Evaluation of Expressions

Outline

- C-Expression
- Expression Tree
    - Definition & Evaluation
    - Building
    - Recursive Implementation
- Postfix Evaluation
    - Evaluation by Using a Stack
    - Conversion by Using a Stack

Expression in C

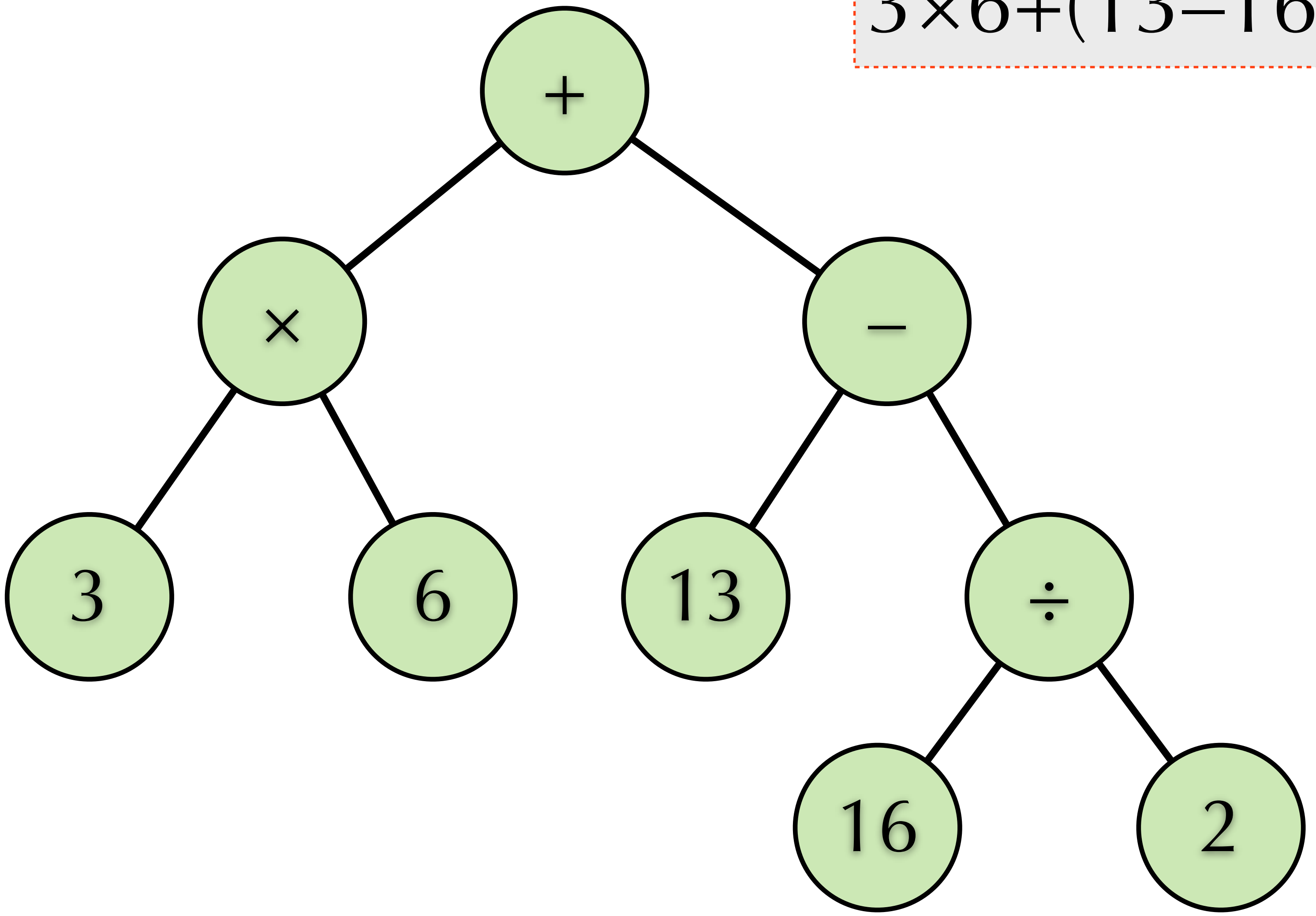
- Composed by operators and operands
  - Unary operator: 1 operand (ex: ++)
  - Binary operator: 2 operands (ex: <<)
  - Ternary operator: 3 operands (ex: ?:)
- Precedence:
  - Multiplication v.s. Addition
- Associativity:
  - Left-to-Right v.s. Right-to-Left

Expression Tree

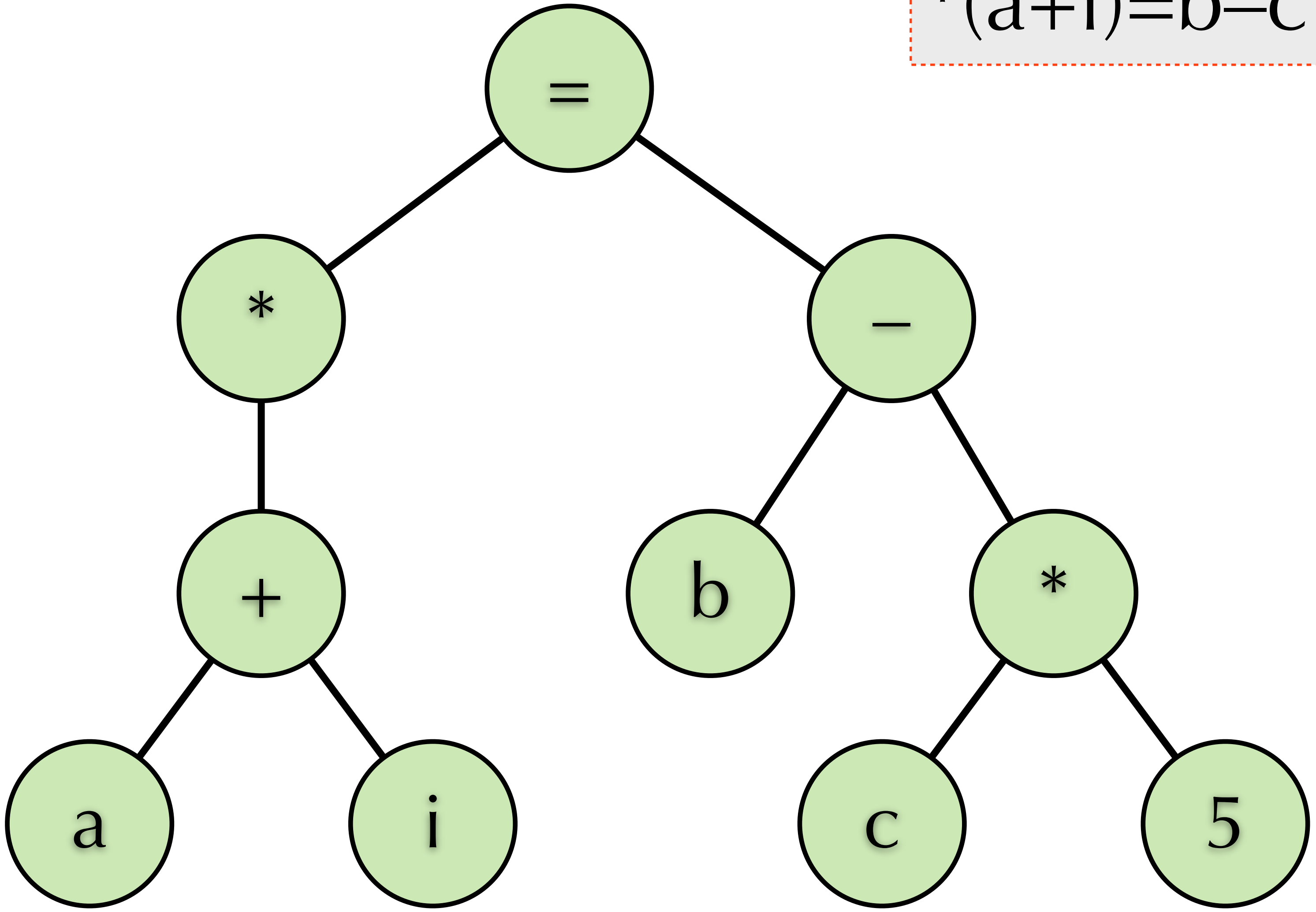
- A rooted tree
  - Internal node: Operator
  - Leaf: Operand
- Root: the operator of the last operation
- Evaluation Process:
  - Evaluate all subtrees of the root
  - Compute the result of the last operation
  - Can be easily implemented by recursion

