Stack

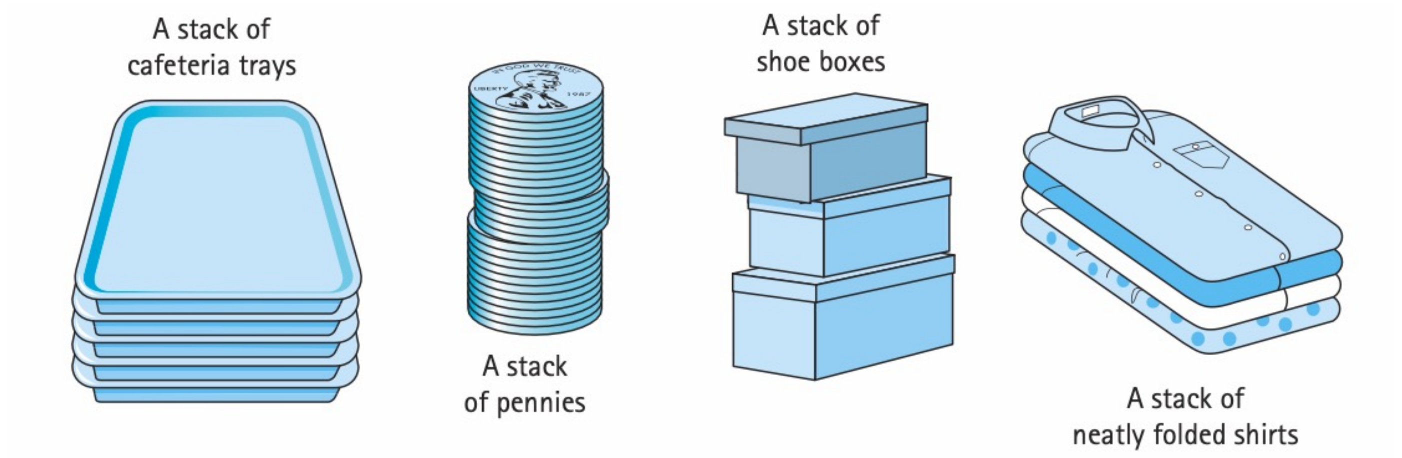
Outline

- Stack
- Array Implementation
- Applications

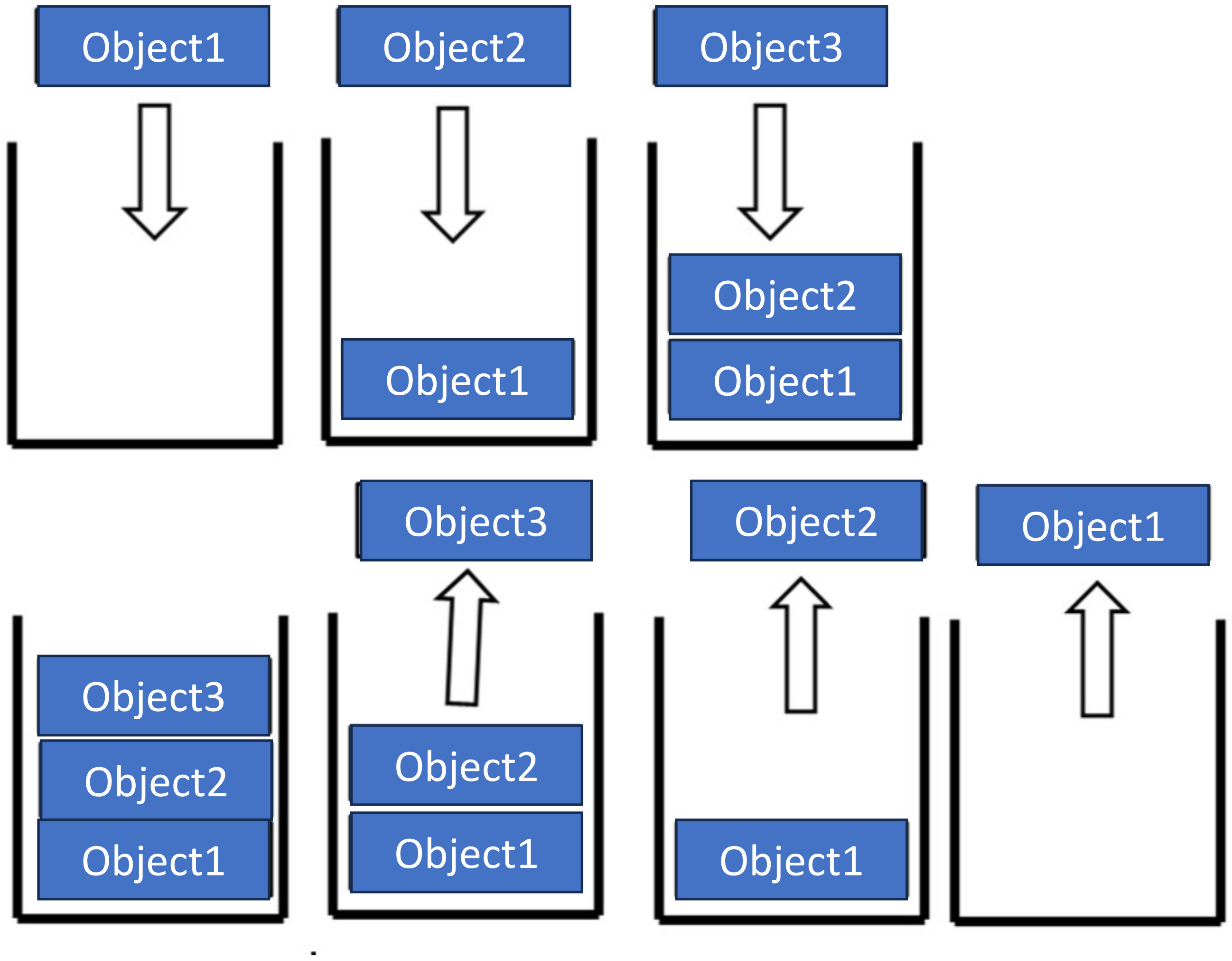
Stack

An Abstract Stack (Stack ADT) is an abstract data type which emphasizes specific operations:

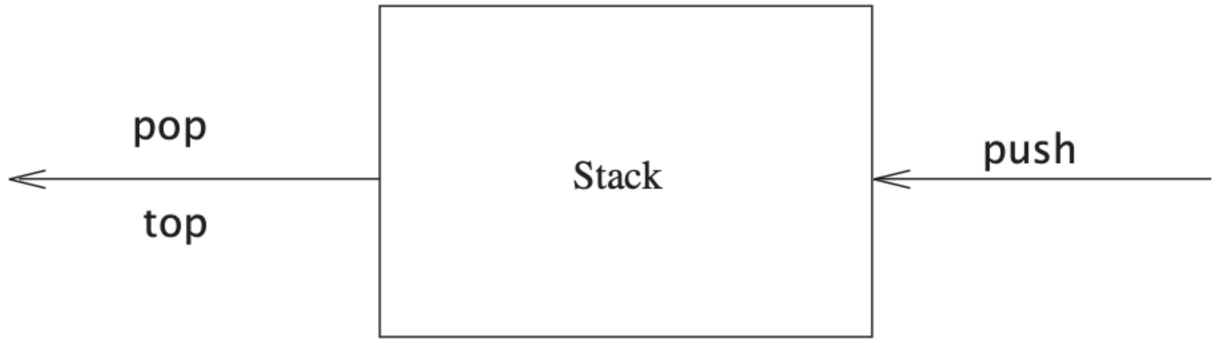
- Uses a explicit linear ordering
- Insertions and removals are performed individually
- Inserted objects are *pushed* *onto* the stack
- The *top* of the stack is the most recently object pushed onto the stack
- When an object is *popped* from the stack, the current *top* is erased

