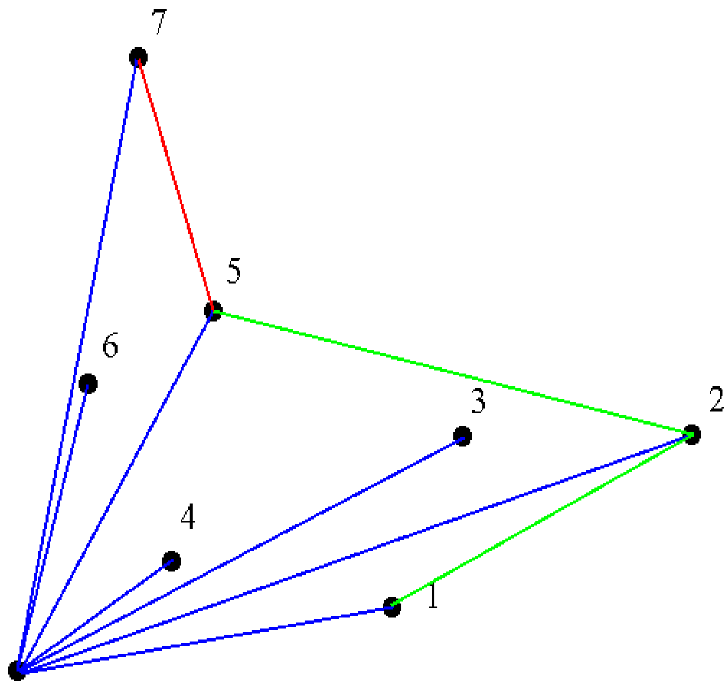
Graham's Scan for Convex Hull



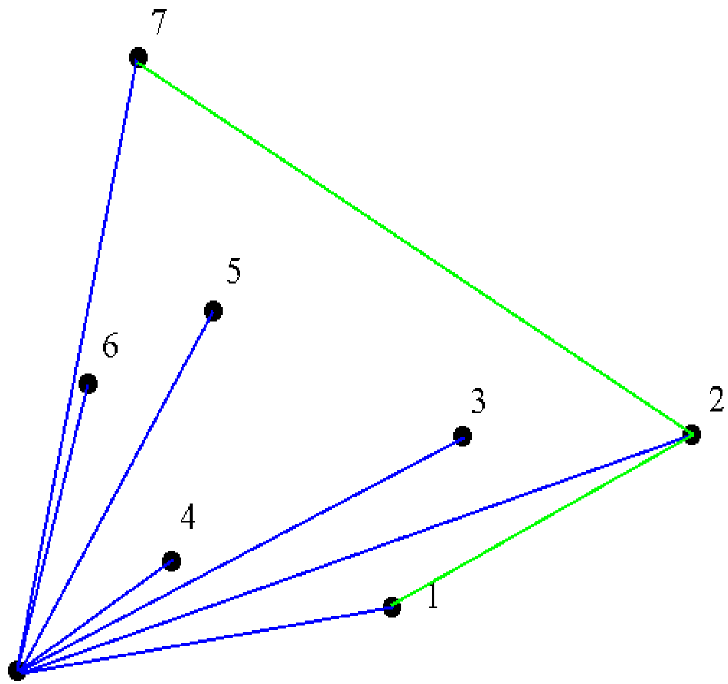
Graham’s Scan for Convex Hull



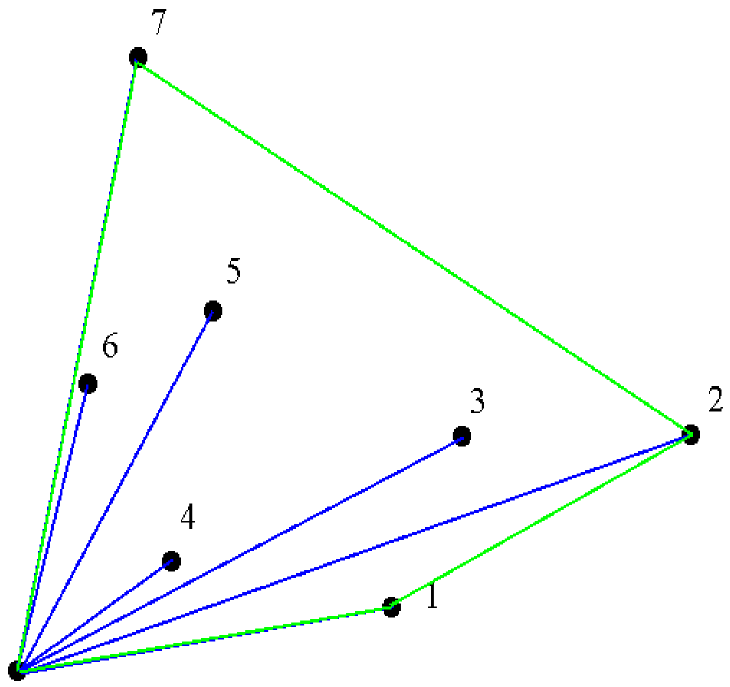
Graham's Scan for Convex Hull



Graham's Scan for Convex Hull



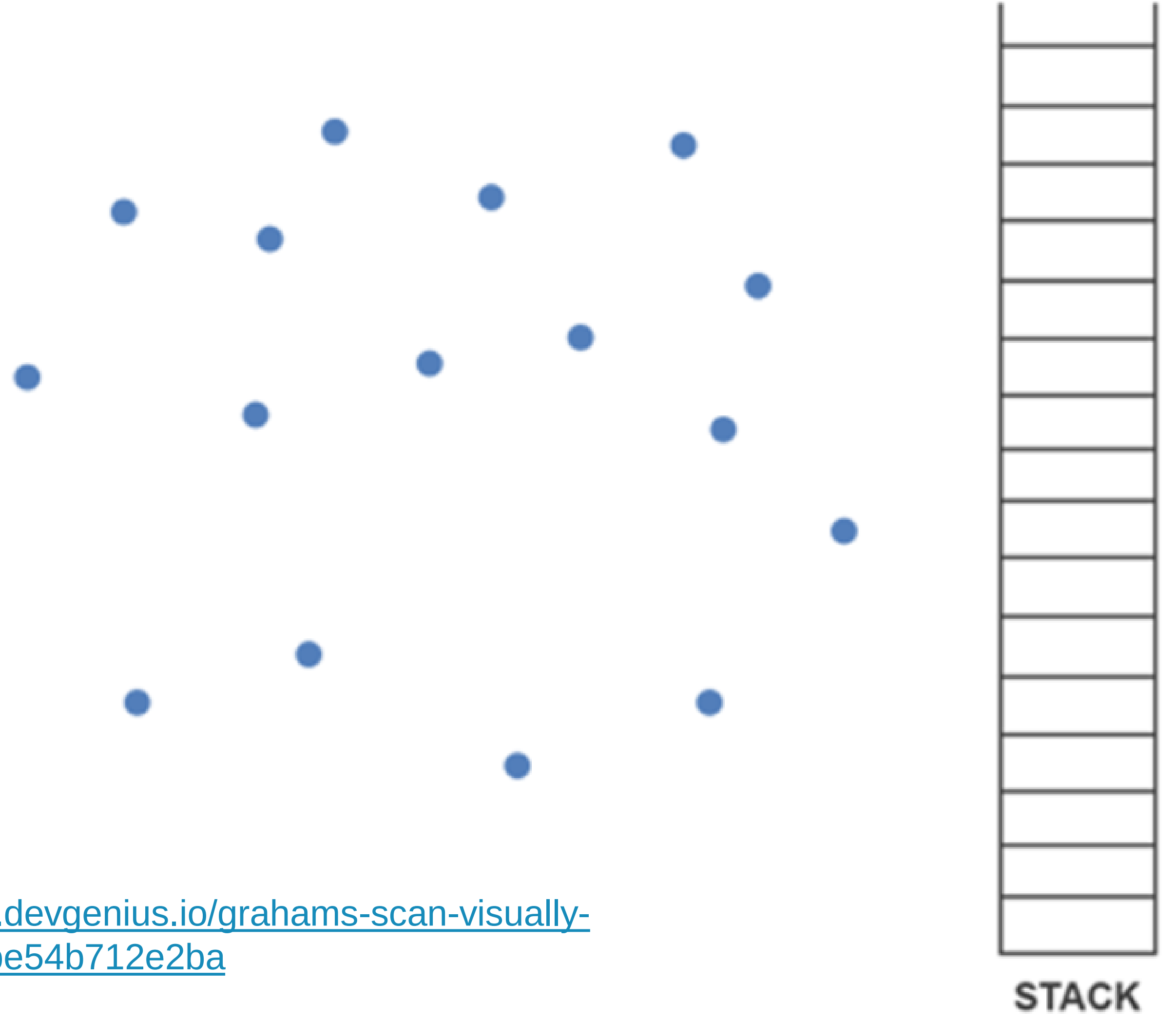