Example

3×6+(13–16÷2)



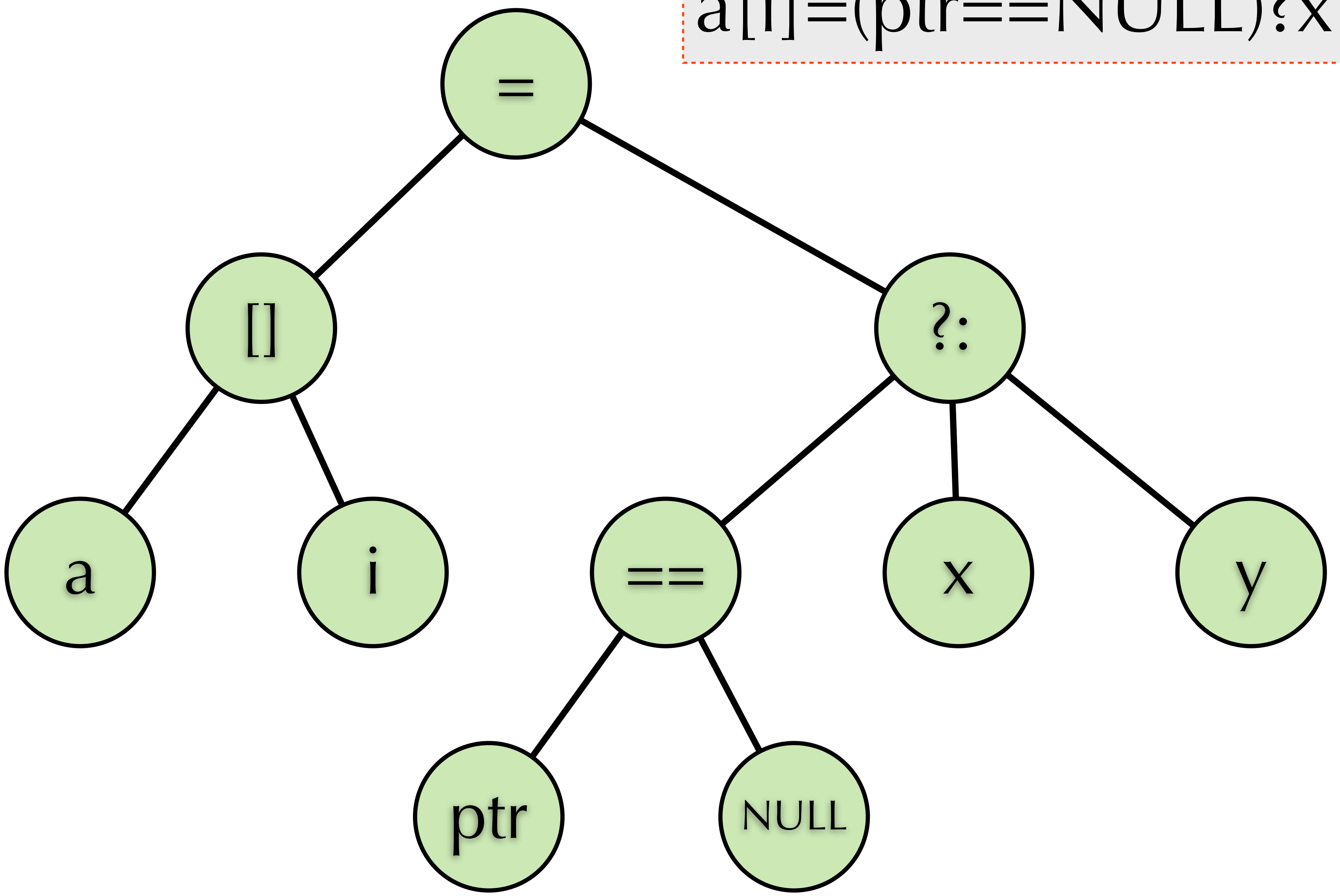
Example



*(a+i)=b–c*5

Example

```
a[i]=(ptr==NULL)?x:y
```

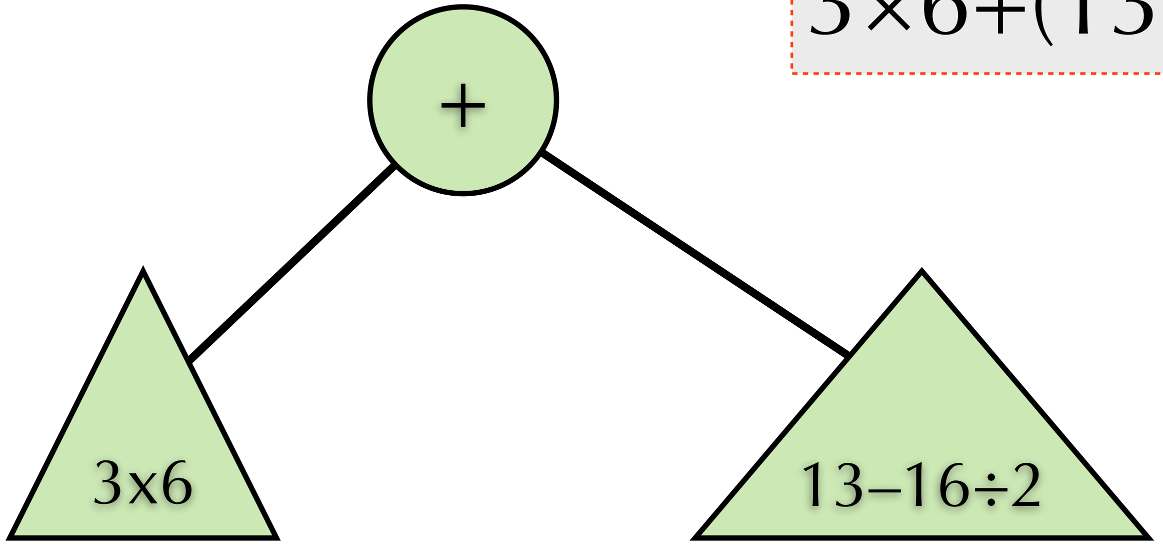


Building Expression Tree

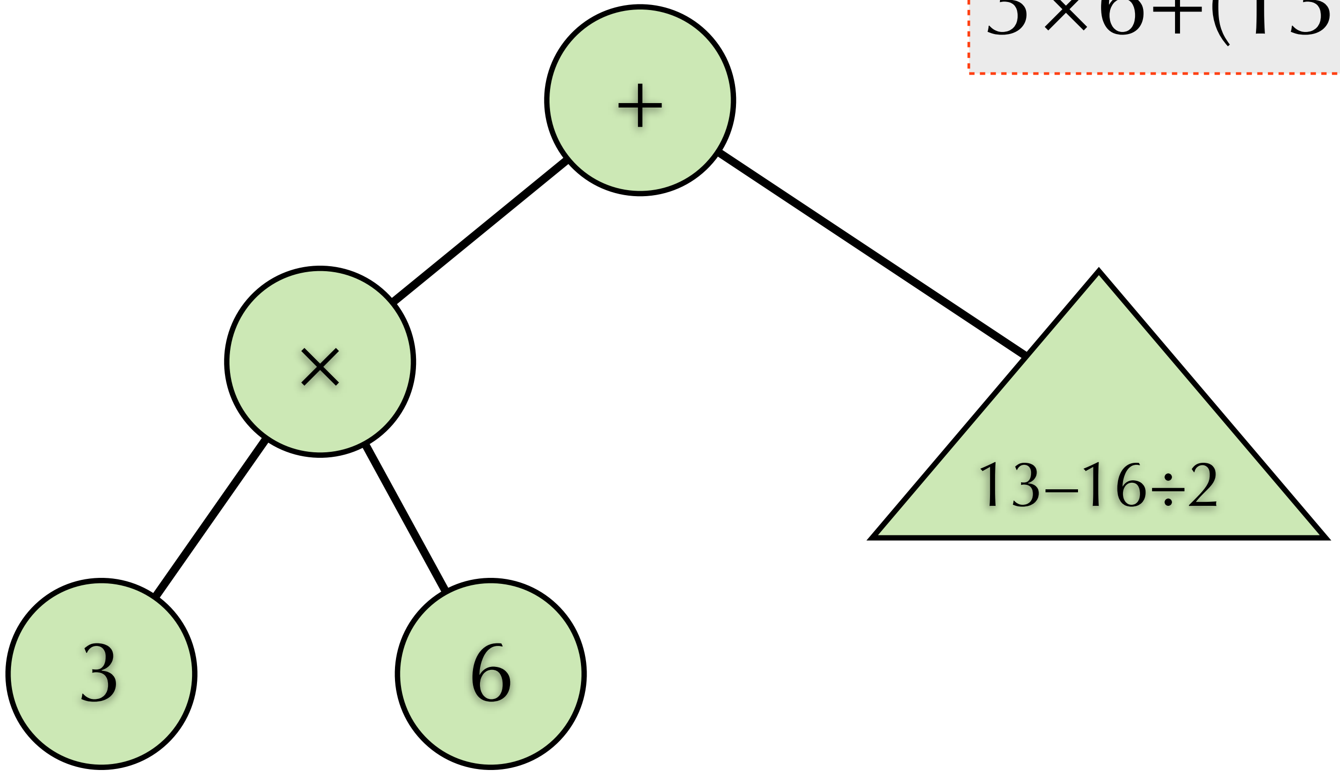
- Recursive algorithm
  - Terminal: If no operators exists, then the root is the operand.
