Data Structures and Algorithms

Lecture 4 – Stacks

Miao Zhang

What Is A Stack

- **Intuitively, a stack** is like a pile of plates where we can only remove a plate from the top and can only add a new plate on the top.
- In computer science we commonly place numbers on a stack, or perhaps place records on the stack.

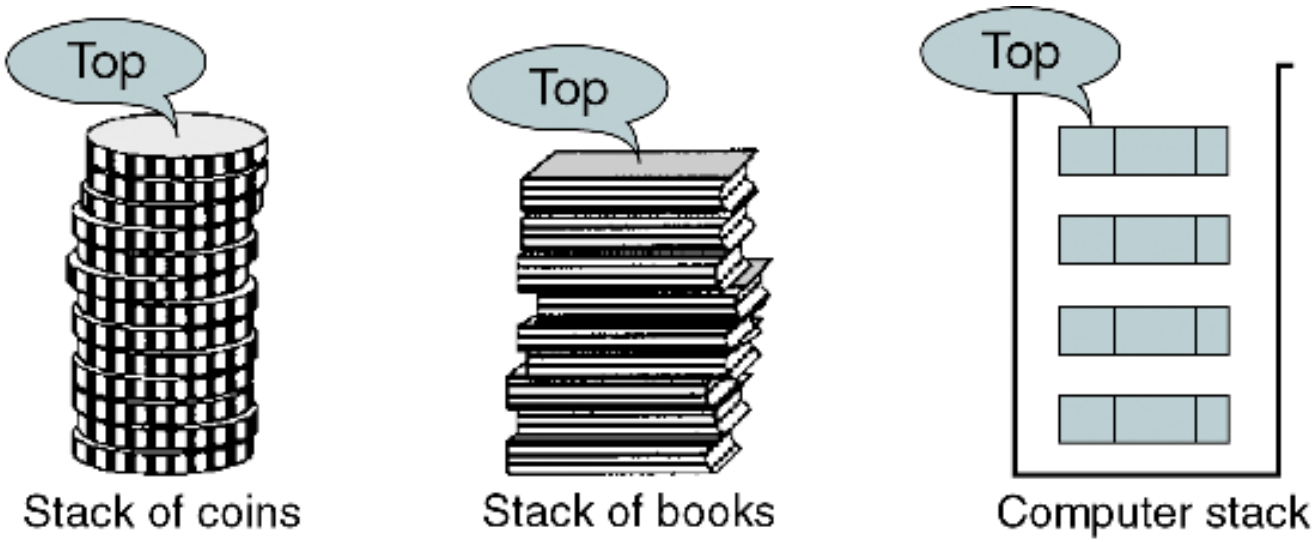


FIGURE 3-1 Stack

Stacks

- A stack is called a "last in, first out" (LIFO) data structure since the last item pushed onto a stack will be the first one popped off.
- It can also be called a "first in, last out" (FILO) data structure since the first item pushed onto it will be the last one popped off.
- Because of these features, a stack is the ideal data structure to use in reversing a sequence of items.

Notation:

- Insert: PUSH
- Remove: POP
- The accessible element is called TOP.

Example of Reversing A sequence of Items

```
cout << "Enter a number (CTRL z to end): ";
cin >> Number;

while (! cin.fail())
    {
    Stack.Push(Number);
    cout << "Enter another number (CTRL z to end): ";
    cin >> Number;
    }

cout << endl << endl << "Numbers in reverse order are:" << endl;
while (! Stack.Empty())
    {
    Stack.Pop(Item);
    cout << Item << endl;
    }
```

Stacks

The Push operation places a new data item on top of the stack

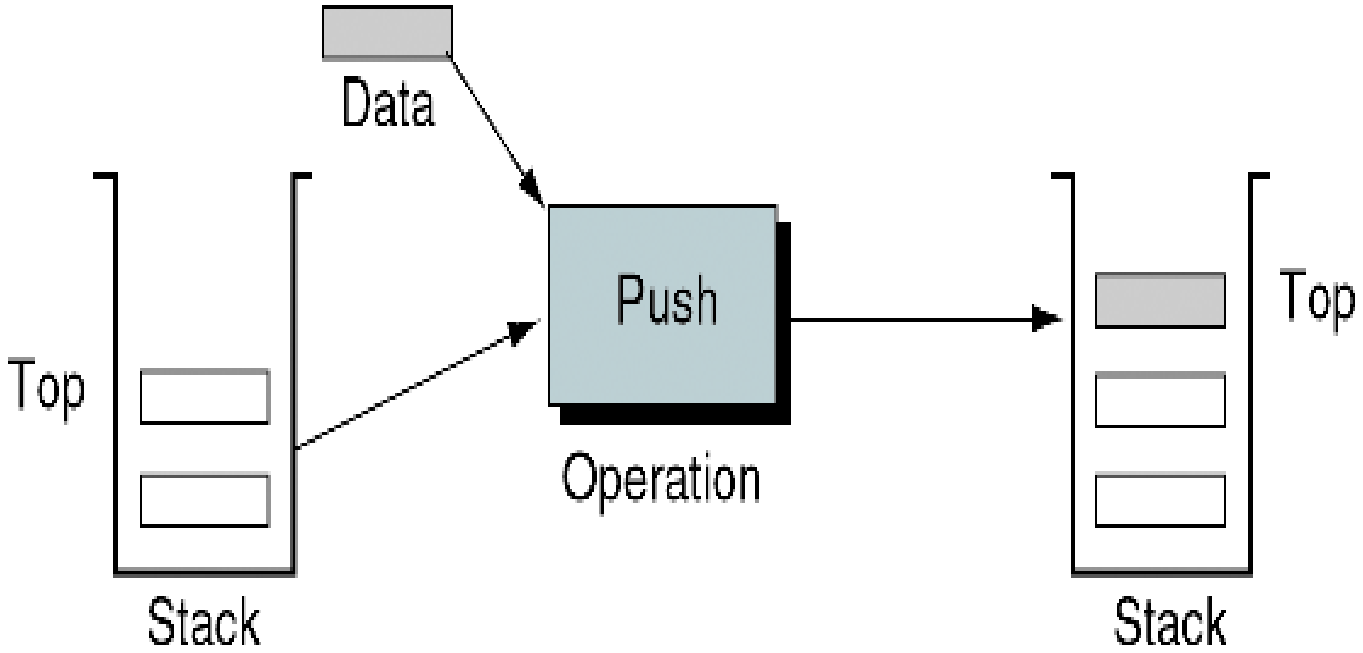


FIGURE 3-2 Push Stack Operation

Stacks

Pop removes the top item

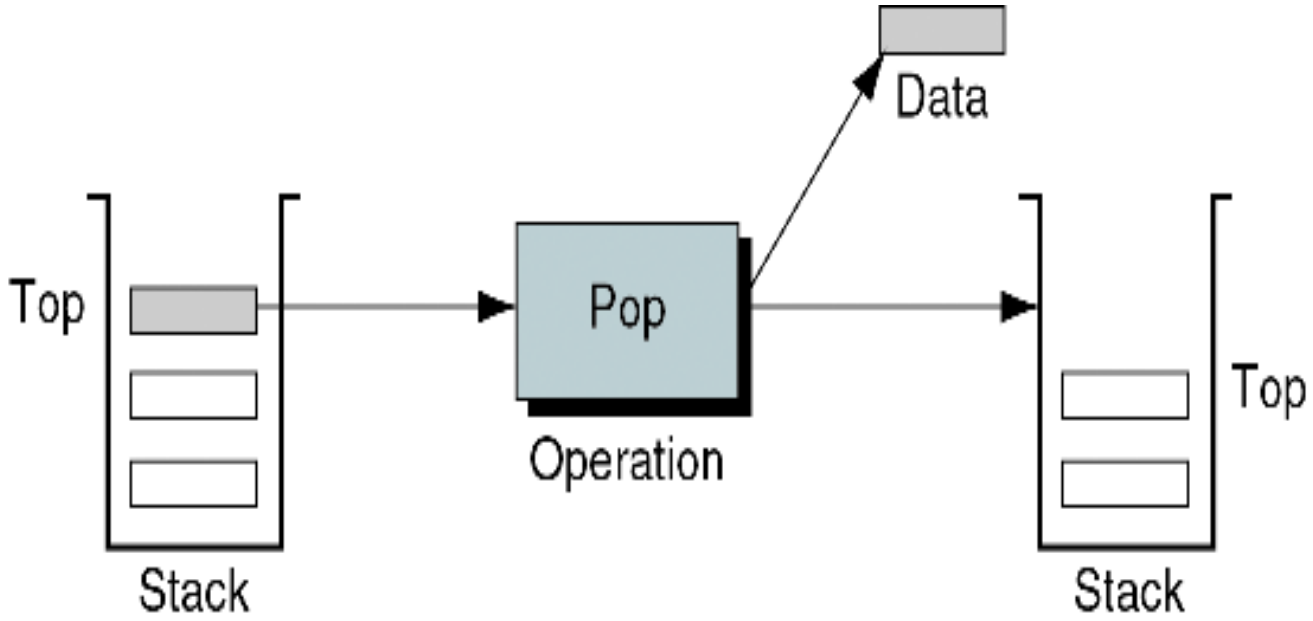


FIGURE 3-3 Pop Stack Operation

Stacks

Top of stack observes the top-most element without removing it from the stack

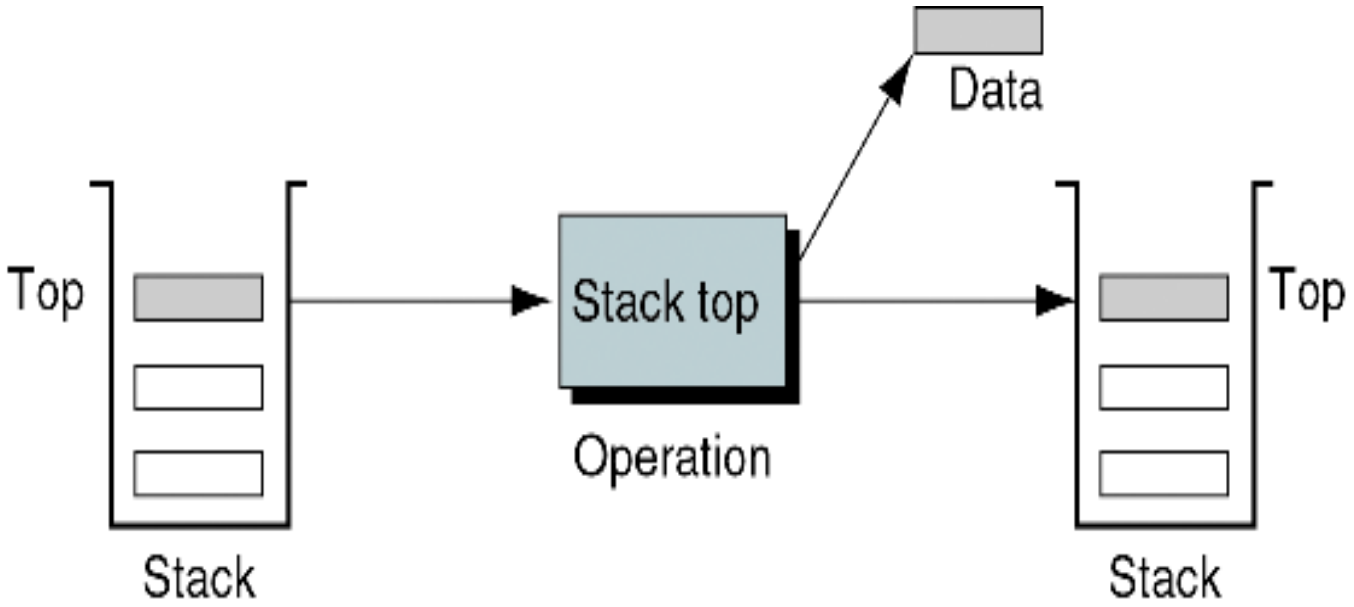
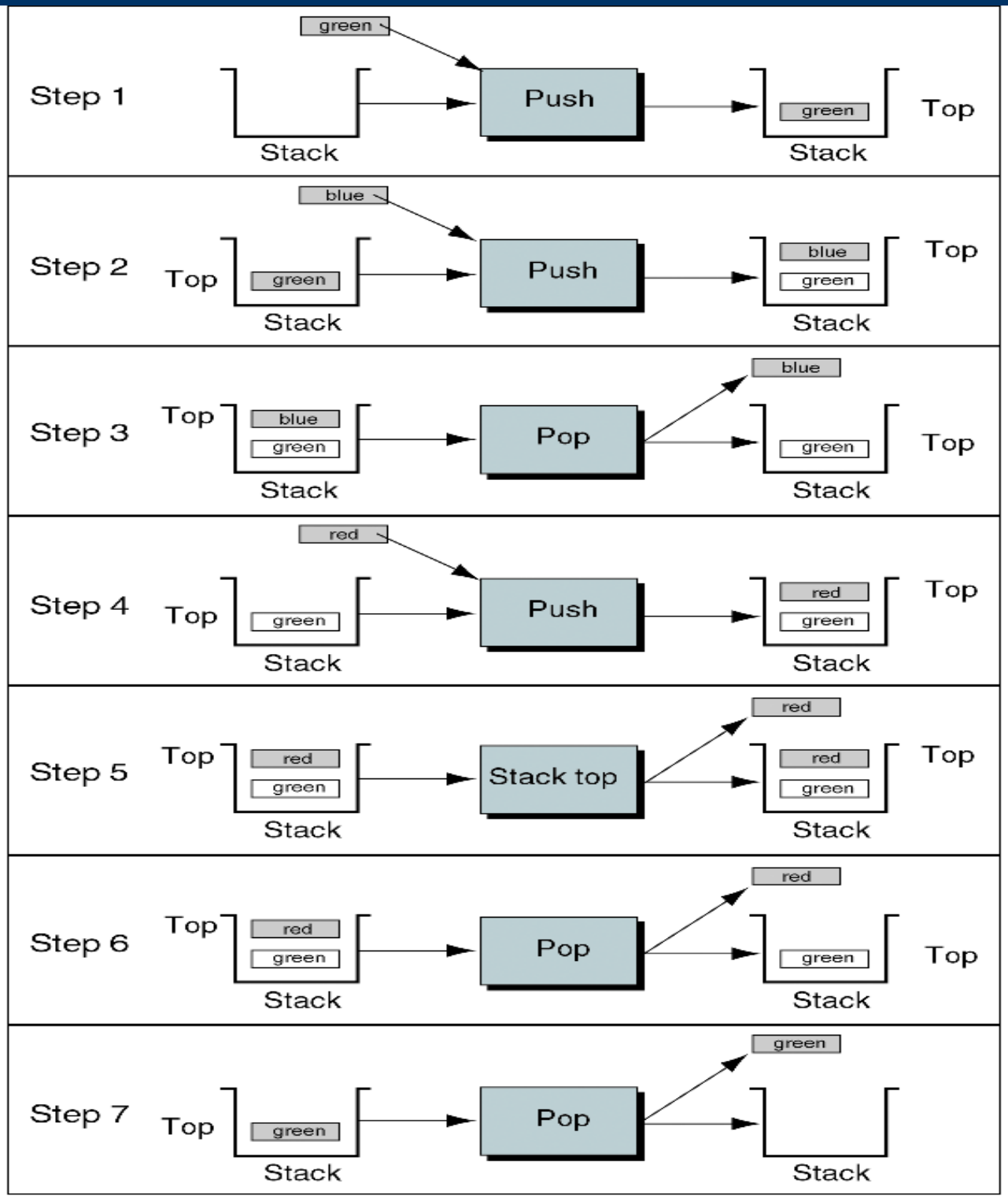