Graham's Scan for Convex Hull



Graham's Scan for Convex Hull

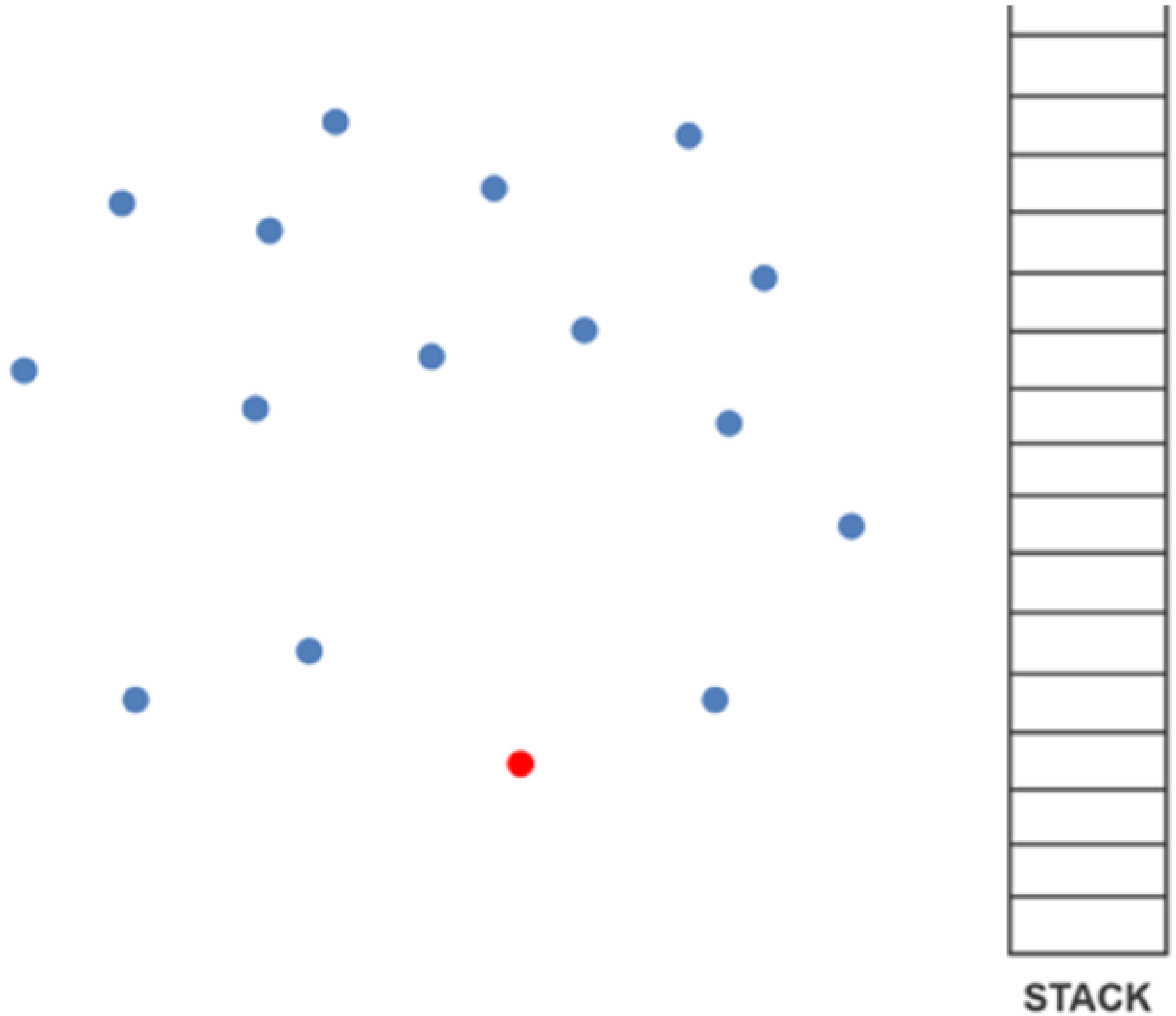
Let $A(x_1, y_1)$ and $B(x_2, y_2)$ be the endpoints of a directed line segment. A point $P(x, y)$ will be to the *left* of the line segment if the expression $C = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$ is positive (see Prob. 5.13). We say that the point is to the *right* of the line segment if this quantity is negative. If a point P is to the right of any one edge of a positively oriented, convex polygon, it is outside the polygon. If it is to the left of *every* edge of the polygon, it is inside the polygon.