  - Find out the last operation σ by checking precedence and associativity
  - Set root as the operator of σ.
  - Build the subtrees recursively. (Note: the constructions can be done in parallel)

Example

3×6+(13–16÷2)

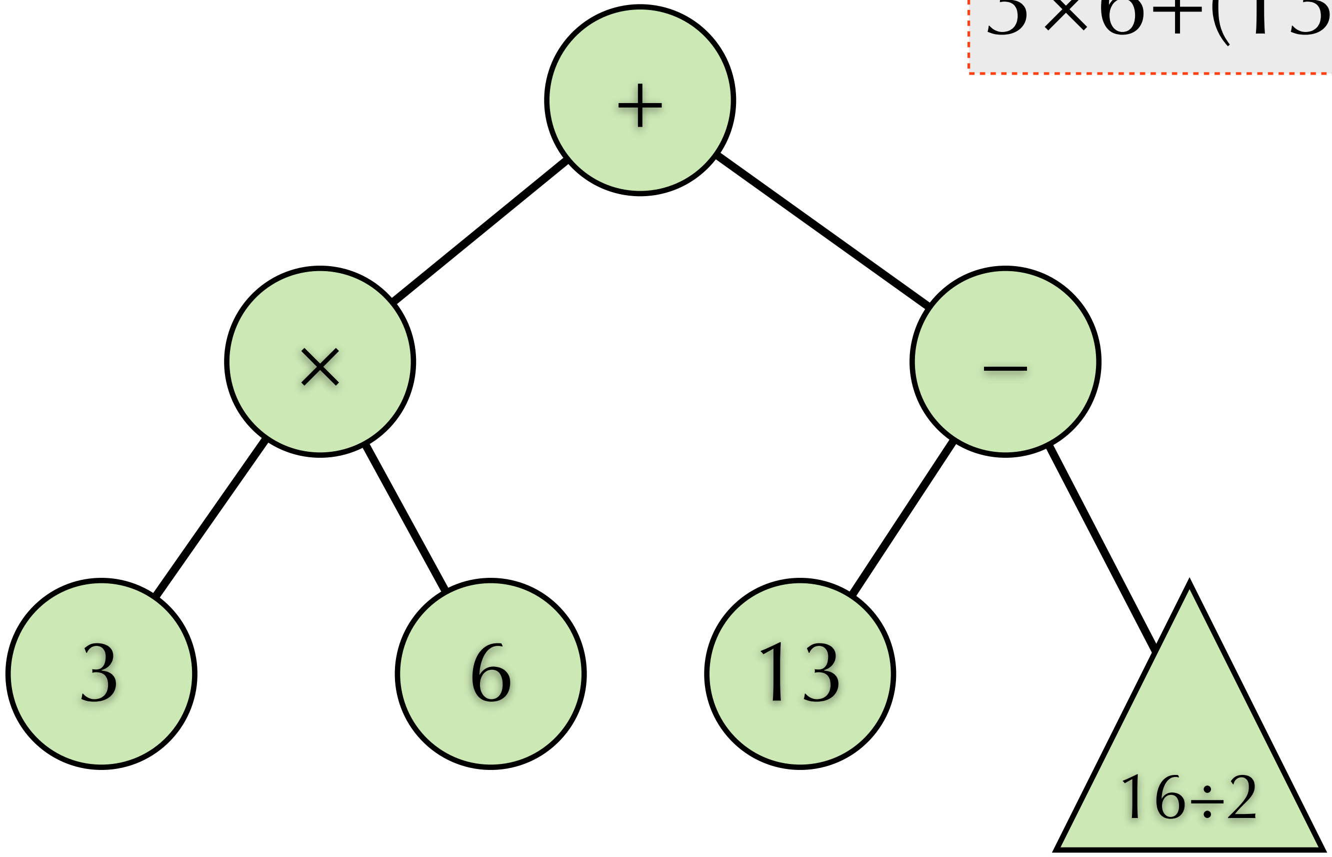


Example



3×6+(13–16÷2)