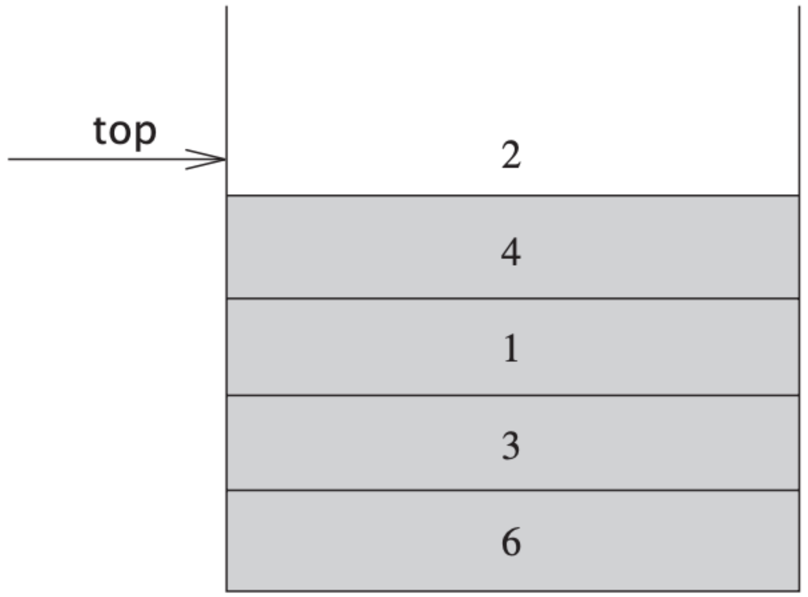


Last In-First Out (LIFO)



Stack Model





Operations of Stack

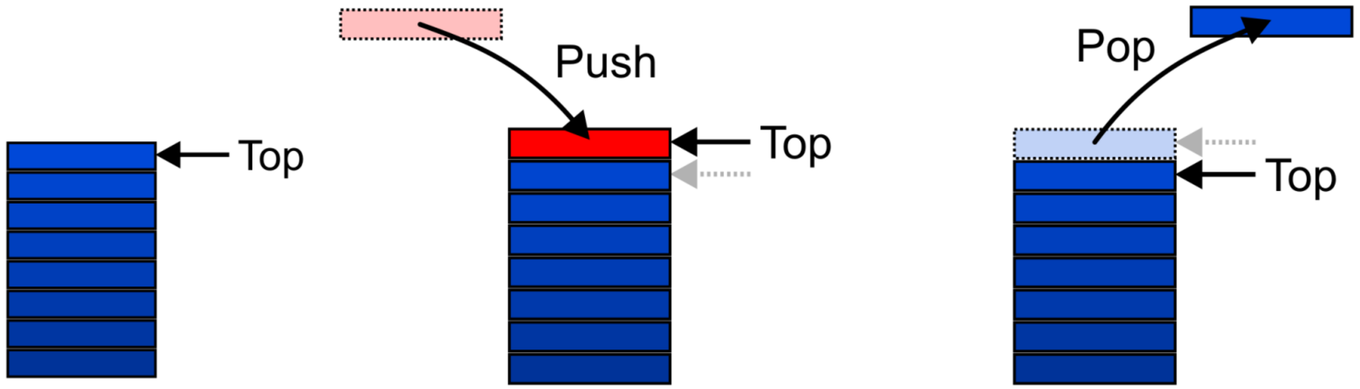
- Push
- Pop

Only the top element is accessible.