FIGURE 3-4 Stack Top Operation


Step 1
green
Push
Stack
green
Top
Stack
Step 2
blue
Top
green
Push
Stack
blue
green
Top
Stack
Step 3
Top
blue
green
Pop
Stack
blue
green
Top
Stack
Step 4
red
Top
green
Push
Stack
red
green
Top
Stack
Step 5
red
Top
red
green
Stack top
Stack
red
red
green
Top
Stack
Step 6
red
Top
red
green
Pop
Stack
red
green
Top
Stack
Step 7
green
Top
green
Pop
Stack
green
Stack

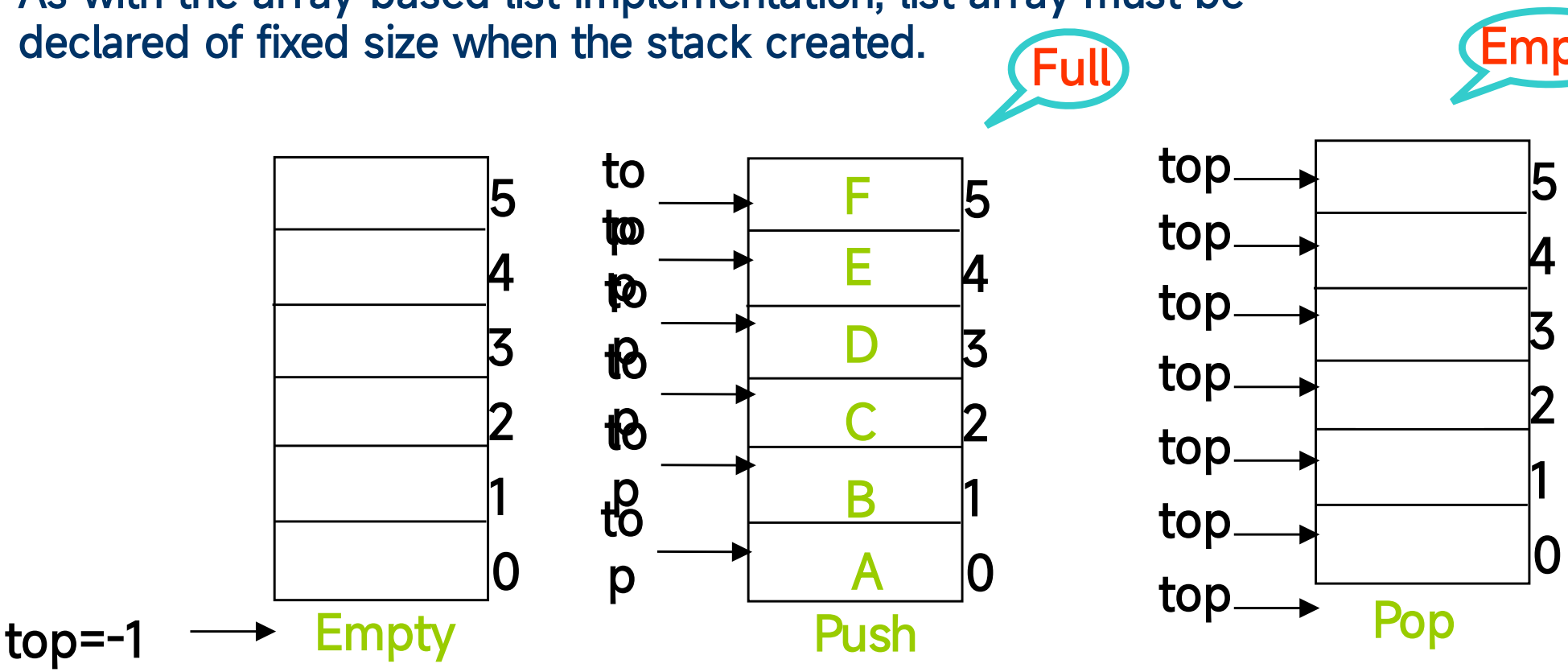
Basic Operations for Stacks

- Clear()——clear contents from the stack, make it empty
- IsEmpty()——determine if the stack is empty
- Push(T item)——push an item onto the top of the stack
- Pop(T& item)——remove the item at the top of the stack
- Top (T& item)——gain access the top item。
- IsFull()——determine if the stack is full

Array-Based Stacks

Array-Based Stacks

As with the array-based list implementation, list array must be declared of fixed size when the stack created.

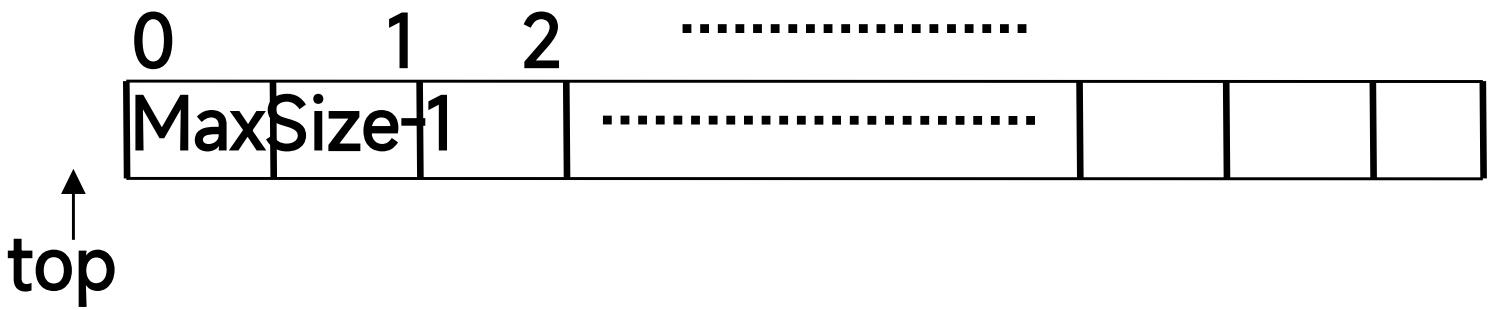


Top is defined to the array index of the first free position in the stack.

An empty stack as top set to -1

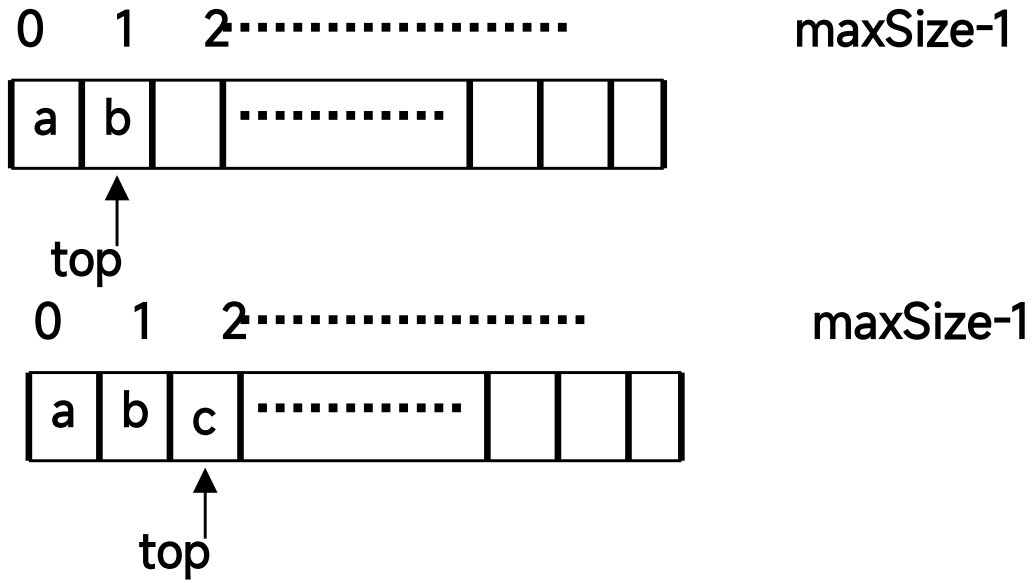
Array-Based Stacks

```
template<class T>
ArrayStack<T>::ArrayStack(int MaxStackSize){
   MaxSize=MaxStackSize-1;
   st=new T[MaxStackSize];
   top=-1;
}
```



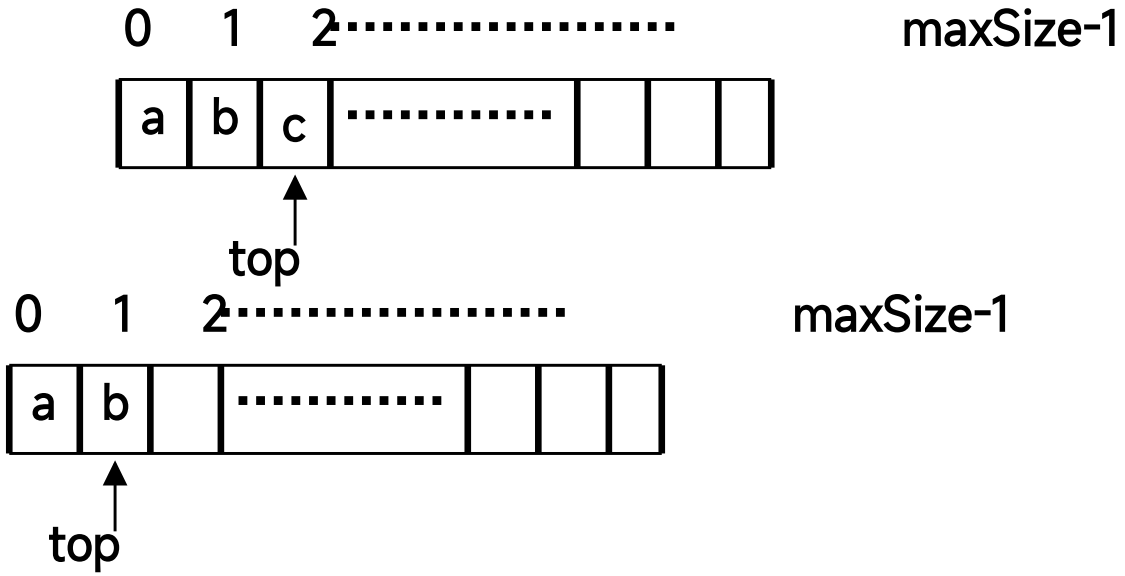
Push

```
template<class T>
bool ArrayStack<T>::Push(const T item ){
   if(IsFull())
   {cout<<"no memory;"<<endl;return false;}
   top=top+1;
   st [top]=item;
   return true;
}
```



Pop

```
template<class T>
bool ArrayStack<T>::Pop(T& item){
   if(IsEmpty())
   {cout<<"no element"<<endl;return false;}
   item=st[top];
   top=top-1;
   return true;
}
```



Two stacks in an array

- When multiple stacks are to be implemented, it is possible to take advantage of one-way growth of the array-based stack.
- This can be done by using a single array to store two stacks.
- Two stacks grow inward to each end.
- This works well when the space requirements of the two stacks are inversely correlated

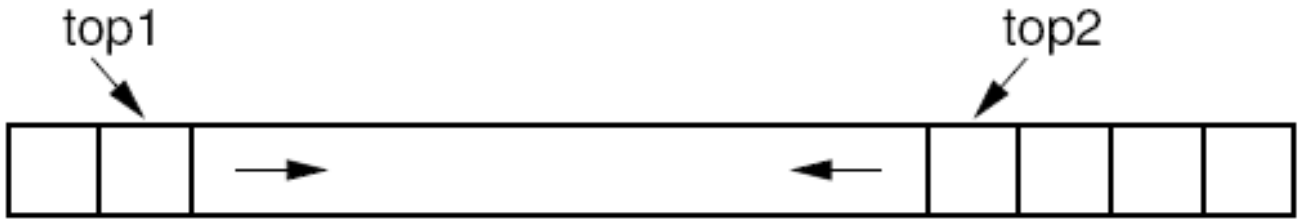


Figure 4.20 Two stacks implemented within in a single array, both growing toward the middle.