Graham’s Scan for Convex Hull

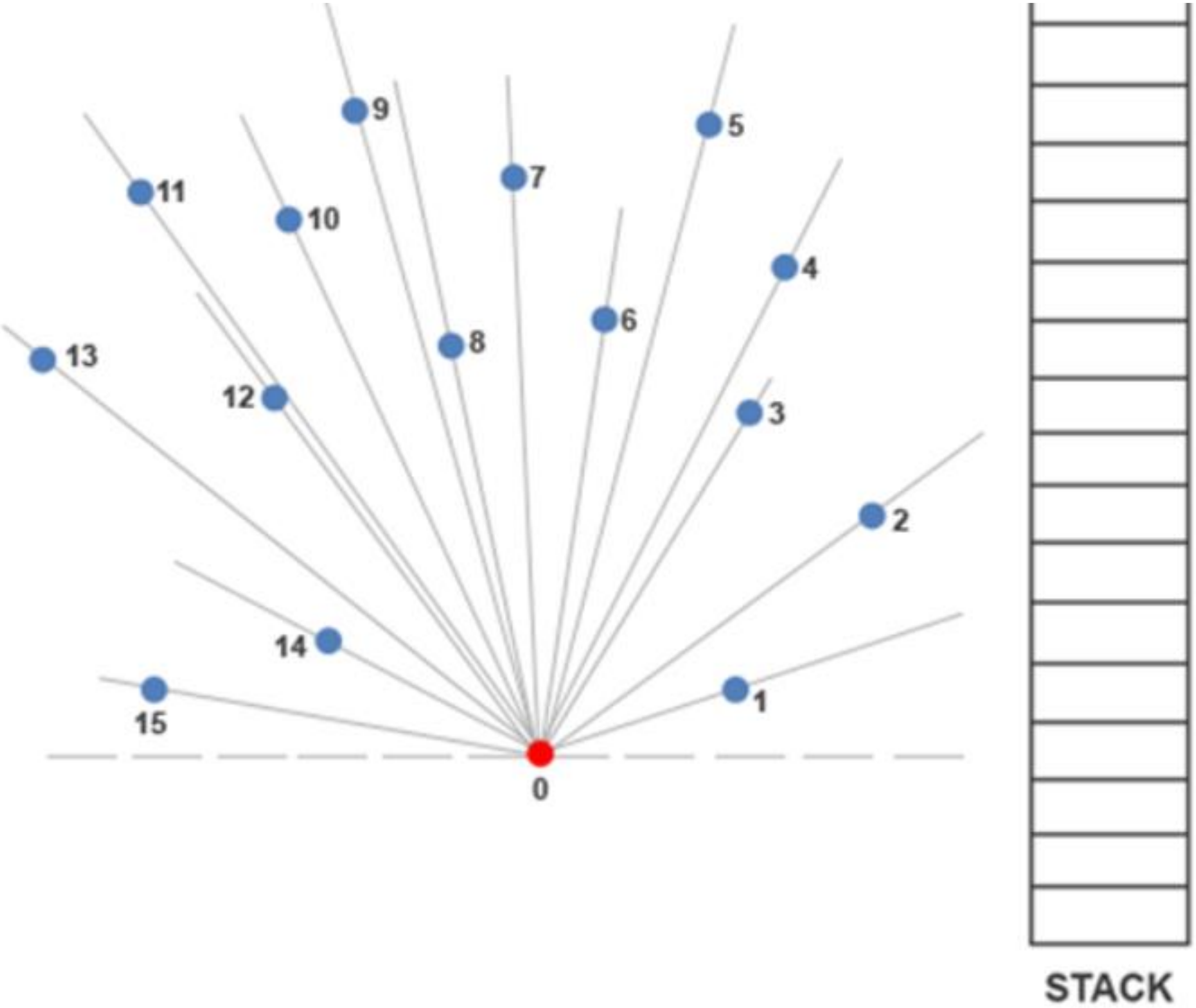


https://blog.devgenius.io/grahams-scan-visually-explained-be54b712e2ba