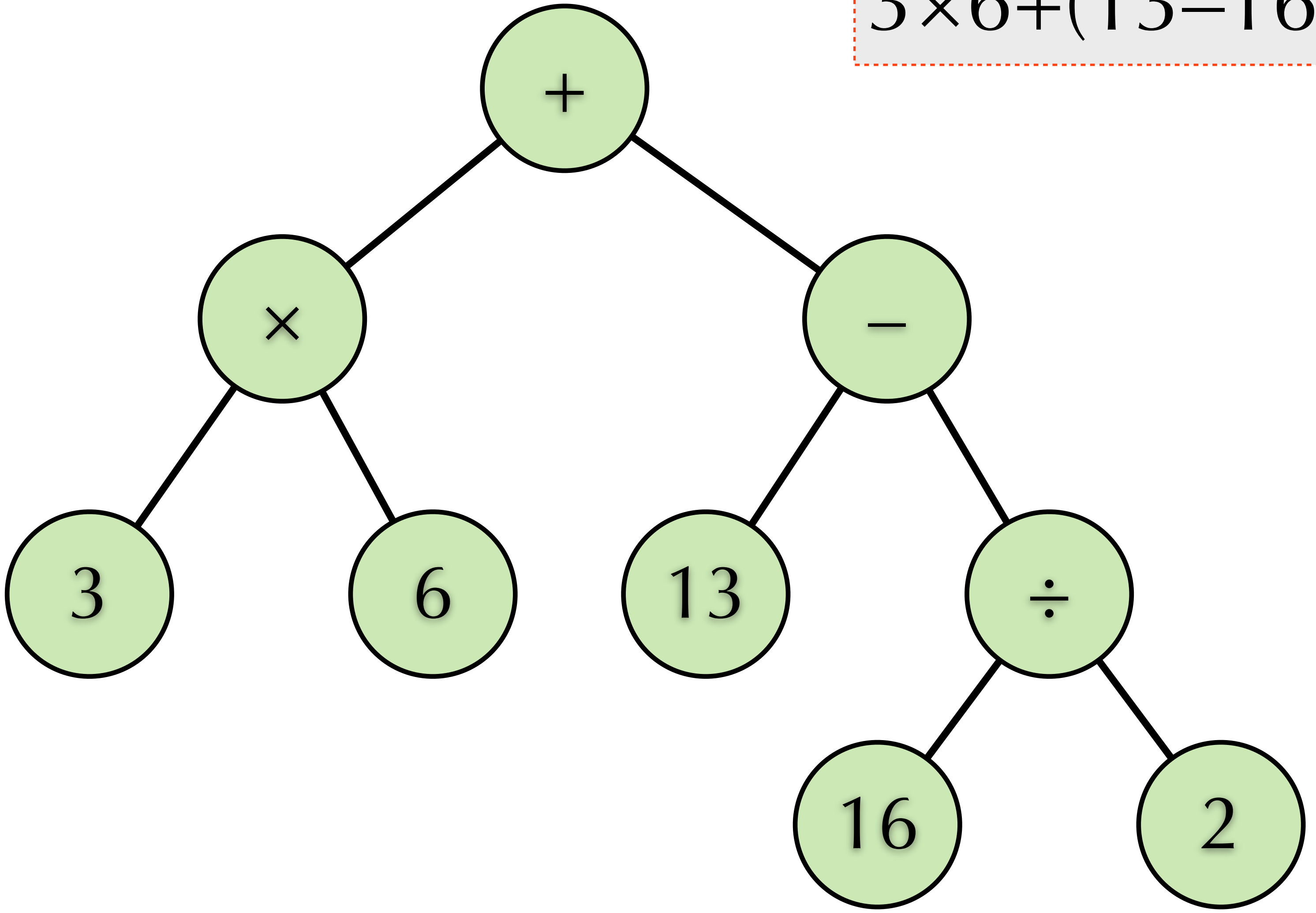
Example



3×6+(13–16÷2)