Stack

Also called a *last-in–first-out* (LIFO) behaviour

- Graphically, we may view these operations as follows:



There are two exceptions associated with abstract stacks:

- It is an undefined operation to call either pop or top on an empty stack

Implementations

We will look at two implementations of stacks:

The optimal asymptotic run time of any algorithm is $\mathbf{O}(1)$

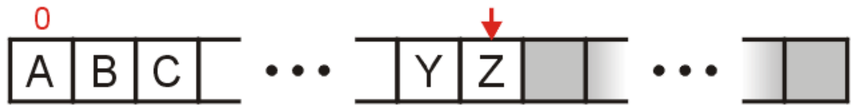
- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

We will look at

- Singly linked lists
- One-ended arrays

Array Implementation

For one-ended arrays, all operations at the back are $\mathrm{O}(1)$



	Front/1st	Back/nth
Find	$\mathrm{O}(1)$	$\mathrm{O}(1)$
Insert	$\mathrm{O}(n)$	$\mathrm{O}(1)$
Erase	$\mathrm{O}(n)$	$\mathrm{O}(1)$

Array Implementation

We need to store an array:

- In C++, this is done by storing the address of the first entry

```
Type *array;
```

We need additional information, including:

- The number of objects currently in the stack

```
int stack_size;
```

- The capacity of the array

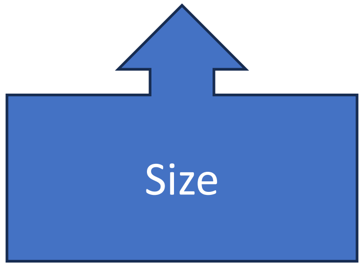
```
int array_capacity;
```

Array Implementation

We need to store an array:

- In C++, this is done by storing the address of the first entry

```
int Array[5];
```



Stack-as-Array Class

```
template <typename Type>
class Stack {
    public:
            int top = 0;
            int Array[5];

            Stack() { Array[0] = 4;} ;
            bool empty();
            bool full();
            void push(int data);
            int pop();
            void print();
};
```

empty()

The stack is empty if top is equal to zero:

```
bool empty()
        if(top == 0)
                return true;
        else
                return false;
}
```

```
#include <bits/stdc++.h>
using namespace std;
class Stack
{
  public:
  int Array[5];
  int top = 0;
  Stack() { Array[0] = 4; }
  bool empty()
  {
    if(top == 0)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  bool full()
  {
    if( top == Array[0] )
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  void push(int data)
  {
    if( !full() )
    {
      top++;
      Array[top] = data;
    }
  }
  int pop()
  {
    if( !empty() )
    {
      int temp = Array[top];
      top--;
      return temp;
    }
    return -1;
  }
  void print()
  {
    cout<<"Stack : ";
    for (int i = 1 ; i <= top ; i++)
    {
      cout << Array[i] << " ";
    }
    cout<<endl;
  }
};
```

full()

The stack is full if top is equal to its size:

```
bool full()
        if(top == Array[0])
                return true;
        else
                return false;
}
```

pop()

Removing an object simply involves reducing the size

```
int pop(){
        if(!empty()) {
                int temp = Array[top];
                top--;
                return temp;
        }
        return -1;
}
```

push()

Pushing an object onto the stack can only be performed if the array is not full

```
void push(int data){
        if( !full() ) {
                top++;
                Array[top] = data;
        }
}
```

Stack in C++

```
#include <bits/stdc++.h>
using namespace std;
class Stack
{
    public:
    int Array[5];
    int top = 0;
    Stack() { Array[0] = 4; }
    bool empty()
    {
        if(top == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    bool full()
    {
        if( top == Array[0] )
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    void push(int data)
    {
        if( !full() )
        {
            top++;
            Array[top] = data;
        }
    }
    int pop()
    {
        if( !empty() )
        {
            int temp = Array[top];
            top--;
            return temp;
        }
        return -1;
    }
    void print()
    {
        cout<<"Stack : ";
        for (int i = 1 ; i <= top ; i++)
        {
            cout << Array[i] << " ";
        }
        cout<<endl;
    }
};
int main()
{
    Stack s;
    s.push(1);  cout<<"Push 1\t";
    s.print();
    s.push(2);  cout<<"Push 2\t";
    s.print();
    s.push(3);  cout<<"Push 3\t";
    s.print();
    s.push(4);  cout<<"Push 4\t";
    s.print();
    s.push(5);  cout<<"Push 5\t";
    s.print();
    cout<<"Pop "<<s.pop()<<"\t";
    s.print();
    cout<<"Pop "<<s.pop()<<"\t";
    s.print();
    cout<<"Pop "<<s.pop()<<"\t";
    s.print();
    cout<<"Pop "<<s.pop()<<"\t";
    s.print();
    cout<<"Pop "<<s.pop()<<"\t";
    s.print();
    return 0;
}
```