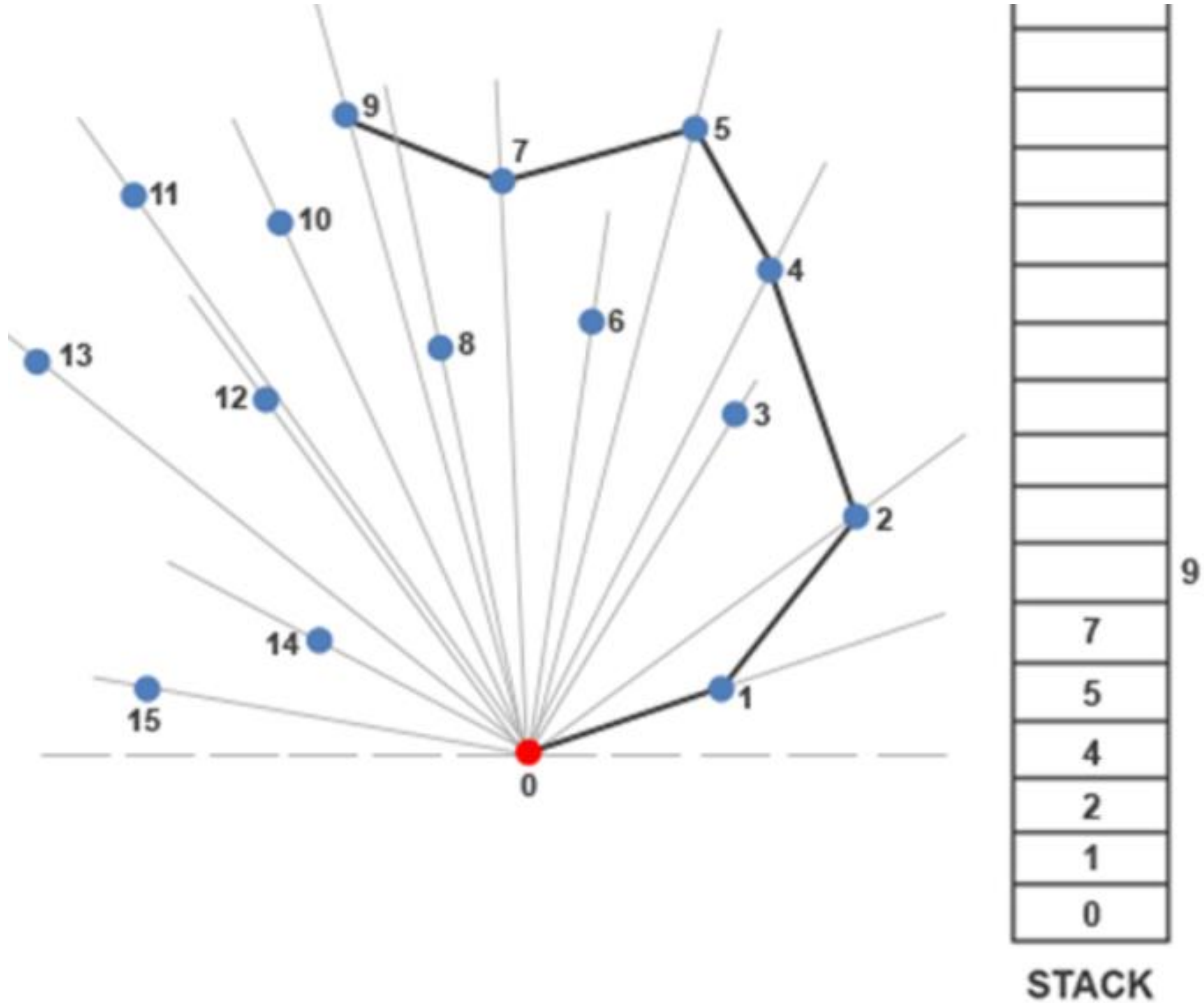
Graham’s Scan for Convex Hull



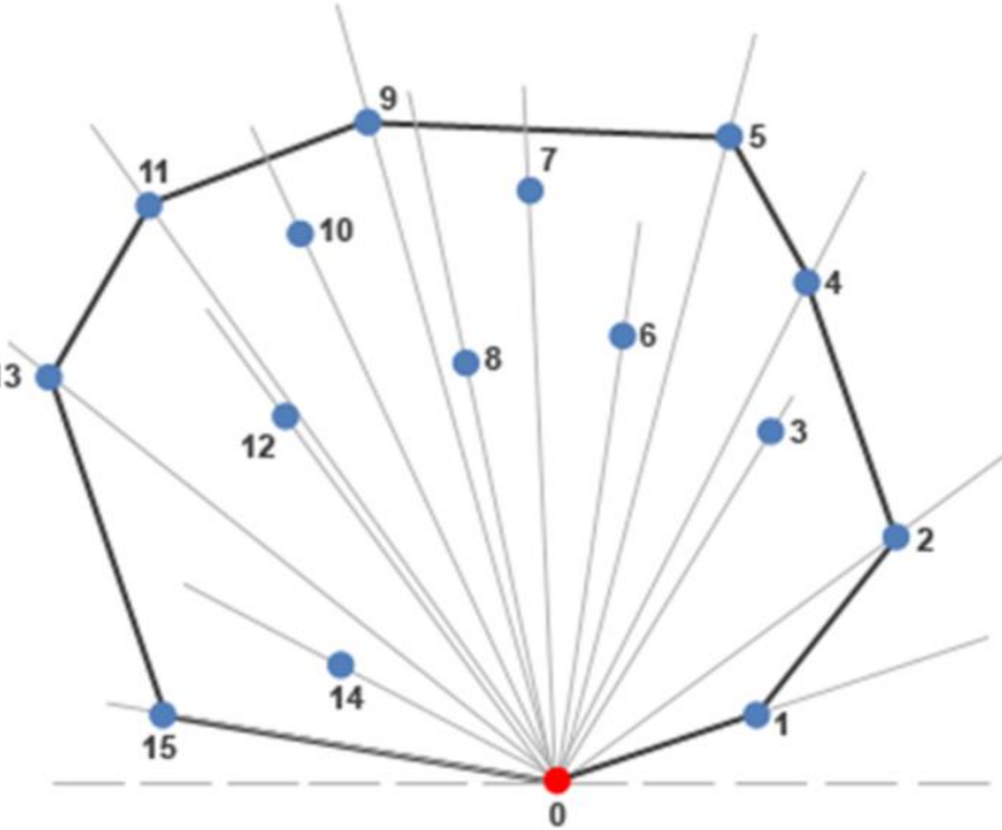
Graham's Scan for Convex Hull