Linked Stacks

Linked Stacks

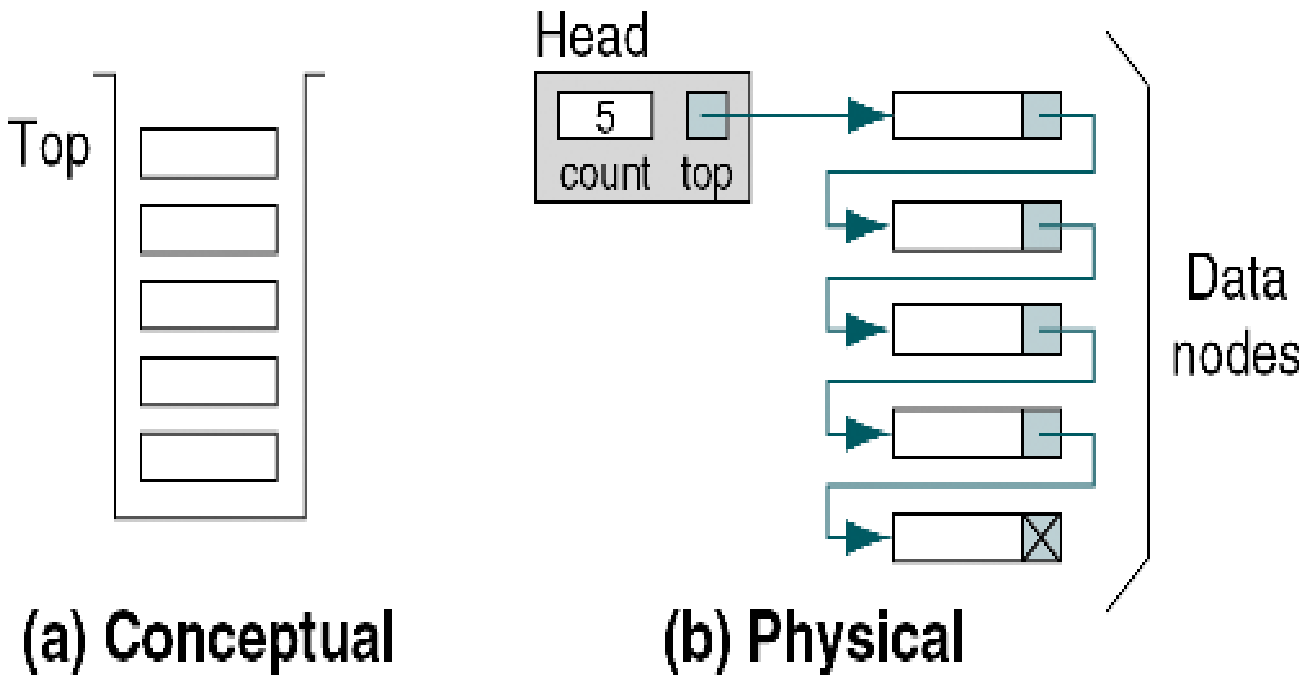
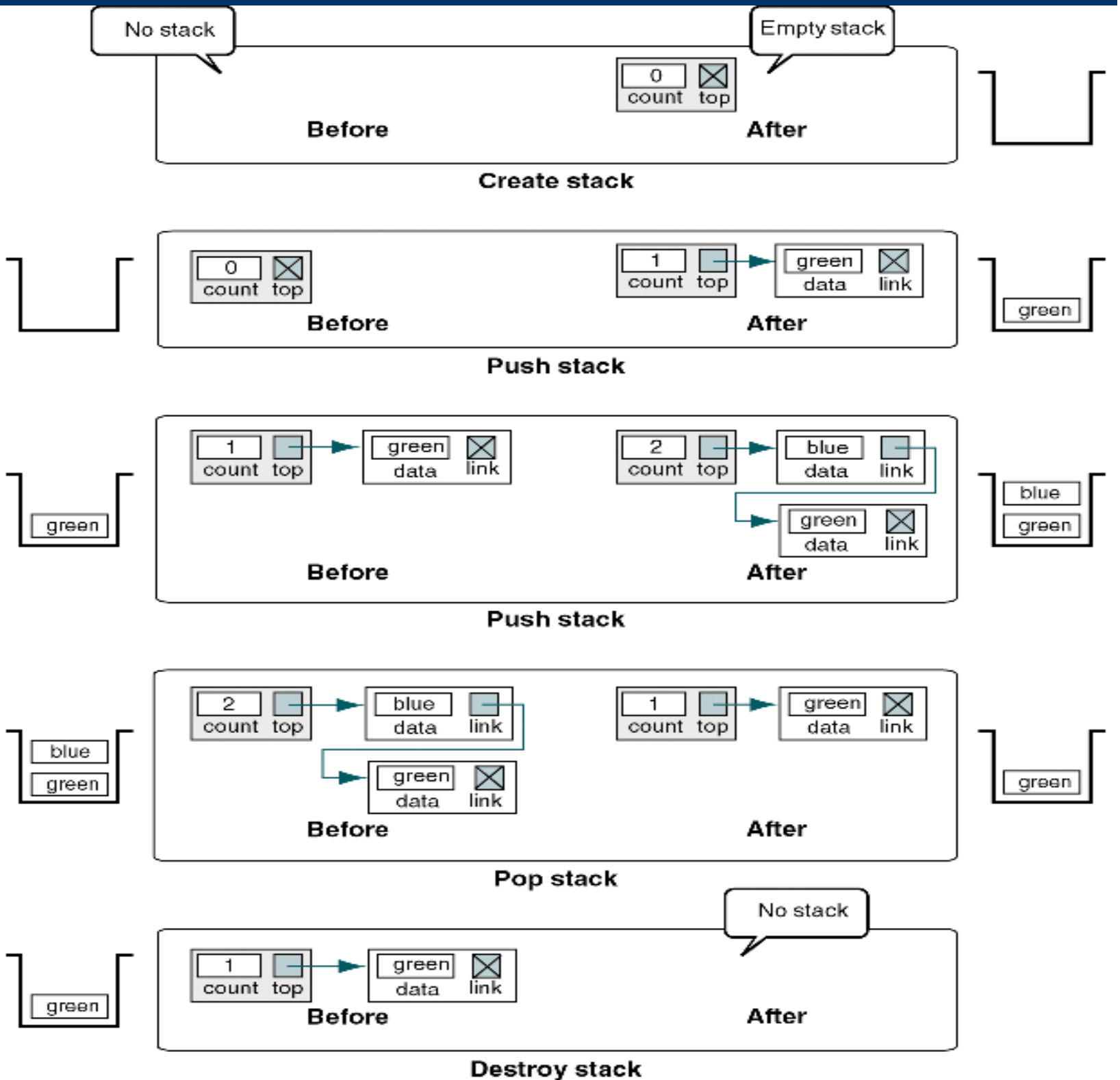


FIGURE 3-6 Conceptual and Physical Stack Implementations


No stack
Empty stack
0
count
top
Before
After
Create stack
0
count
top
1
count
top
green
data
link
Before
After
green
Push stack
1
count
top
green
data
link
2
count
top
blue
data
link
green
data
link
Before
After
green
blue
green
Push stack
2
count
top
blue
data
link
green
data
link
1
count
top
green
data
link
Before
After
blue
green
green
Pop stack
No stack
1
count
top
green
data
link
Before
After
green
Destroy stack

Linked stacks

```
// Linked stack implementation
template <typename E> class LStack: public Stack<E> {
private:
  Link<E>* top;                  // Pointer to first element
  int size;                      // Number of elements

public:
  LStack(int sz =defaultSize) // Constructor
    { top = NULL; size = 0; }

  ~LStack() { clear(); }              // Destructor

  void clear() {                      // Reinitialize
    while (top != NULL) {             // Delete link nodes
      Link<E>* temp = top;
      top = top->next;
      delete temp;
    }
    size = 0;
  }
```

Push & Pop Operations

```
void push(const E& it) { // Put "it" on stack
  top = new Link<E>(it, top);
  size++;
}

E pop() {                          // Remove "it" from stack
  Assert(top != NULL, "Stack is empty");
  E it = top->element;
  Link<E>* ltemp = top->next;
  delete top;
  top = ltemp;
  size--;
  return it;
}
```

Return top value

```
  const E& topValue() const { // Return top value
    Assert(top != 0, "Stack is empty");
    return top->element;
  }

  int length() const { return size; } // Return length
};
```

Comparison of Array-Based and Linked Stacks

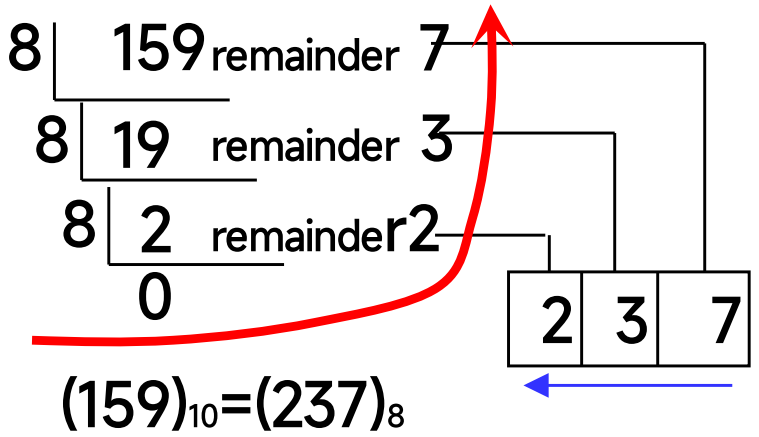
- The array-based stack must declare a fixed-size array initially, and some of that space is wasted whenever the stack is not full.
- The linked stack can shrink and grow but requires the overhead of a link field for every element.
- When multiple stacks are to be implemented, it is possible to take advantage of the one-way growth of the array-based stack.
  - this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink.

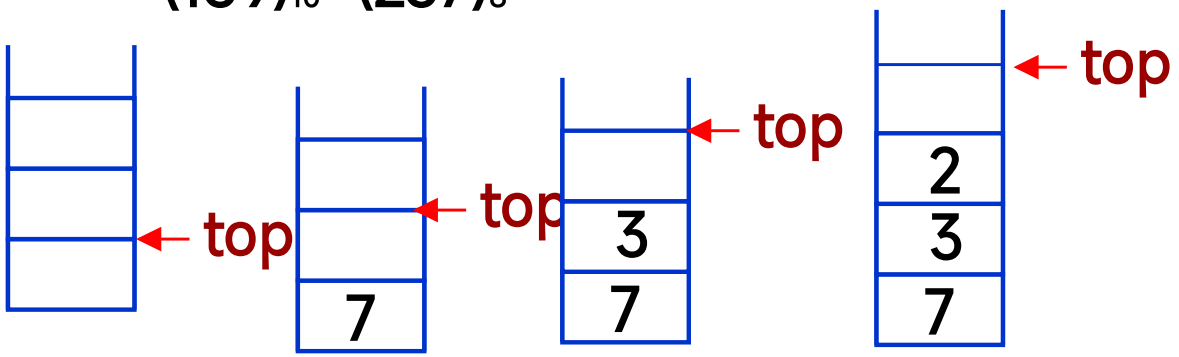
STACK APPLICATION

HOW COMPILERS IMPLEMENT RECURSION

Number System Conversion

➤Example: convert a decimal to a octal number 。





Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Position	1	2	3	4	5	6	7	8	9	10	11

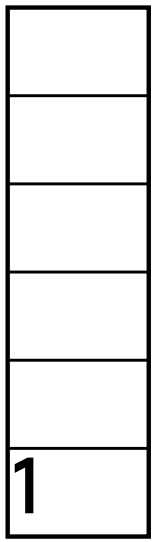
Algorithms：From left to right.

(1)If it is left bracket, push its position number in stack.

(2)If it is right bracket, match to the top of stack.