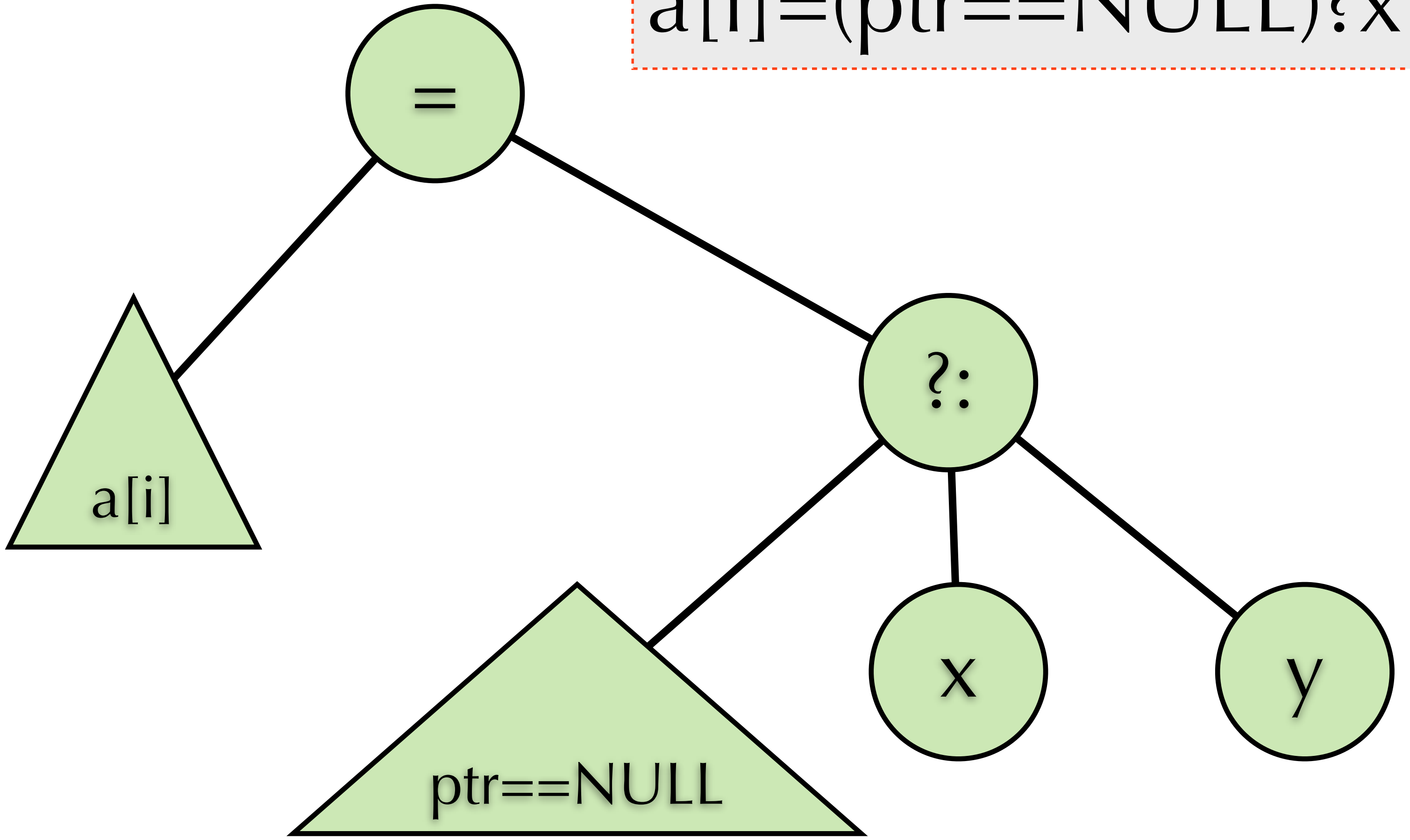
Example

3×6+(13–16÷2)