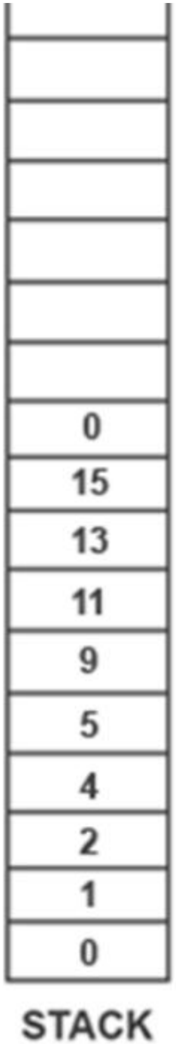


Graham's Scan for Convex Hull



Graham's Scan for Convex Hull

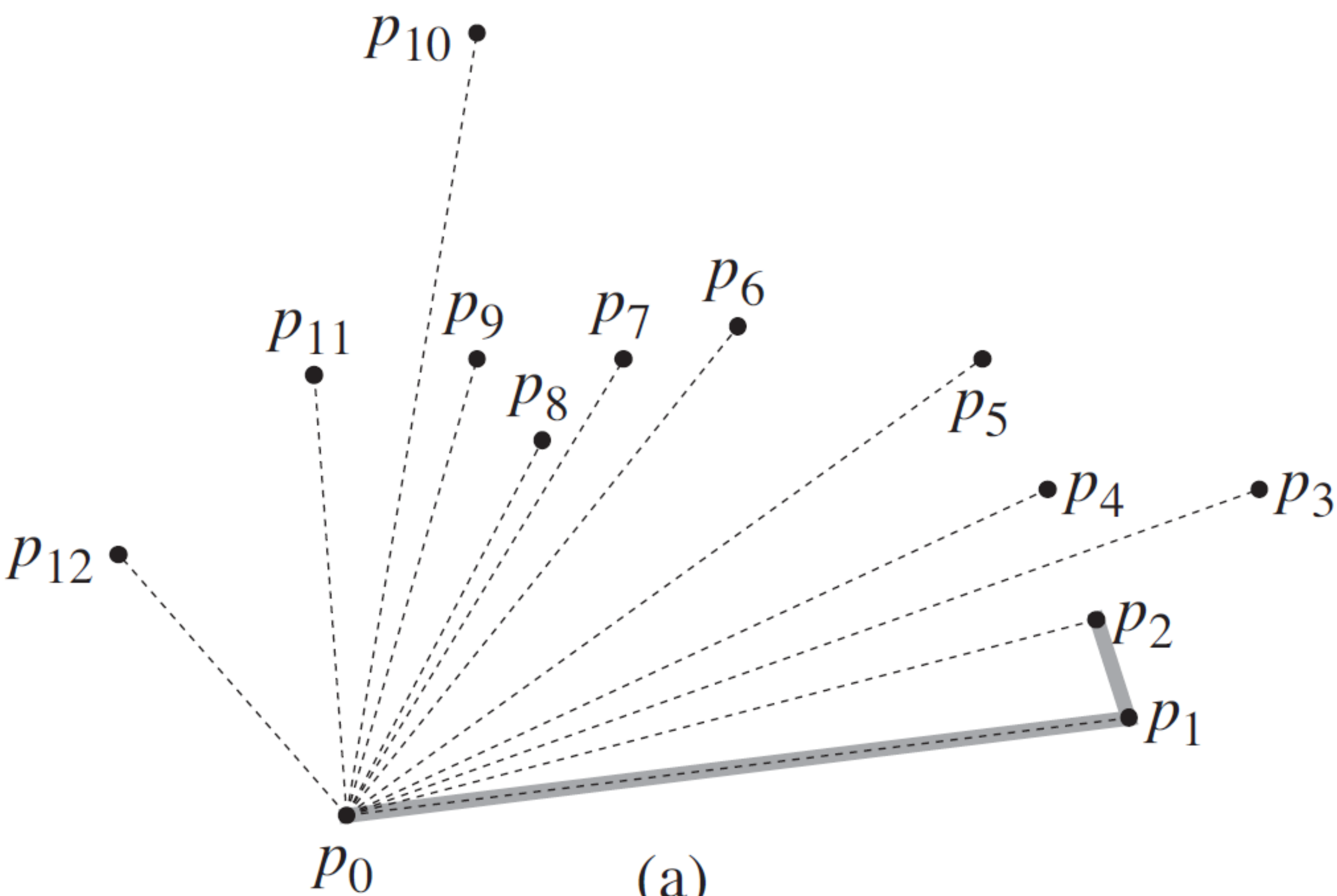