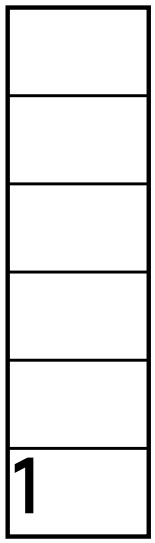
Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Position	1	2	3	4	5	6	7	8	9	10	11



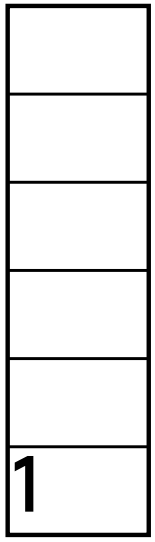
Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions 10 11	1	2	3	4	5	6	7	8	9		



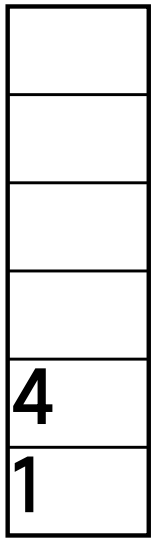
Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions 10 11	1	2	3	4	5	6	7	8	9		



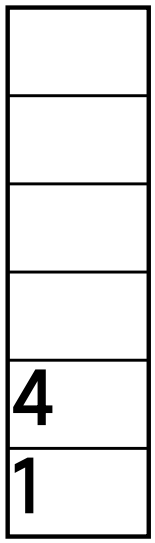
Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions	1	2	3	4	5	6	7	8	9		
10 11											



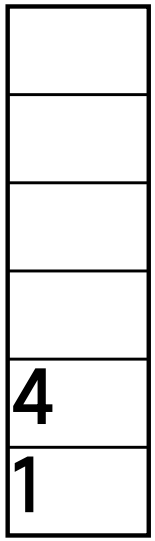
Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions 10 11	1	2	3	4	5	6	7	8	9		



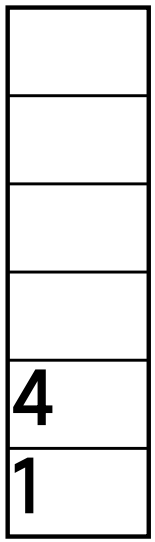
Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions	1	2	3	4	5	6	7	8	9		
10 11											

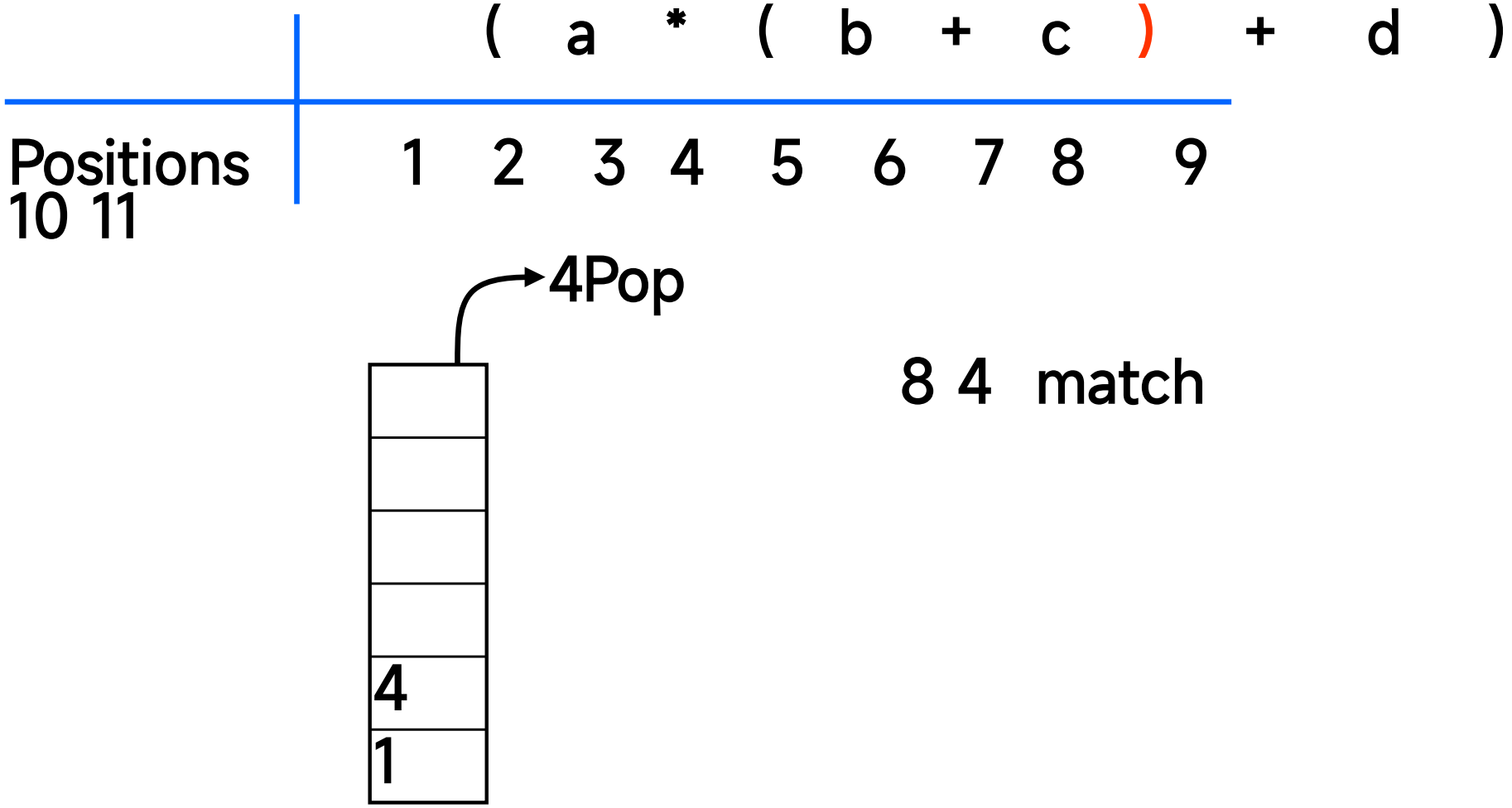


Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions 10 11	1	2	3	4	5	6	7	8	9		

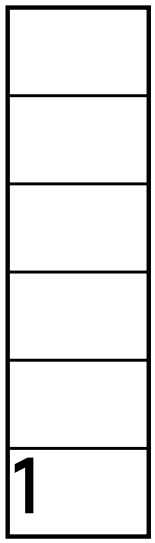


Brackets Matching



Brackets Matching

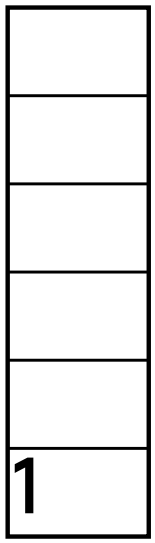
	(	a	*	(	b	+	c	)	+	d	)
Positions 10 11	1	2	3	4	5	6	7	8	9		



8 4 match

Brackets Matching

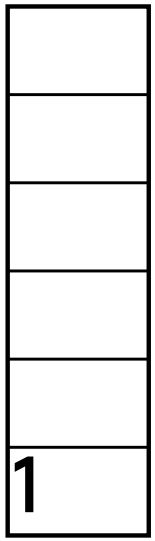
	(	a	*	(	b	+	c	)	+	d	)
Postions 10 11	1	2	3	4	5	6	7	8	9		



8 4 match

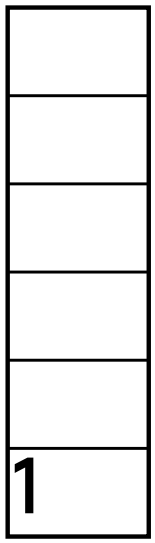
Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions	1	2	3	4	5	6	7	8		9	
10 11											



8 4 match

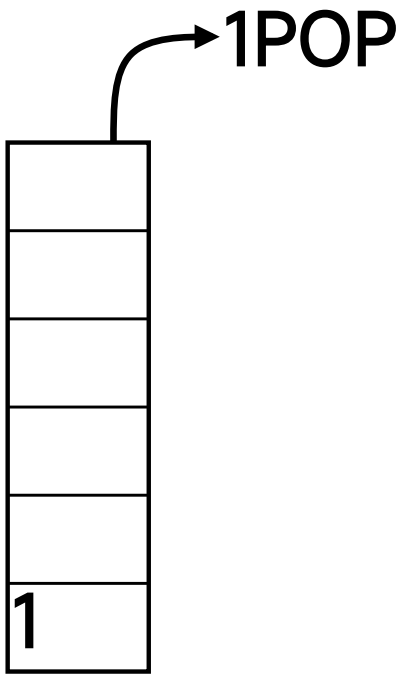
	(	a	*	(	b	+	c	)	+	d	)
Positions 10 11	1	2	3	4	5	6	7	8	9		



8 4 match

Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions	1	2	3	4	5	6	7	8	9	10	11

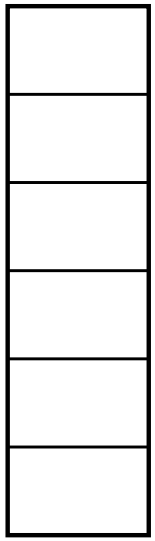


8 4 match

11 1 match

Brackets Matching

	(	a	*	(	b	+	c	)	+	d	)
Positions	1	2	3	4	5	6	7	8	9		
10 11											



8 4 match

11 1 match

Palindrome Game

Palindromes: mom, dad, bob, anna, hannah.....

Algorithm：

- Read a string
- Push into a stack
- Compare the original string with the poped-out string
- if not matched, non-palindrome
- if matched, palindromes。

Evaluate Expression

- Expression consists of operand(操作数), operator（操作符.

- Three ways of expression：

(1) infix (中缀)：

A+B

(2) prefix(前缀)：

+ AB

(3) postfix(后缀)：

AB +

Evaluate Expression

- Expression consists of operand(操作数), operator（操作符.

- Three ways of expression：

(1) infix (中缀)：

A+B

(2) prefix(前缀)：

+ AB

(3) postfix(后缀)：

AB +

infix

- EX.

  A+B*(C-D)-E/F

- Priority issues

- Two stacks. One is for operands, the other is for operators

infix

- EX.

  A+B*(C-D)-E/F

- Priority issues

- Two stacks. One is for operands, the other is for operators

infix

- Two stacks
  - OPERATORS (OPTR), the bottom of stack is #(表达式起始符#)
  - OPERAND (OPND), Empty stack

infix

- Suppost isp(.)represents the priority in the stack,
- icp(.)represents the priority out of the stack
- For each character,If it is
  - operand, push in the OPND;
  - Operator s1, compare with the s2 (the top of OPTR)
    - If icp(s1)>isp(s2), Push;
    - If icp(s1)<isp(s2), Pop two operands from stack (Y , X), do the calculation X<op>Y with the top in OPTR stack, push the calculation result in the stack, pop the top out of OPRT.
    - If icp(s1)==isp(s2), Pop the item of OPTR
- Until the end of expression。

infix

- Suppost isp(.)represents the priority in the stack,
- icp(.)represents the priority out of the stack
- For each character,If it is
  - operand, push in the OPND;
  - Operator s1, compare with the s2 (the top of OPTR)
    - If icp(s1)>isp(s2), Push;
    - If icp(s1)<isp(s2), Pop two operands from stack (Y , X), do the calculation X<op>Y with the top in OPTR stack, push the calculation result in the stack, pop the top out of OPRT.
    - If icp(s1)==isp(s2), Pop the item of OPTR
- Until the end of expression。

The priority of operators

Operators	#	(	^ *,/,%		+,-	)	
isp	01		7	5		3	8
icp	0		8	6	4		2

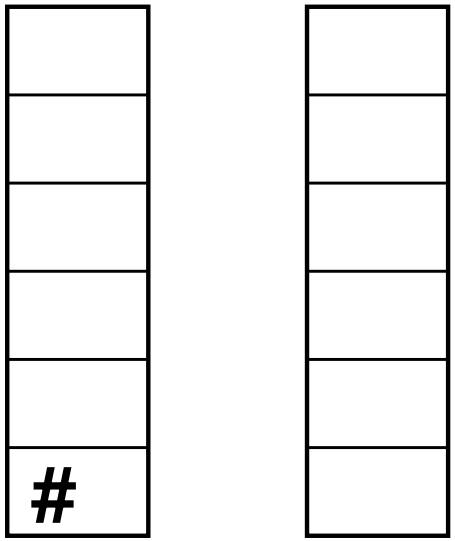
1

Infix

Ex： x = 3 × (7-2);

Process： # 3 × (7-2)