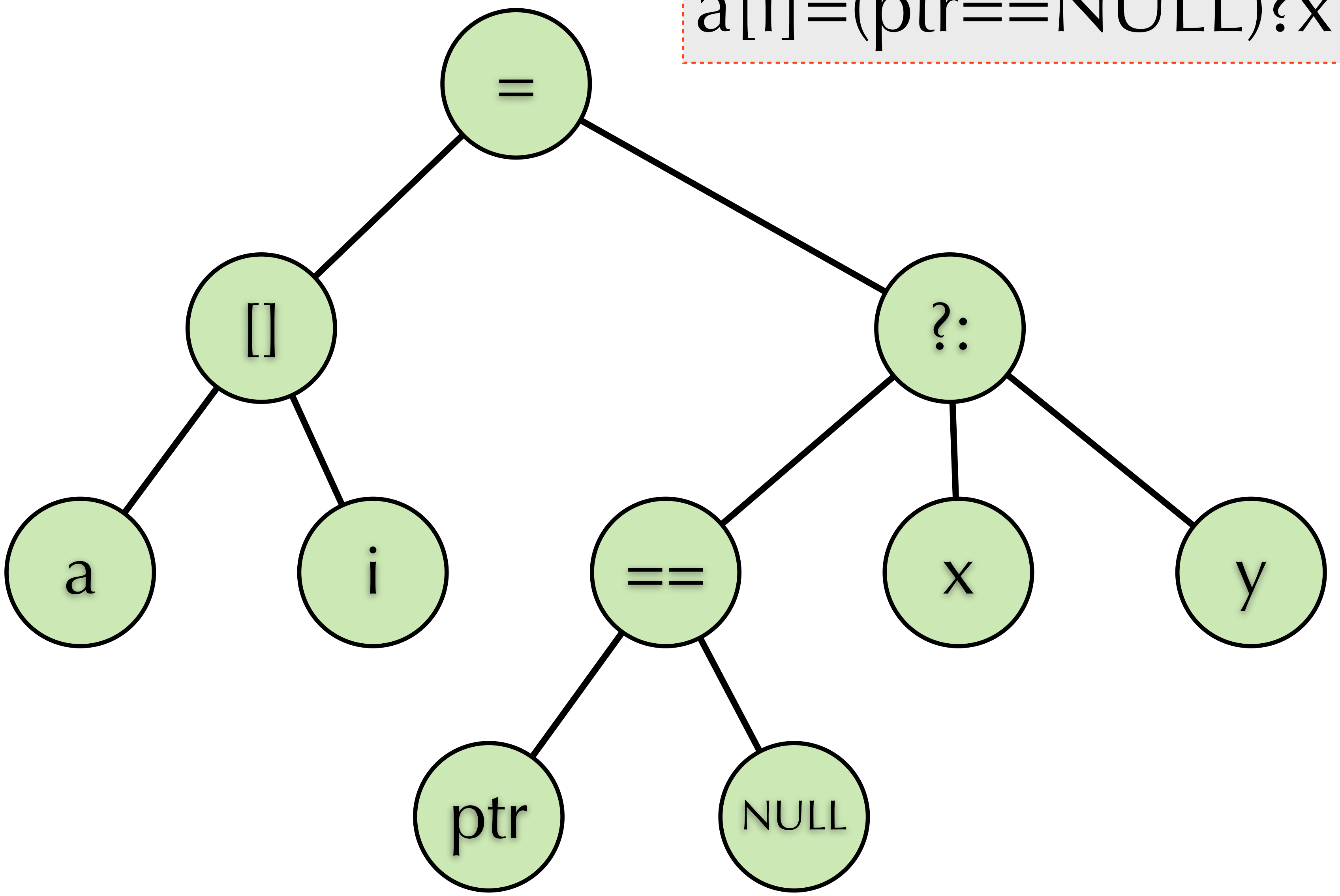
Example

a[i]=(ptr==NULL)?x:y



Example

a[i]=(ptr==NULL)?x:y



Homework 4.1

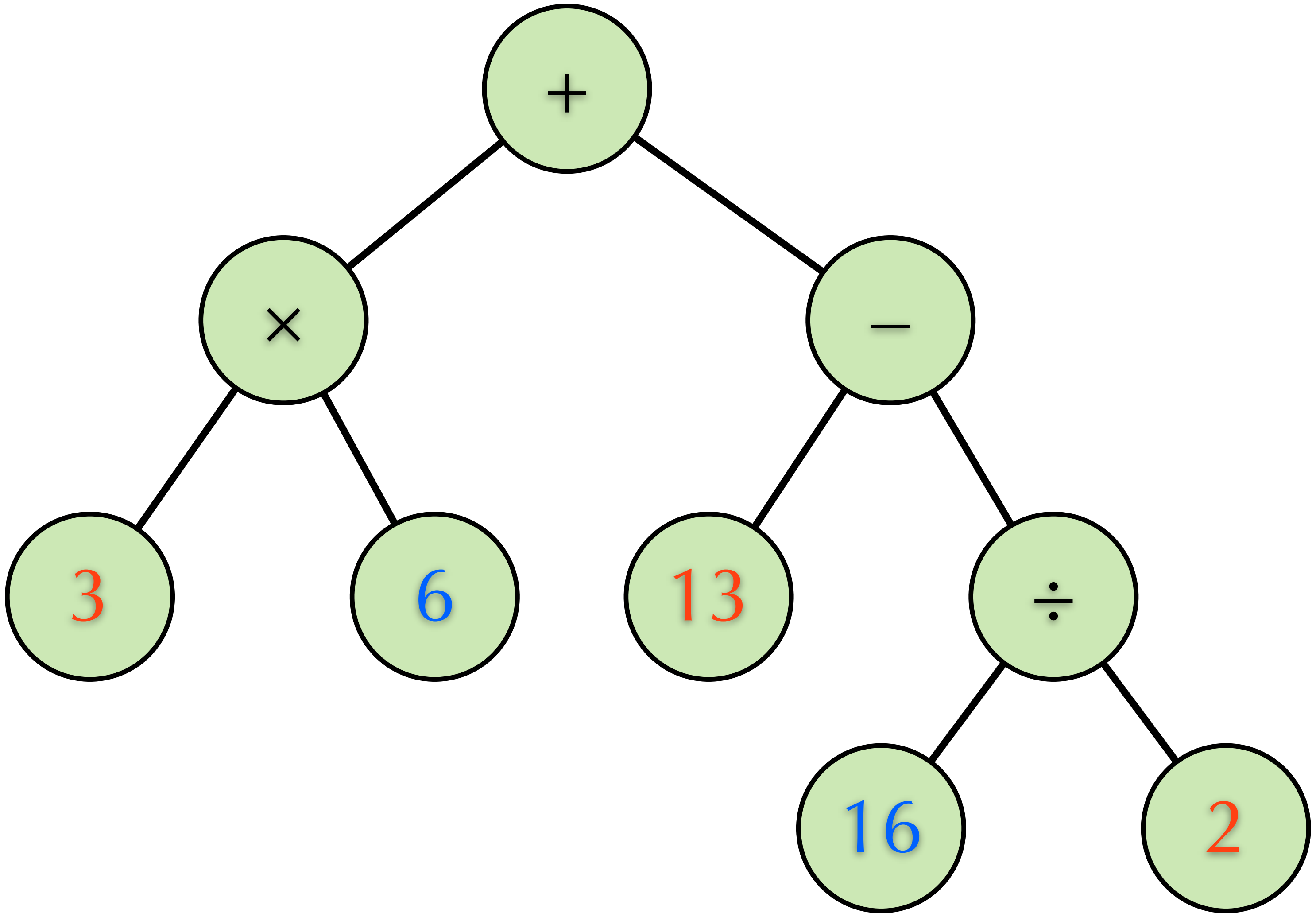
- a) Define a structure for expression tree.
- b) Implement a C program to construct a tree from an expression.
- c) Implement a C program to evaluate an expression tree.
- d) What is the time complexity of the recursive algorithm building the expression tree?

Bonus

- Write a calculator (5pts)
  - Support variables
    - ex: define int x
    - ex: undef x
  - Can evaluate C expressions
  - Support print
    - ex: print x
- Demo is required

Postfix expression

- Postfix expression is generated by the post-order traversal of an expression tree.



Infix

3×6+(13–16÷2)