The points that are needed to create the convex hull are

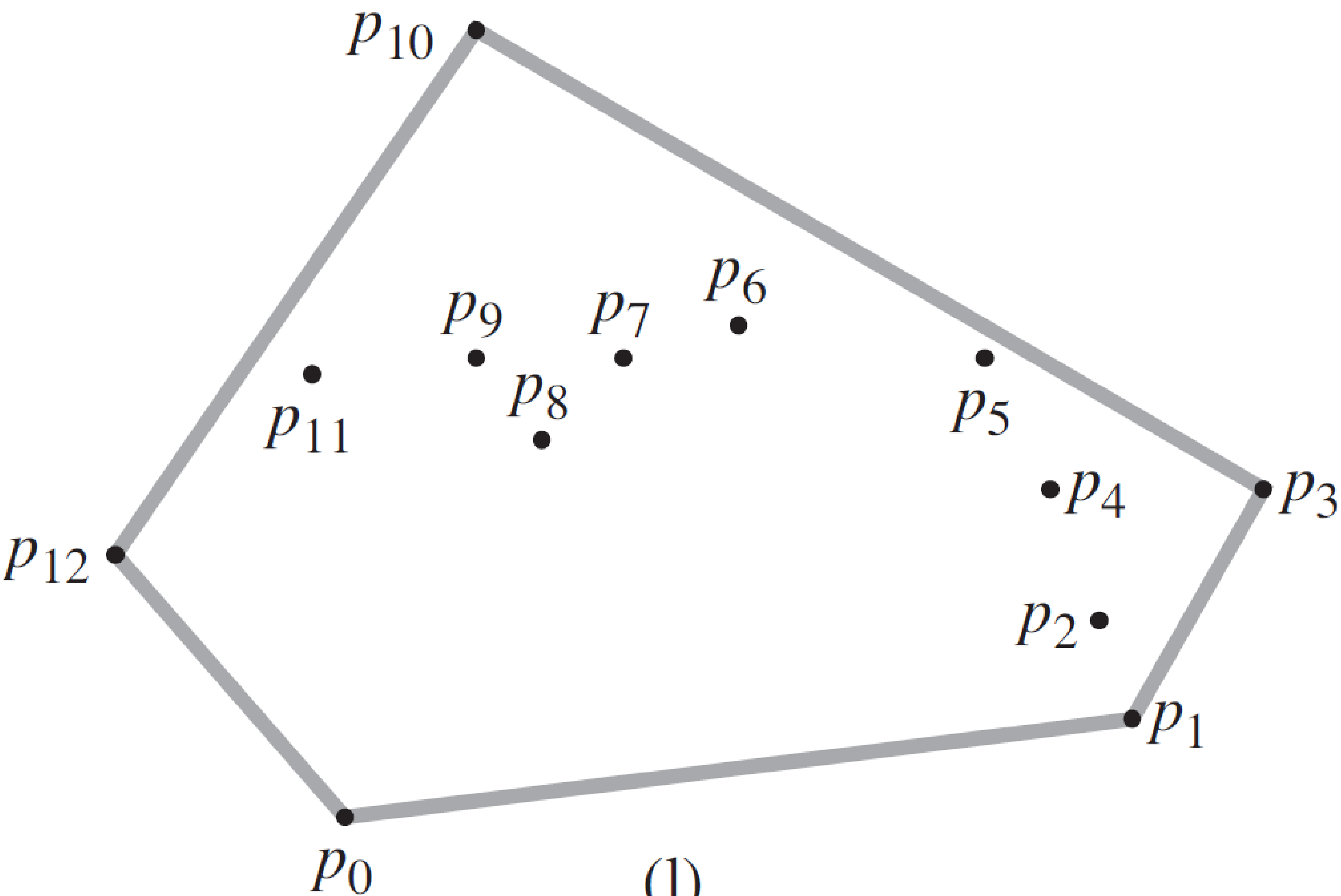
0–1–2–4–5–9–11–13–15