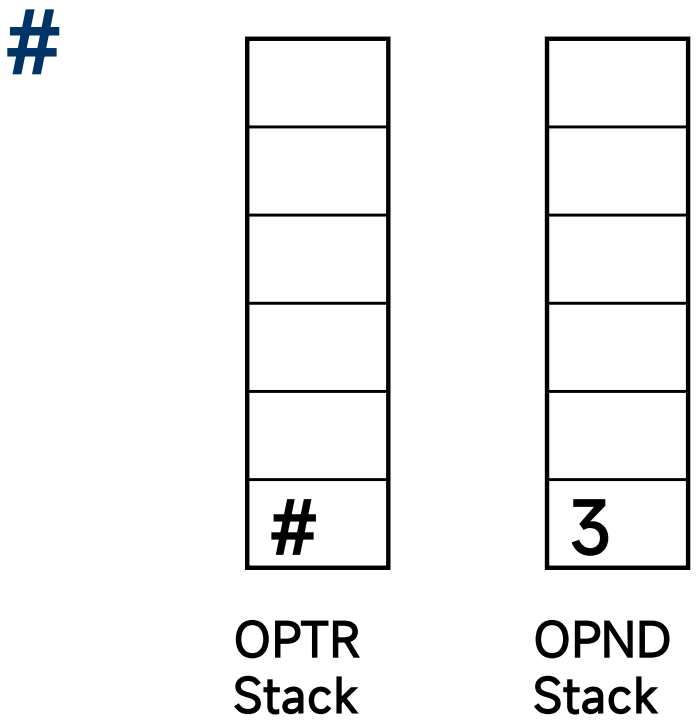
#



Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

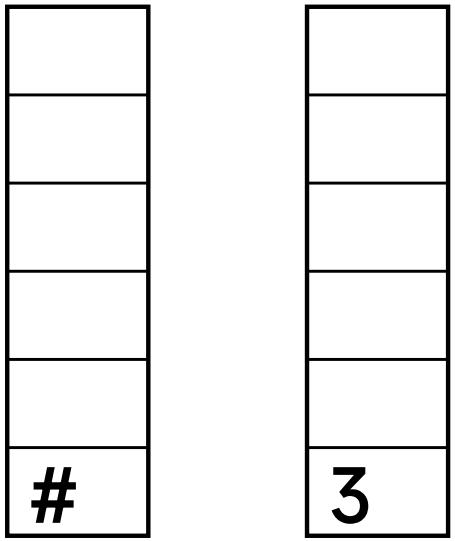


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

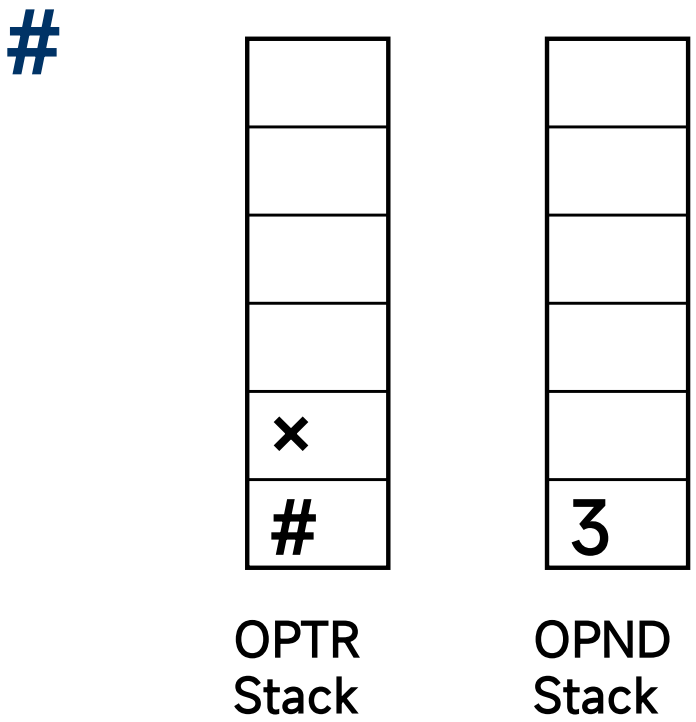
#



Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

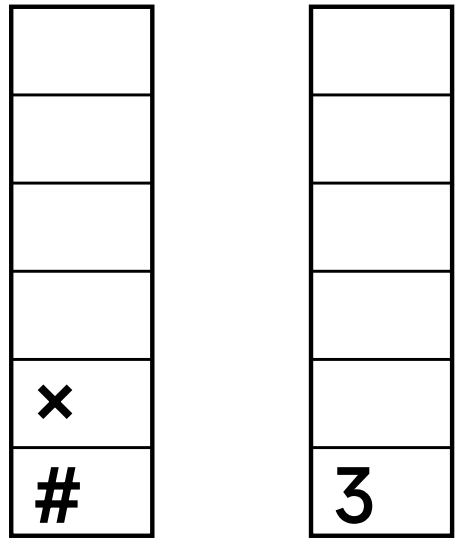


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

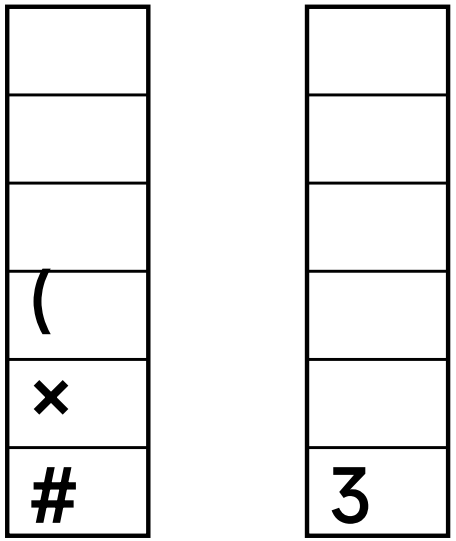


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

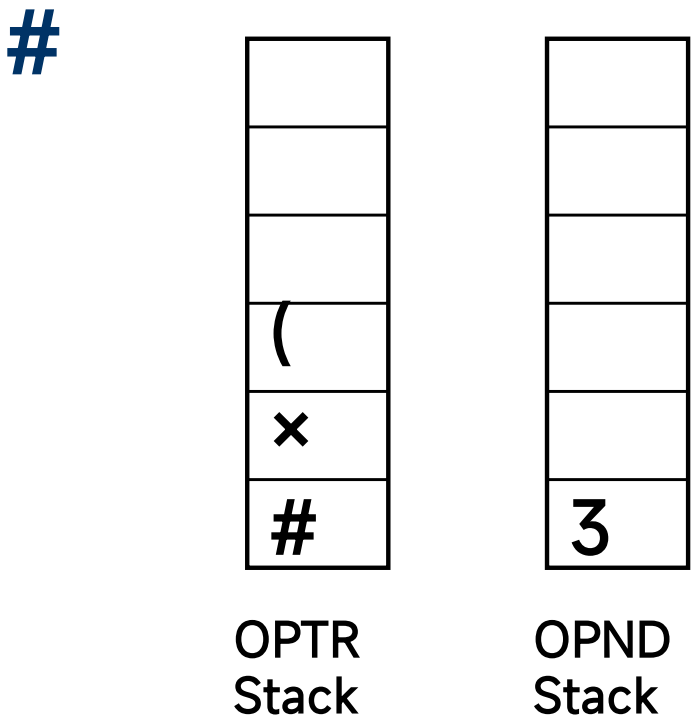
#



Infix

Ex： x = 3 × (7-2)

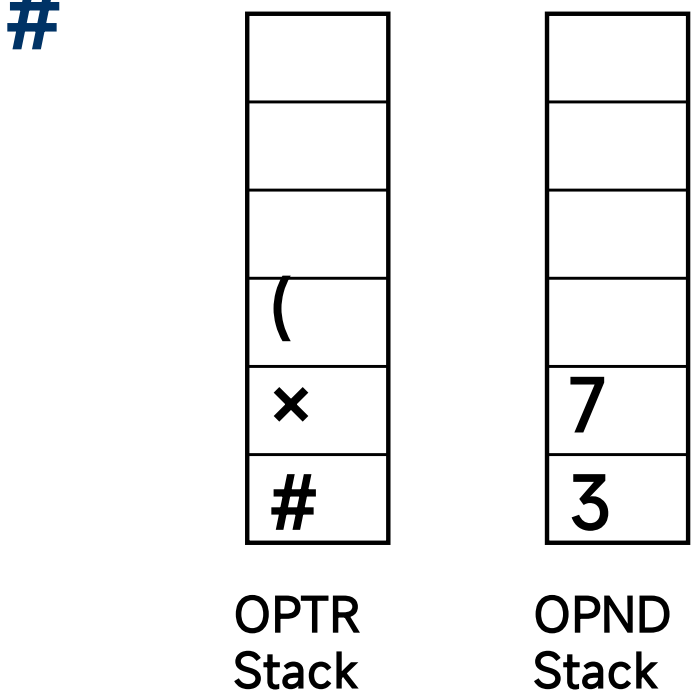
Process： # 3 × (7-2)



Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

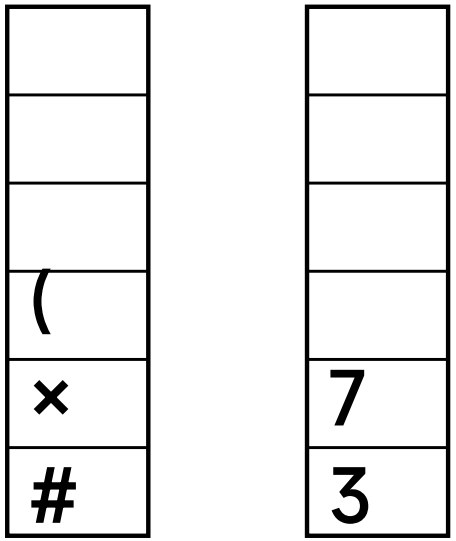


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

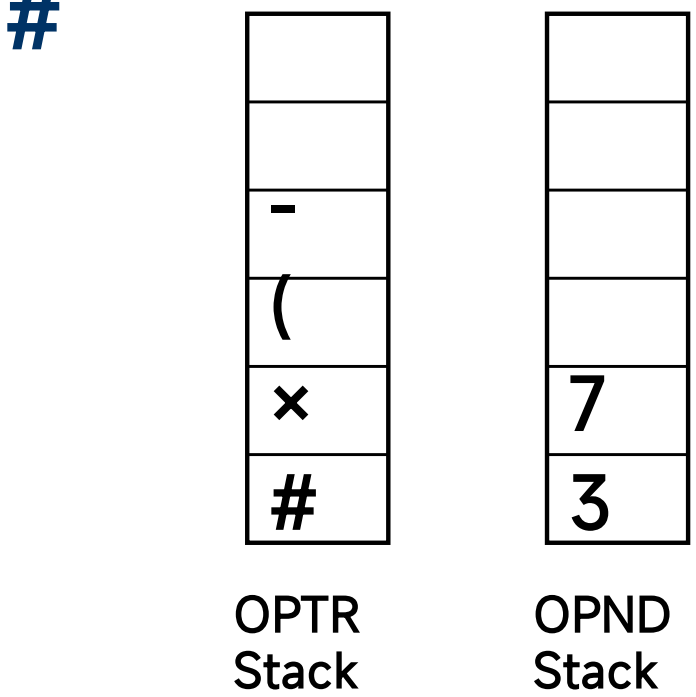
#



Infix

Ex： x = 3 × (7-2)

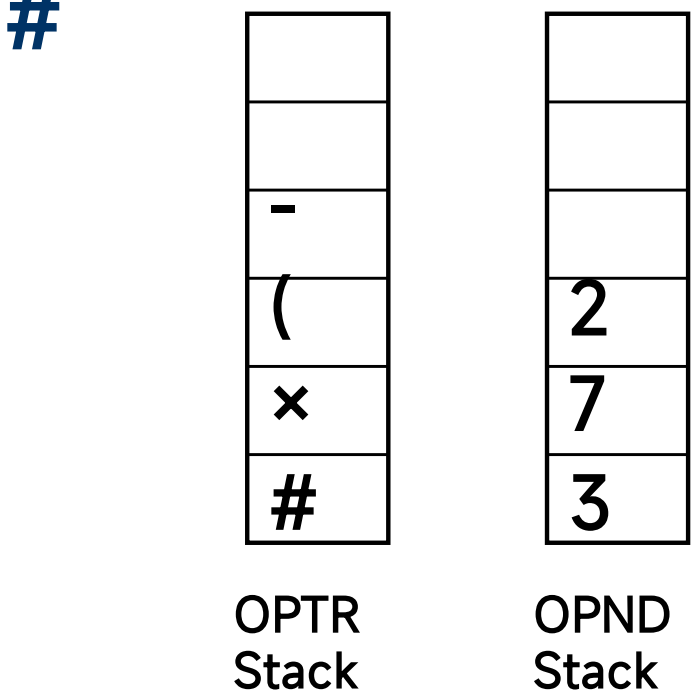
Process： # 3 × (7-2)



Infix

Ex： x = 3 × (7-2)

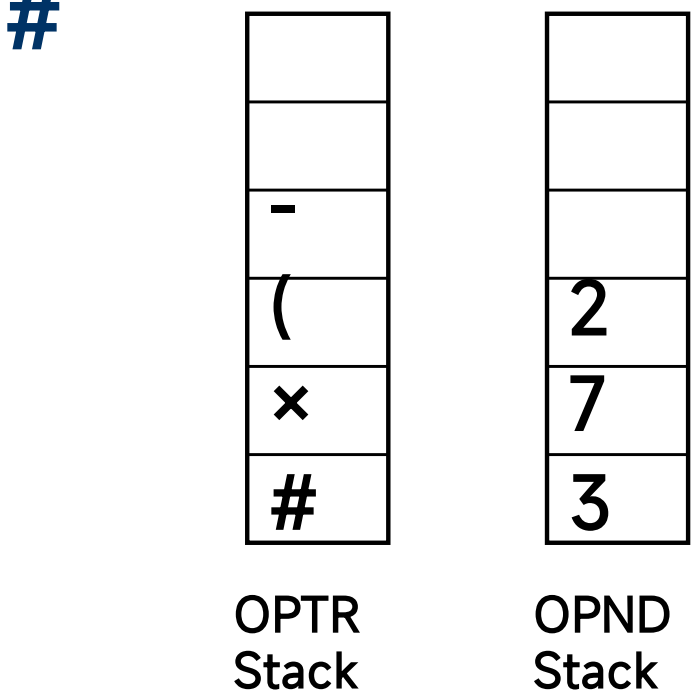
Process： # 3 × (7-2)



Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

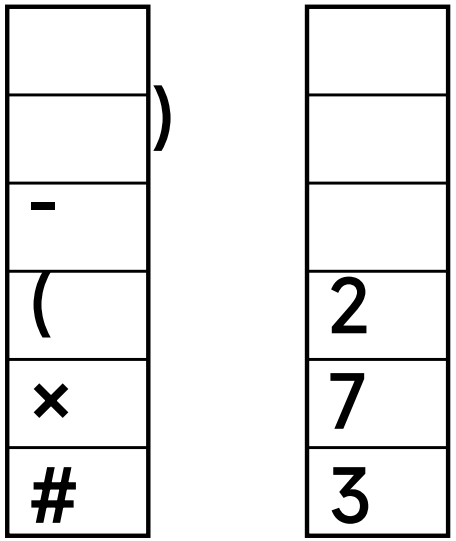


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

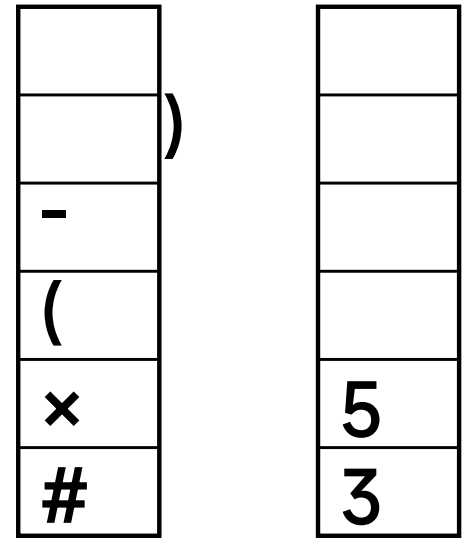
#



Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2) #

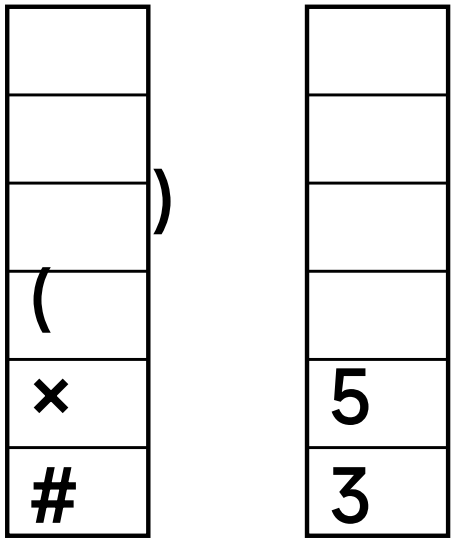


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

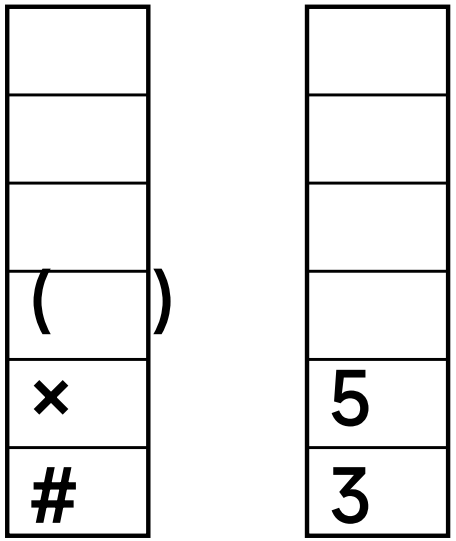


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

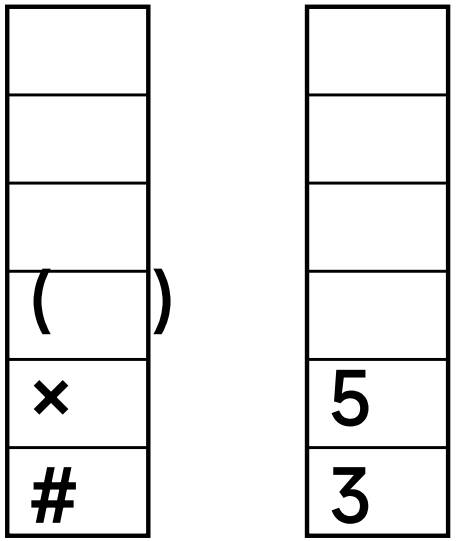


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

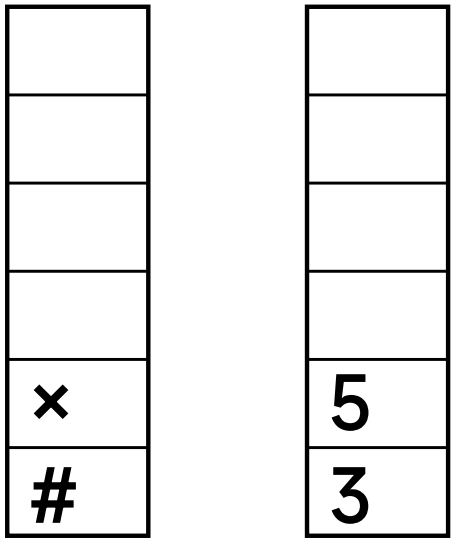


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

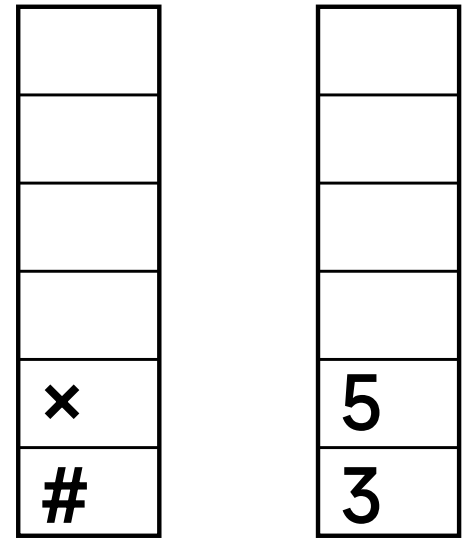


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

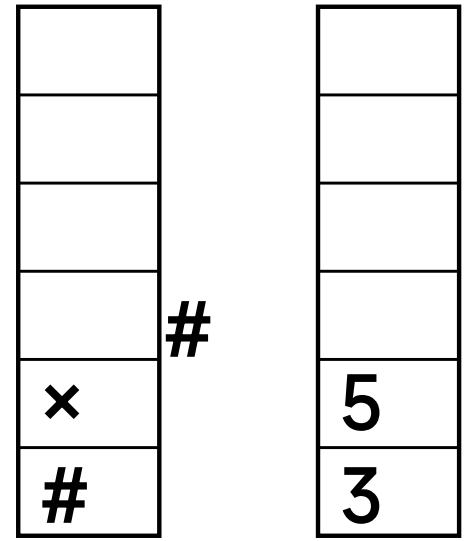


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

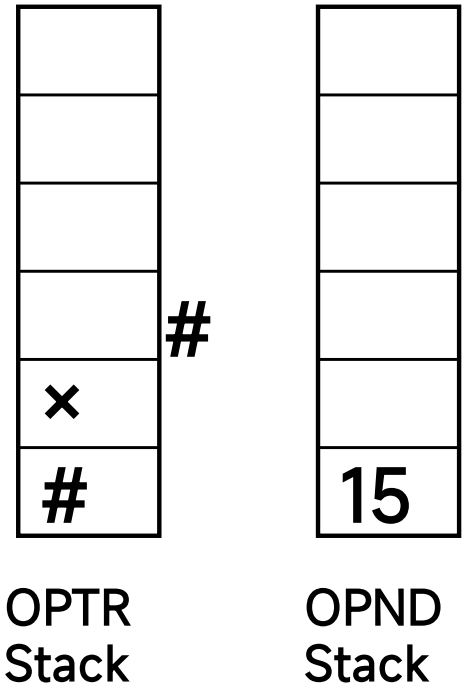


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

#

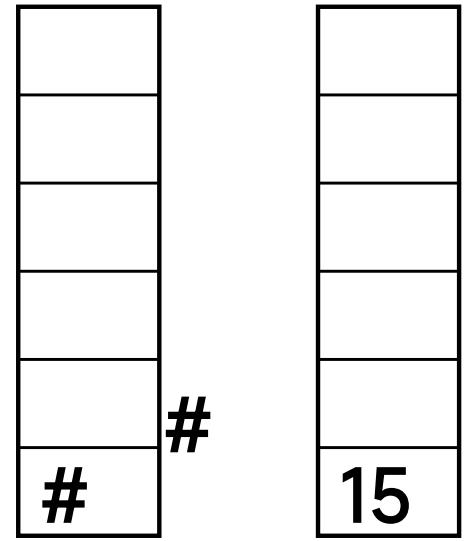


Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)

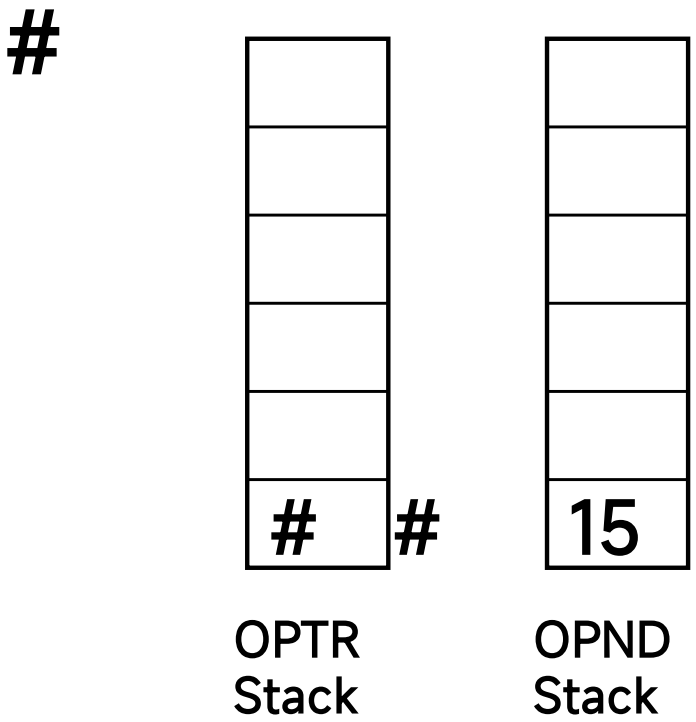
#



Infix

Ex： x = 3 × (7-2)

Process： # 3 × (7-2)



Reverse-Polish Notation(postfix)

Normally, mathematics is written using what we call *in-fix* notation:

$$(3 + 4) \times 5 - 6$$

The operator is placed between to operands

One weakness: parentheses are required

$$(3 + 4) \times 5 - 6 = 29$$

$$3 + 4 \times 5 - 6 = 17$$

$$3 + 4 \times (5 - 6) = -1$$

$$(3 + 4) \times (5 - 6) = -7$$

Reverse-Polish Notation(postfix)

Alternatively, we can place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3\ 4\ +\ 5\ \times\ 6\ -$$

Parsing reads left-to-right and performs any operation on the last two operands:

$$3\ 4\ +\ 5\ \times\ 6\ -$$

$$7\quad 5\ \times\ 6\ -$$

$$35\quad 6\ -$$

$$29$$

Reverse-Polish Notation

Other examples:

3 4 5 × + 6 −

3 20 + 6 −

23 6 −

17

3 4 5 6 − × +

3 4 −1 × +

3 −4 +

−1

Reverse-Polish Notation

Benefits:

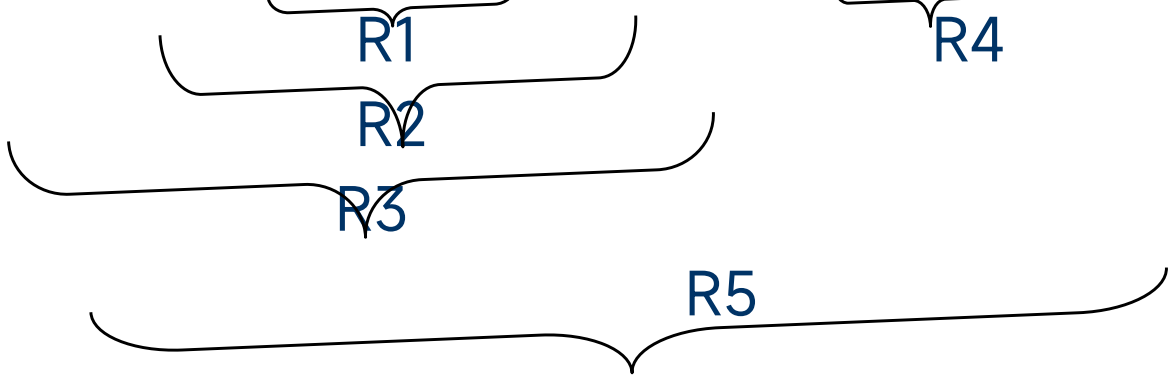
- No ambiguity and no brackets are required
- It is the same process used by a computer to perform computations:
  - operands must be loaded into registers before operations can be performed on them
- Reverse-Polish can be processed using stacks

Evaluate Postfix Expression

- From left to right, each item,
  - operands, push in stack;
  - Operators<op>, Pop two operands from stack (Y , X), do the calculation X<op>Y, push the calculation result in the stack.
- The top of stack is the final result when the scan is done.