Postfix

36×13162÷–+

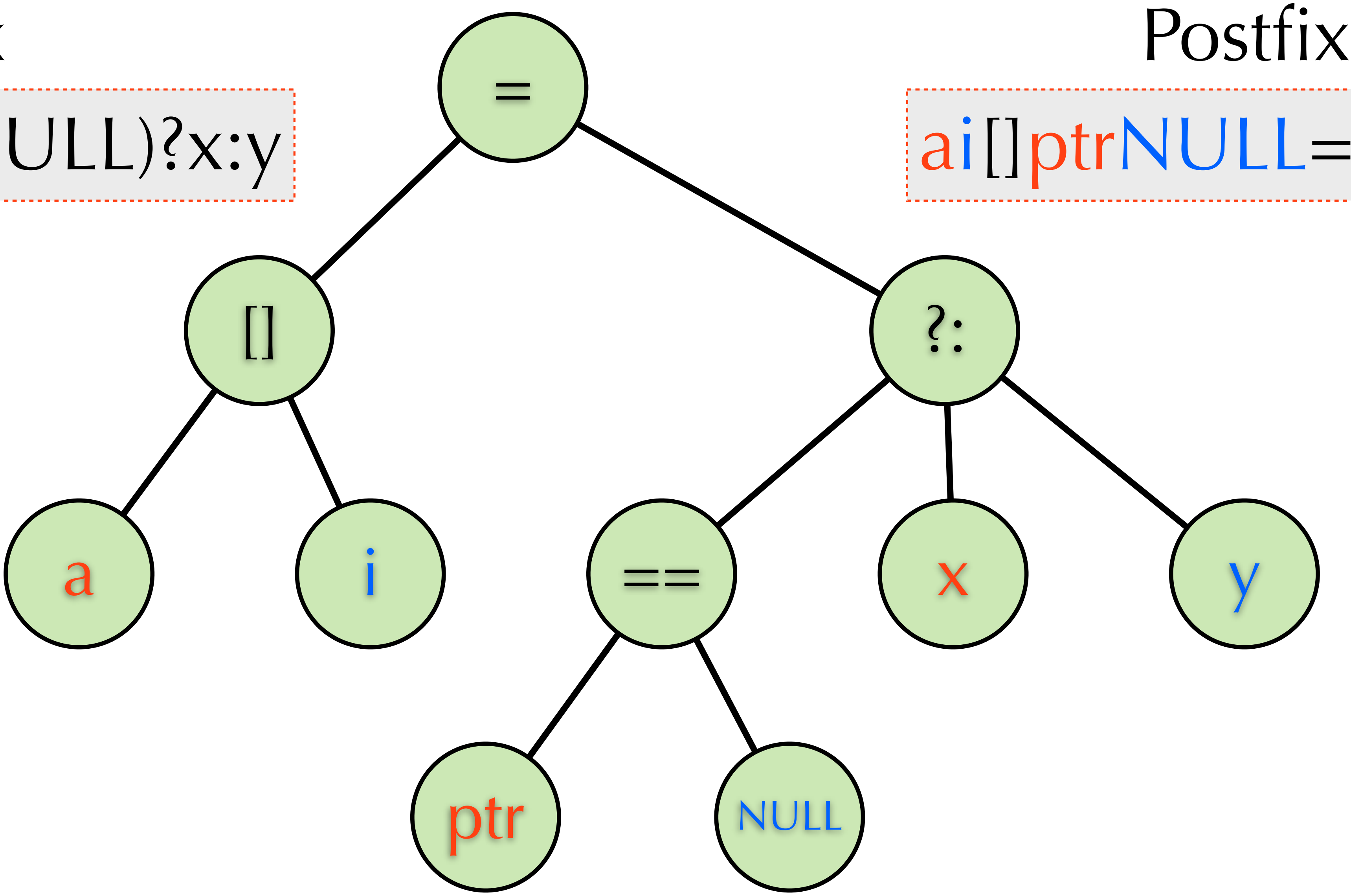
Example

Infix

a[i]=(ptr==NULL)?x:y

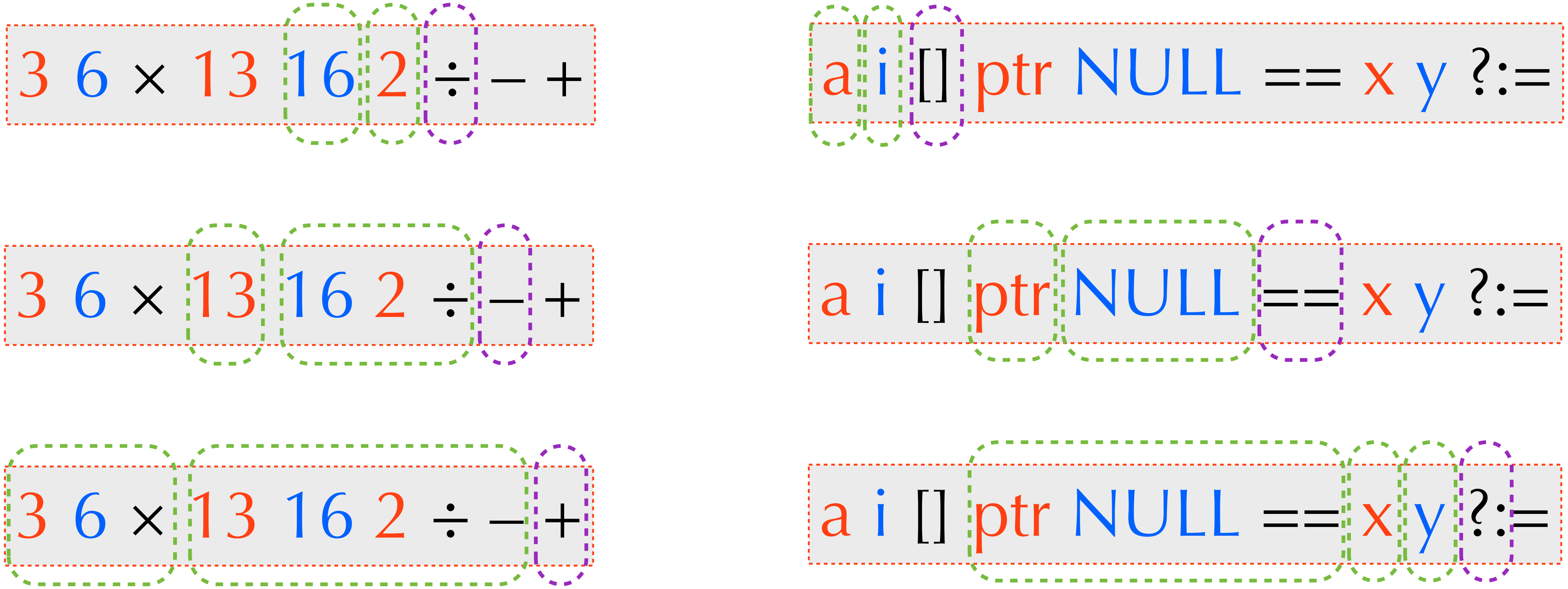
Postfix

ai[]ptrNULL==xy?:=



Evaluation

- Observation: Operands are right before their operator.



3 6 × 13 16 2 ÷ − +

a i [] ptr NULL == x y ?:=

3 6 × 13 16 2 ÷ − +

a i [] ptr NULL == x y ?:=

3 6 × 13 16 2 ÷ − +

a i [] ptr NULL == x y ?:=

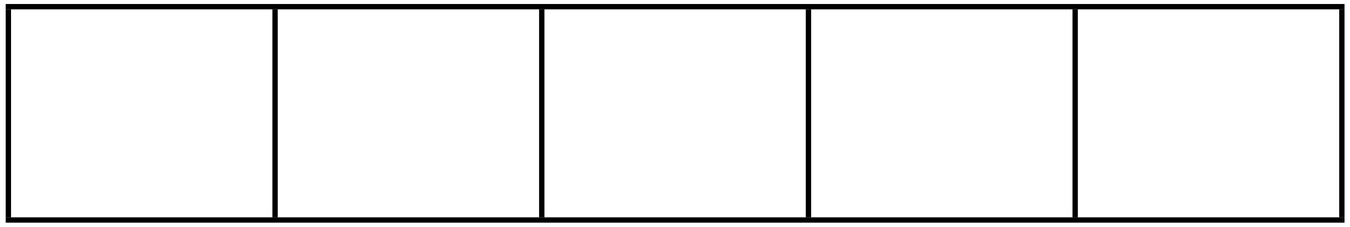
Using Stack

- Reading symbols from left to right.
  - If the symbol is an operand, push it in to the stack.
  - If the symbol is an operator, then pop corresponding number of operands from the stack. Evaluate the result of the operation, then push the result back to the stack.