Graham's Scan for Convex Hull



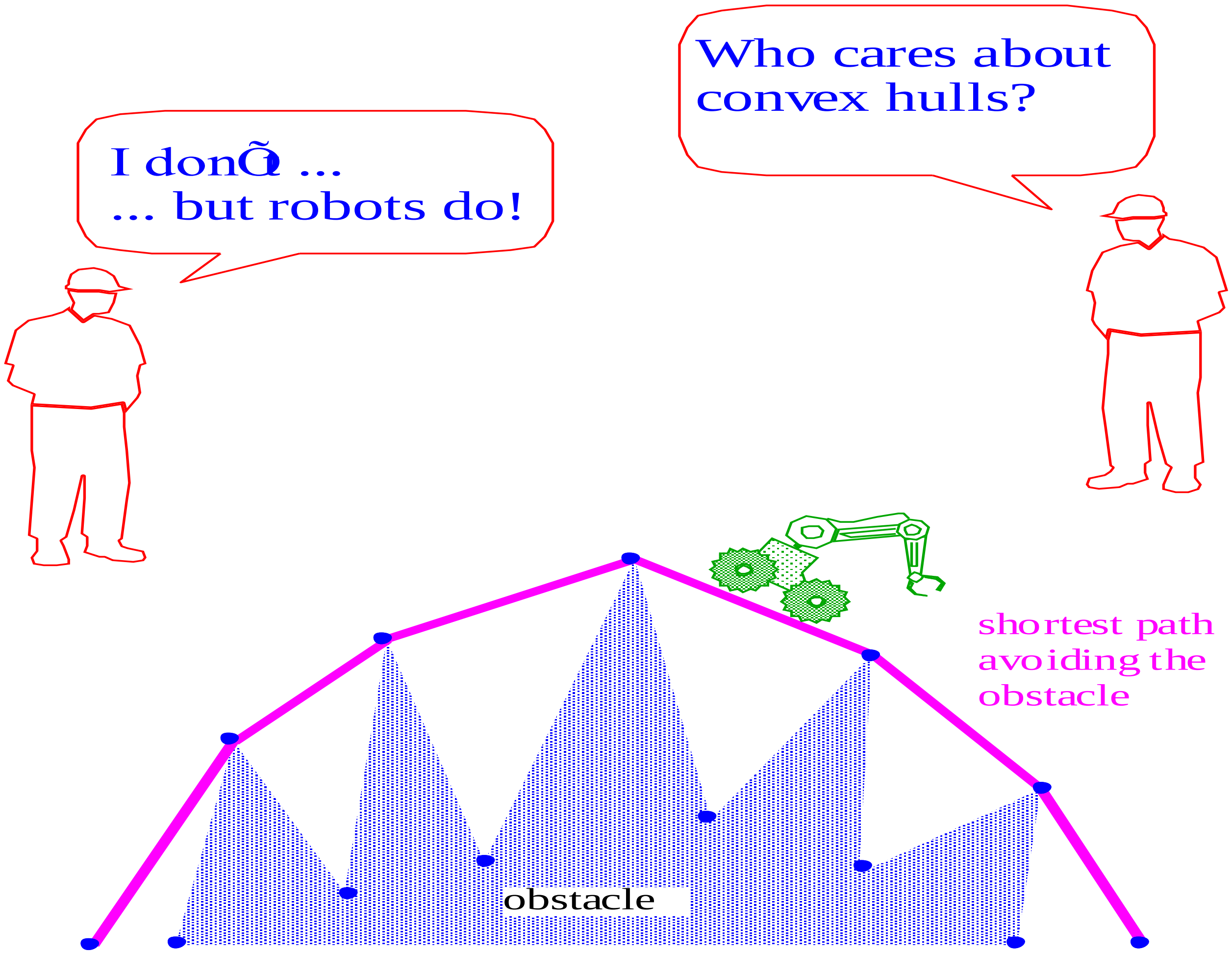
(a)

Graham's Scan for Convex Hull



(1)

Why Convex Hulls?




Thank You!