Evaluate Postfix Expression

- A B C D - * + E F / -



- Is equal to：A+B*(C-D)-E/F

postfix

A B C D - * + E F / -

	Item	Do	Stack
1		Empty	Empty
2	A operand	Push	A
3	B operand	Push	A B
4	C operand	Push	A B C
5	D operand	Push	A B C D
6	- operator	Pop D,C, C-D, Push Rerult R1	A B R1
7	* operator	Pop R1,B, B*R1, push result R2	A R2
8	+ operator	Pop R2,A, A+R2, Push result R3	R3
9	E operand	Push	R3 E
10	F operand	Push	R3 E F
11	/ operator	Pop F,E, E/F, Push result R4	R3 R4
12	- operator	Pop R4,R3,R3-R4,Push resultR5	R5

Reverse-Polish Notation

Evaluate the following reverse-Polish expression using a stack:

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

Reverse-Polish Notation

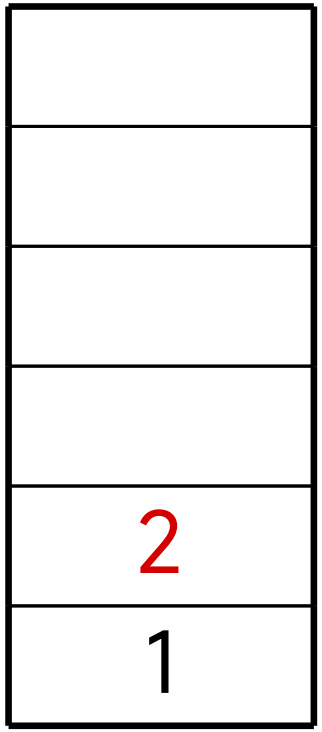
Push 1 onto the stack

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

Reverse-Polish Notation

Push 1 onto the stack

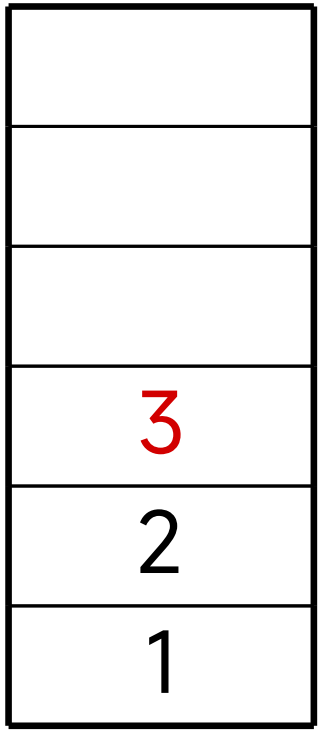
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 3 onto the stack

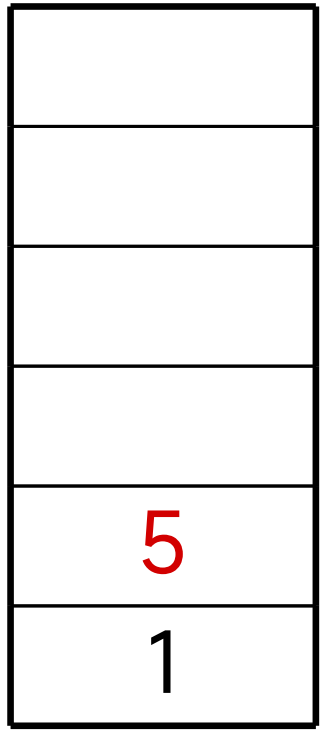
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 3 and 2 and push 2 + 3 = 5

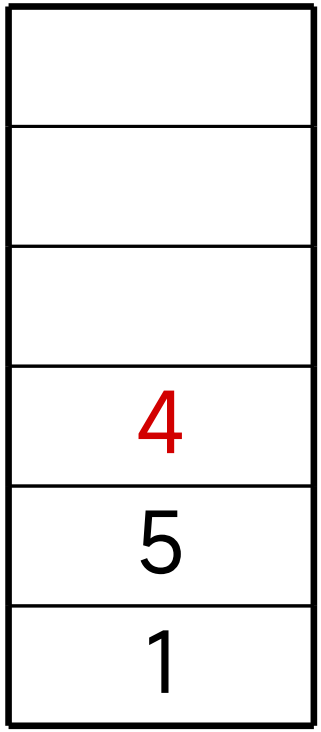
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 4 onto the stack

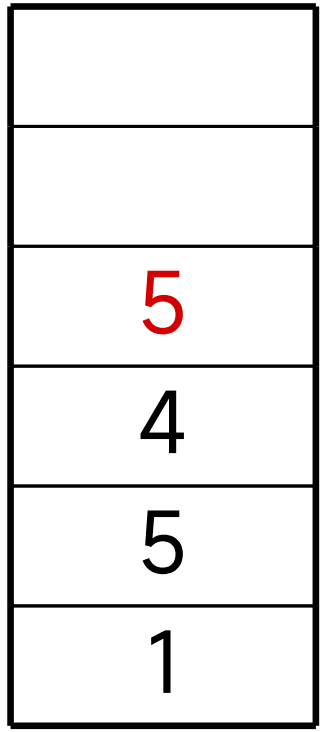
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 5 onto the stack

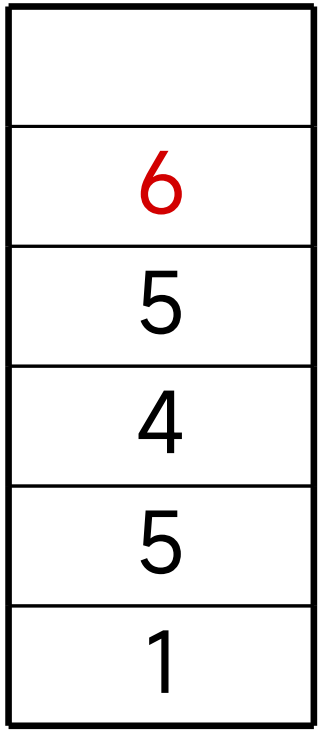
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 6 onto the stack

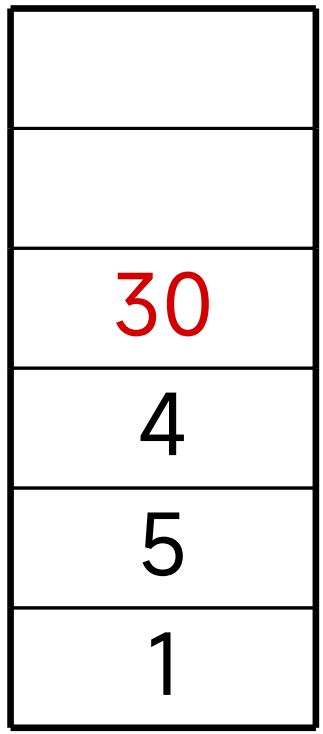
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 6 and 5 and push 5 × 6 = 30

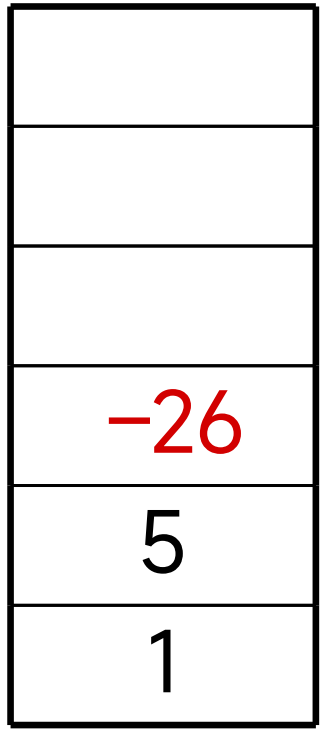
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 30 and 4 and push 4 − 30 = −26

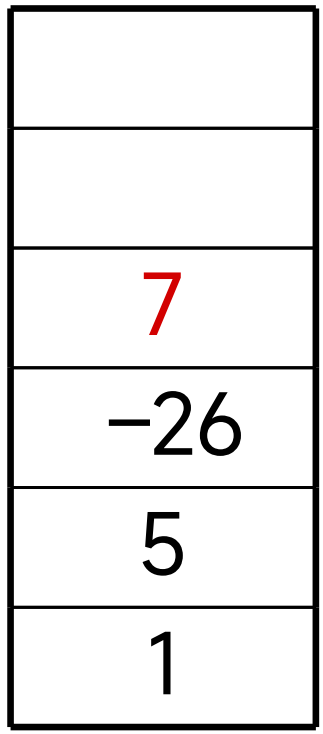
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 7 onto the stack

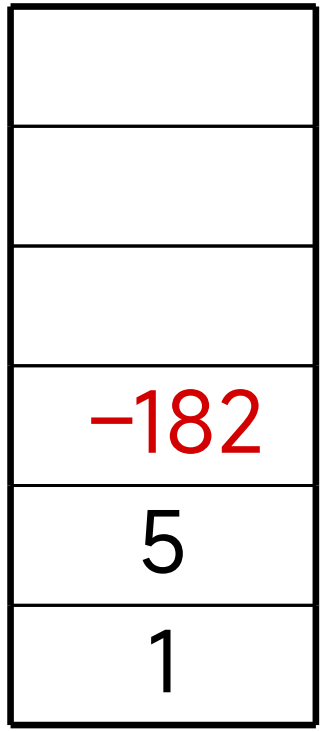
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 7 and −26 and push −26 × 7 = −182

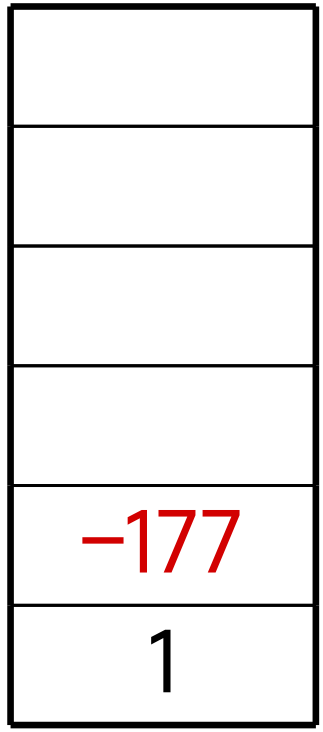
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop −182 and 5 and push −182 + 5 = −177

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

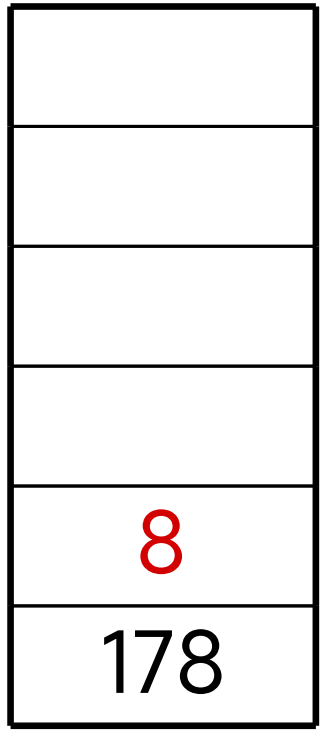
Pop −177 and 1 and push 1 − (−177) = 178

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

Reverse-Polish Notation

Push 8 onto the stack

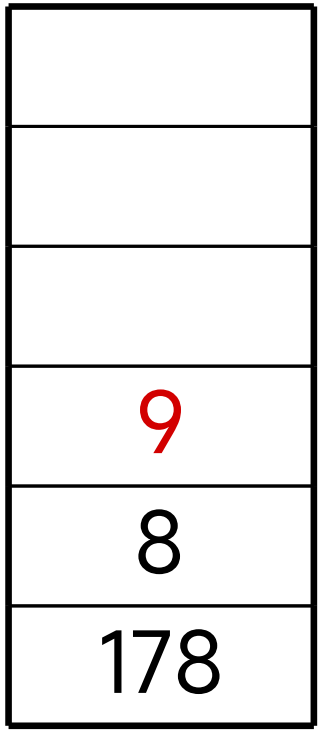
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 1 onto the stack

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

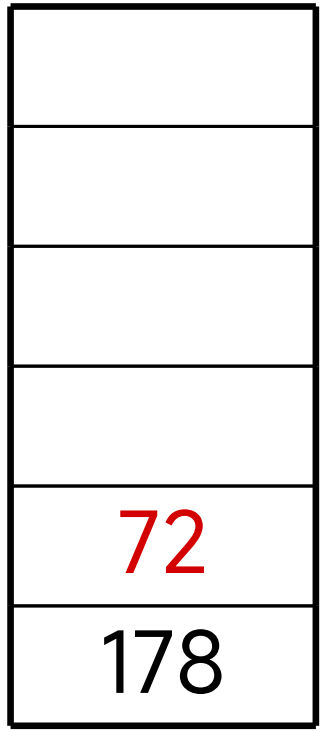


Reverse-Polish Notation

Pop 9 and 8 and push 8 × 9 = 72

1 2 3 + 4 5 6 × − 7 × + − 8 9 ×

+



Reverse-Polish Notation

Pop 72 and 178 and push 178 + 72 = 250

1 2 3 + 4 5 6 × − 7 × + − 8 9 ×

+

Recursion

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type
- A procedure that is defined in terms of itself