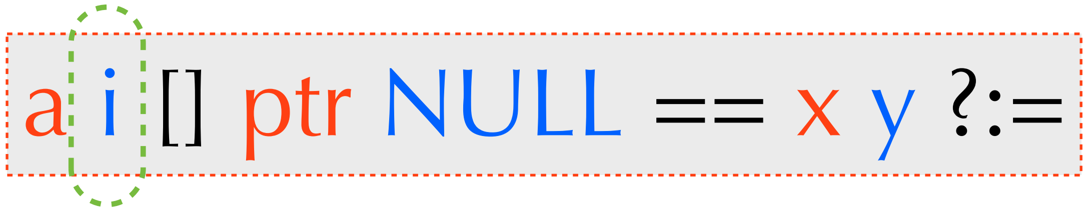
Example

a i [] ptr NULL == x y ?:=

Push a



Example



a i [] ptr NULL == x y ?:=

Push i

a				

Example

a i [] ptr NULL == x y ?:=

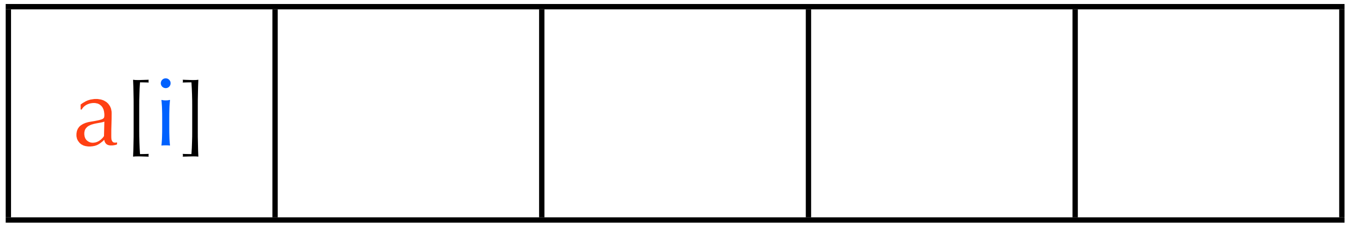
v_1=Pop(); v_2=Pop(); Push $v_2[v_1]$

a	i			

Example

a i [] ptr NULL == x y ?:=

Push ptr



Example

a i [] ptr NULL == x y ?:=

Push NULL

a[i]	ptr			

Example

a i [] ptr NULL == x y ?:=

v_1=Pop(); v_2=Pop(); Push v_2==v_1

a[i]	ptr	NULL		

Example

a i [] ptr NULL == x y ?:=

Push x

a[i]	1			

Example

a i [] ptr NULL == x y ?:=

Push y

a[i]	1	x		

Example

a i [] ptr NULL == x y ?: =

v_1=Pop(); v_2=Pop(); v_3=Pop(); Push $v_3?v_2:v_1$

a[i]	1	x	y	

Example

a i [] ptr NULL == x y ?:=

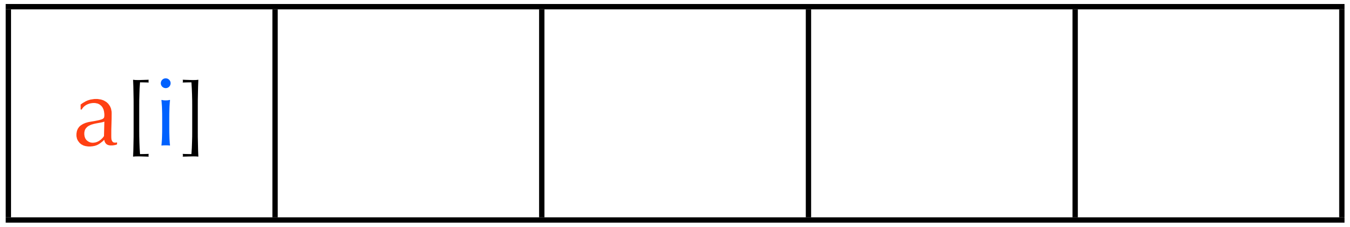
v_1=Pop(); v_2=Pop(); Push v_2=v_1

a[i]	x			

Example

a i [] ptr NULL == x y ?:=

Result=Pop()



a[i]

Note: a[i] stores x now.

Conversion

- Evaluating an n-symbol postfix expression takes O(n) time.
- Why postfix?
  - It should be faster than evaluating the expression by expression tree.
- The rest problem is:
  - How to convert an infix expression into a postfix expression?
  - Is it fast enough?

Conversion

- Observation: a symbol x is either operated by the operator on its left hand side or on its right hand side!
  - Which should be done first? That is the question!
- Parenthesis: Expression between a pair of parentheses should be evaluated before the outer expression.

Conversion

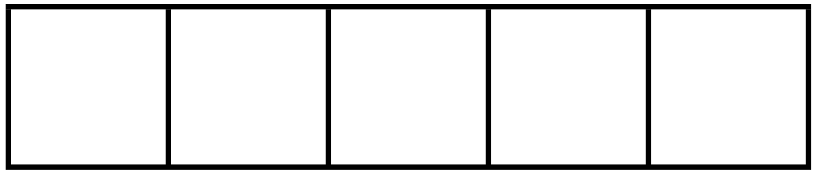
- Strategy: Process symbols one-by-one.
  - Operand: Output it directly.
  - Left parenthesis (: Push it into the stack.
  - Right parenthesis): Repeat popping operators until popping a left parenthesis (.
  - Operator σ: Pop all operators should be executed before σ, then push it into the stack.
  - End of input: Repeat popping operators the stack is empty.
  - Note: All operator should be output when it is popped.

Example

3 × 6 + (13 – 16 ÷ 2)

Output

Output 3



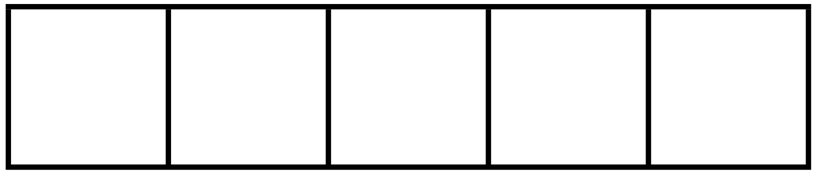
Example

3 × 6 + (13 – 16 ÷ 2)

3

Output

The stack is empty.
Push ×



Example

3 × 6 + (13 – 16 ÷ 2)

3

Output

Output 6

×				

Example

3 × 6 + (13 – 16 ÷ 2)

3 6

Output

Check if + is before ×…

No! Pop()! Output ×

×				

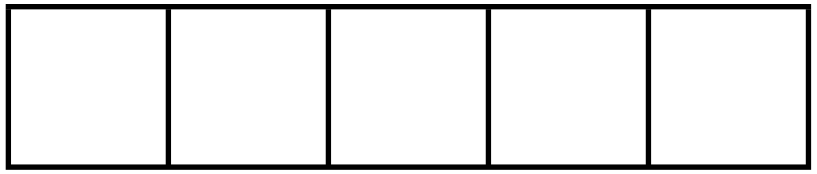
Example

3 × 6 + (13 – 16 ÷ 2)

3 6 ×

Output

The stack is empty.
Push +



Example

3 × 6 + (13 – 16 ÷ 2)

3 6 ×

Output

Push (

+				

Example

3 × 6 + (13 – 16 ÷ 2)

Output 13

+	(			

3 6 ×

Output

Example

3 × 6 + (13 – 16 ÷ 2)

3 6 × 13

Output

Check if – is before (…

Yes! Push –

+	(			

Example

3 × 6 + (13 − 16 ÷ 2)

3 6 × 13

Output

Output 16

+	(	−		

Example

3 × 6 + (13 – 16 ÷ 2)

3 6 × 13 16

Output

Check if ÷ is before –…

Yes! Push ÷

+	(	–		

Example

3 × 6 + (13 – 16 ÷ 2)

3 6 × 13 16

Output

Output 2

+	(	–	÷	

Example

3 × 6 + (13 – 16 ÷ 2)

3 6 × 13 16 2

Output

Repeat popping until (

+	(	–	÷	

Example

3 × 6 + (13 – 16 ÷ 2)

End of input!
Repeat popping
until stack empty.

+				

3 6 × 13 16 2 ÷ –

Output

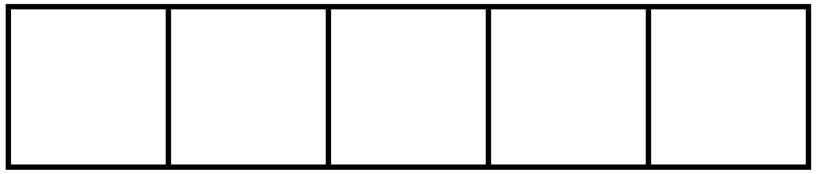
Example

3 × 6 + (13 – 16 ÷ 2)

3 6 × 13 16 2 ÷ – +

Output

Done!



Homework 4.2

- a) Submit some infix expressions for your midterm.
- b) The conversion algorithm works well when the operators are all binary. If we allow some unary operators (such as ++ and !) and ternary operators (such as ?:), then how should you modify the algorithm?