Applications

Numerous applications:

- Parsing code:
  - Matching parenthesis
  - XML (e.g., XHTML)
- Tracking function calls
- Dealing with undo/redo operations
- Reverse-Polish calculators
- Assembly language

The stack is a very simple data structure

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Stack: Applications

Problem solving:

- Solving one problem may lead to subsequent problems
- These problems may result in further problems
- As problems are solved, your focus shifts back to the problem which lead to the solved problem

Notice that function calls behave similarly:

- A function is a collection of code which solves a problem

Reference: Donald Knuth

Application: Parsing

Most parsing uses stacks

Examples includes:

- Matching tags in XHTML
- In C++, matching
  - parentheses **(...)**
  - brackets, and **[...]**
  - braces **{ ... }**

Application: Parsing XHTML

The first example will demonstrate parsing XHTML

We will show how stacks may be used to parse an XHTML document

You will use XHTML (and more generally XML and other markup languages) in the workplace

Application: Parsing XHTML

A *markup language* is a means of annotating a document to given context to the text

- The annotations give information about the structure or presentation of the text

The best known example is HTML, or HyperText Markup Language

- We will look at XHTML

Application: Parsing XHTML

XHTML is made of nested

- *opening tags*, e.g., <some_identifier>, and
- matching *closing tags*, e.g., </some_identifier>

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

Application: Parsing XHTML

Nesting indicates that any closing tag must match the most recent opening tag

Strategy for parsing XHTML:

- read though the XHTML linearly
- place the opening tags in a stack
- when a closing tag is encountered, check that it matches what is on top of the stack

Application: Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>			

Application: Parsing XHTML

```
<html>
 <head><title>Hello</title></head>
 <body><p>This appears in the
 <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Application: Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

`<html>`	`<head>`	`<title>`	

Application: Parsing XHTML

```
<html>
 <head><title>Hello</title></head>
 <body><p>This appears in the
 <i>browser</i>.</p></body>
</html>
```

`<html>`	`<head>`		

Application: Parsing XHTML

```
<html>
 <head><title>Hello</title></head>
 <body><p>This appears in the
 <i>browser</i>.</p></body>
</html>
```

`<html>`	`<body>`	`<p>`	`<i>`

Application: Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

`<html>`	`<body>`	`<p>`	

Application: Parsing XHTML

```
<html>
 <head><title>Hello</title></head>
 <body><p>This appears in the
 <i>browser</i>.</p></body>
</html>
```

<html>			

XML

XHTML is an implementation of XML

XML defines a class of general-purpose *eXtensible Markup Languages* designed for sharing information between systems

The same rules apply for any flavour of XML:

- opening and closing tags must match and be nested

Reverse-Polish Notation

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

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:

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

$$7 \;\; 5 \;\; \times \;\; 6 \;\; -$$

$$35 \;\; 6 \;\; -$$

$$29$$

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

Reverse-Polish Notation

Reverse-Polish notation is used with some programming languages

- *e.g.*, postscript, pdf, and HP calculators

Similar to the thought process required for writing assembly language code

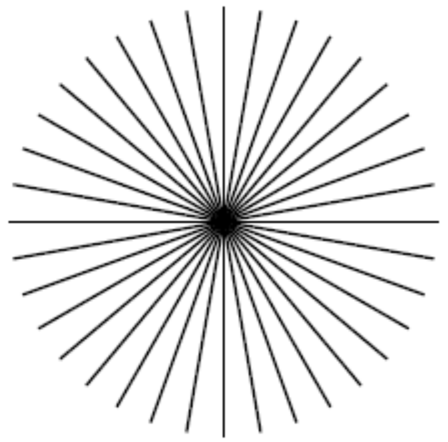
- you cannot perform an operation until you have all of the operands loaded into registers