Recursion

When you turn that into a program, you end up with functions that call themselves:

Recursive Functions

Recursion


This video is part of
Algorithms in Motion
from Manning Publications
Recursion

Recursion

What's behind this function ?

```
public int f(int a){
        if (a==1)
            return(1);
        else
            return(a * f( a-1));
}
```

It computes f! (factorial)

Factorial

Factorial:

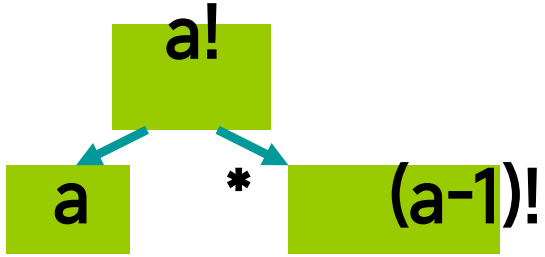
$$a! = 1 * 2 * 3 * \ldots * (a-1) * a$$

Note:

$$a! = a * (a-1)!$$

remember:

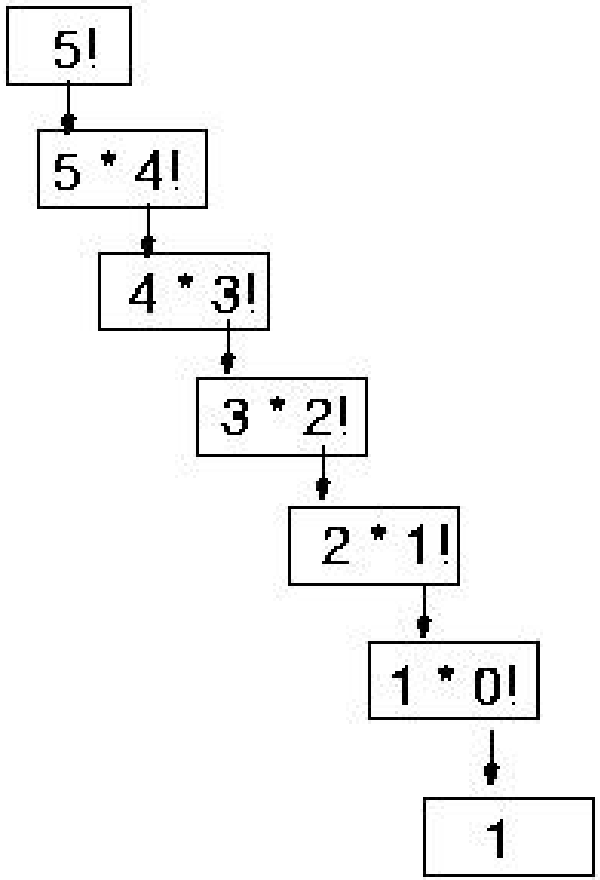
...splitting up the problem into a smaller problem of the same type...

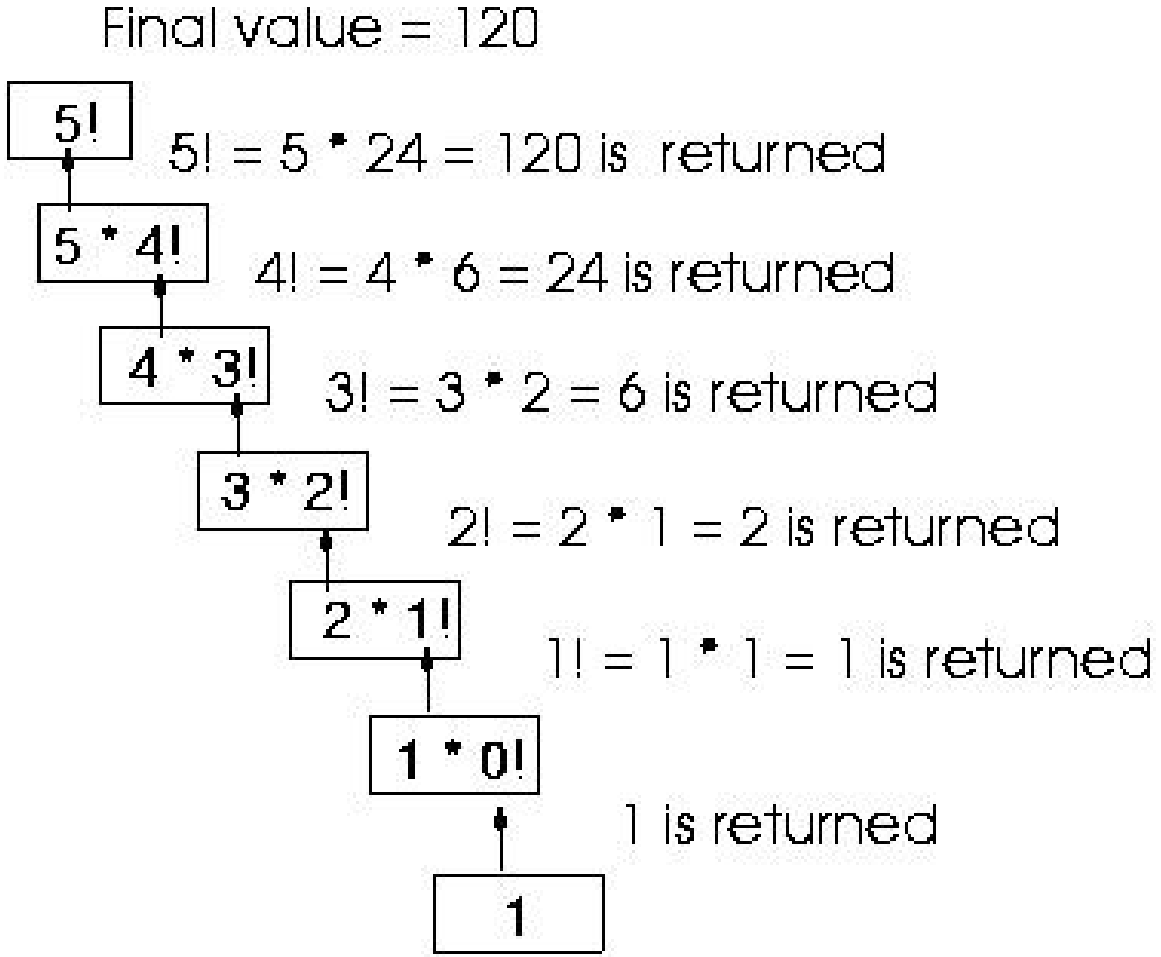


Tracing the example

```
public int factorial(int a){
        if (a==0)
            return(1);
        else
            return(a * factorial( a-1));
}
```

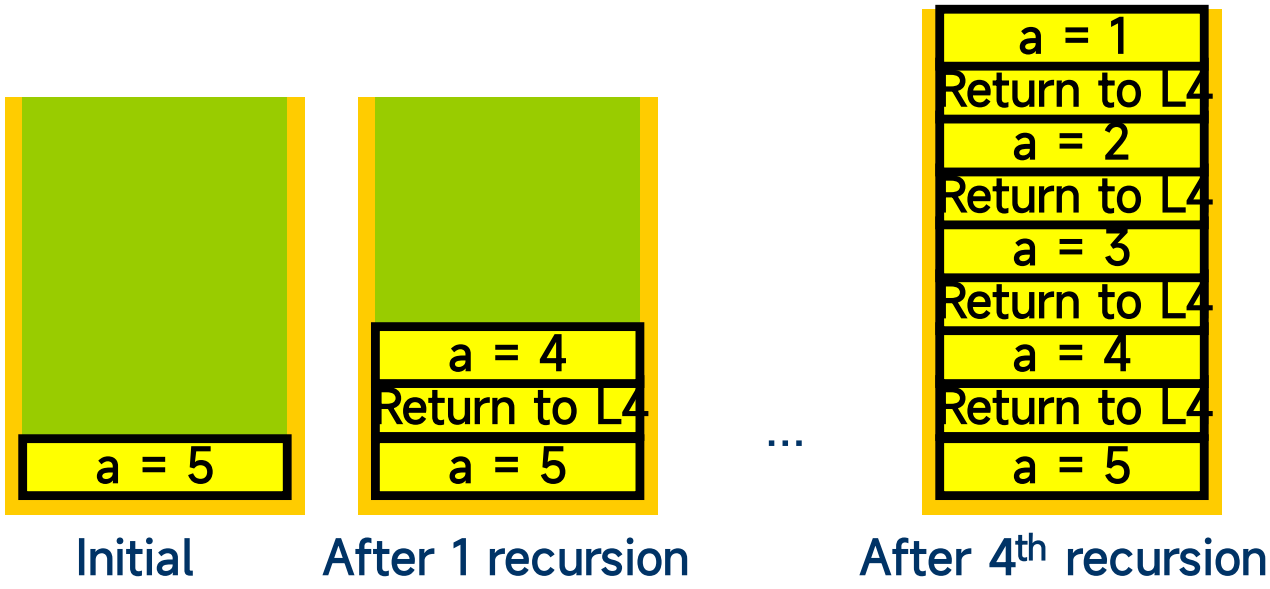
RECURSION !





Watching the Stack

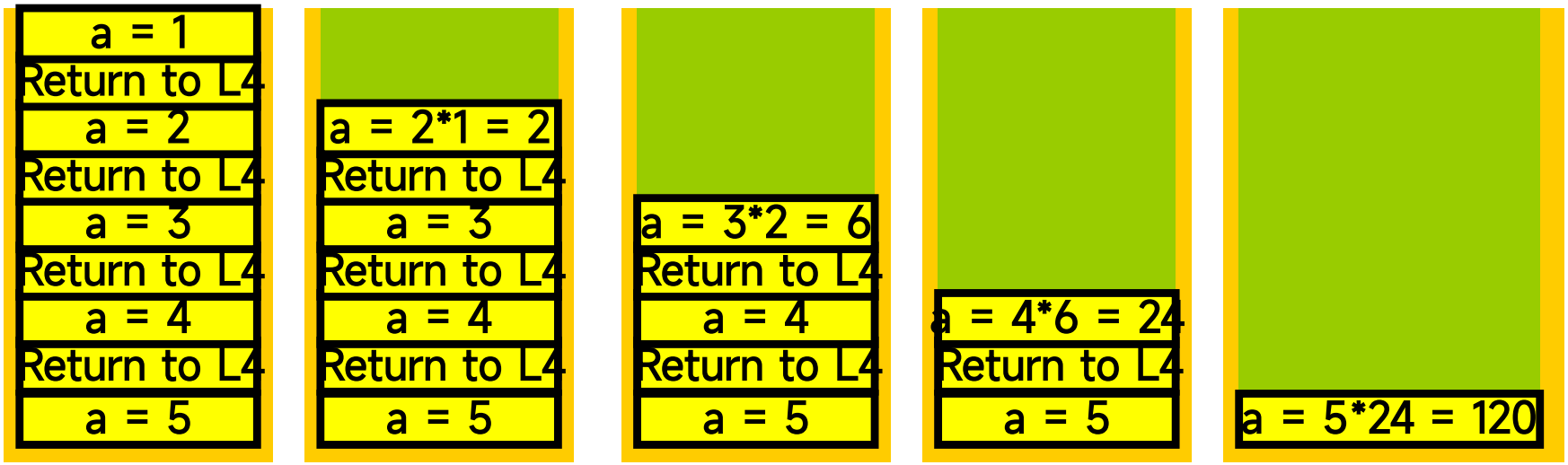
```
public int factorial(int a){
        if (a==1)
            return(1);
        else
            return(a * factorial( a-1));
}
```



Every call to the method creates a new set of local variables !

Watching the Stack

```
public int factorial(int a){
        if (a==1)
            return(1);
        else
            return(a * factorial( a-1));
}
```



Properties of Recursive Functions

Problems that can be solved by recursion have these characteristics:

- One or more stopping cases have a simple, nonrecursive solution
- The other cases of the problem can be reduced (using recursion) to problems that are closer to stopping cases
- Eventually the problem can be reduced to only stopping cases, which are relatively easy to solve

Follow these steps to solve a recursive problem:

- Try to express the problem as a simpler version of itself
- Determine the stopping cases
- Determine the recursive steps

Recursion vs. Iteration

You could have written the power-function *iteratively*, i.e. using a loop construction

Where's the difference ?

Recursion vs. Iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code
- (Nearly) every recursively defined problem can be solved iteratively iterative optimization can be implemented after recursive design

Deciding whether to use a Recursive Function

- When the depth of recursive calls is relatively “shallow”
- The recursive version does about the same amount of work as the nonrecursive version
- The recursive version is shorter and simpler than the nonrecursive solution

The Relationship Between Stacks and Recursion

- A strong relationship exists between recursion and stacks
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm

Homework

- Please refer to Icourse，Huawei Cloud.

- Due date for quiz: 23:30　2022/3/22
- Due date for homework: 23:30 2022/3/27
- Due data for online lab assignment:　2022/3/27 23：30