```
MOVE.L #$2A, D1       ; Load  42 into Register D1
MOVE.L #$100, D2      ; Load 256 into Register D2
ADD D2, D1            ; Add D2 into D1
```

Reverse-Polish Notation

A quick example of postscript:

```
0 10 360 {          % Go from 0 to 360 degrees in 10-degree steps
  newpath           % Start a new path
  gsave             % Keep rotations temporary
    144 144 moveto
    rotate          % Rotate by degrees on stack from 'for'
    72 0 rlineto
    stroke
  grestore          % Get back the unrotated state
} for % Iterate over angles
```



http://www.tailrecursive.org/postscript/examples/rotate.html

Reverse-Polish Notation

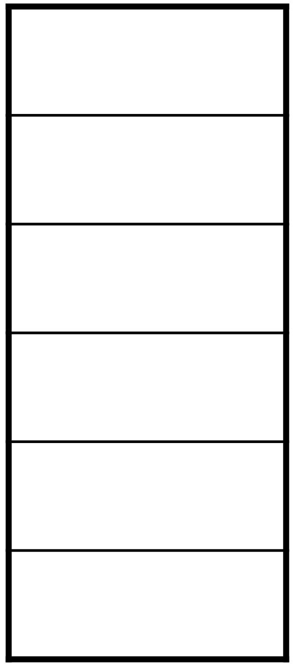
The easiest way to parse reverse-Polish notation is to use an operand stack:

- operands are processed by pushing them onto the stack
- when processing an operator:
    - pop the last two items off the operand stack,
    - perform the operation, and
    - push the result back onto the stack

Reverse-Polish Notation

Evaluate the following reverse-Polish expression using a stack:

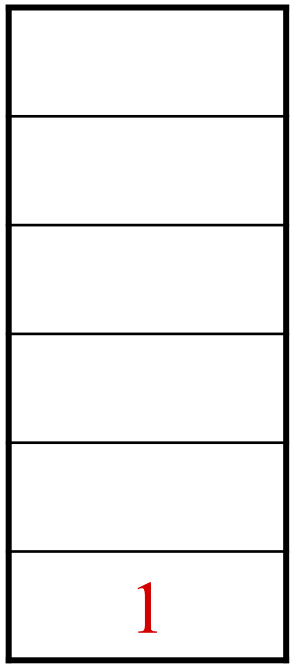
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Reverse-Polish Notation

Push 1 onto the stack

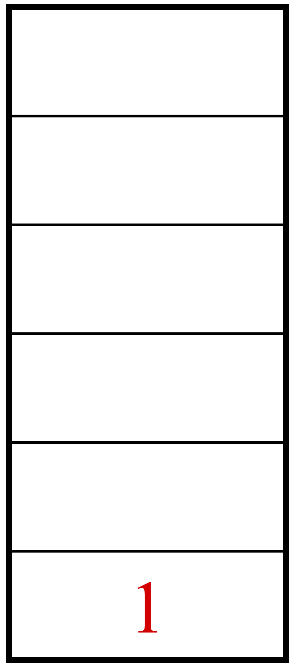
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 1 onto the stack

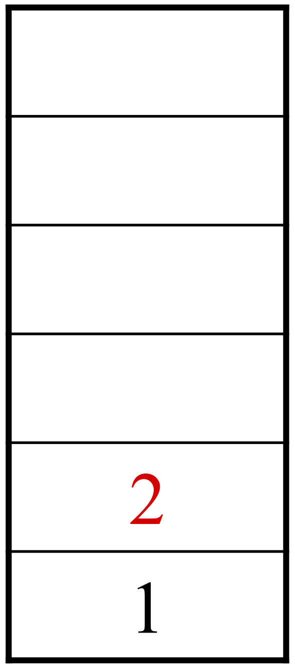
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 1 onto the stack

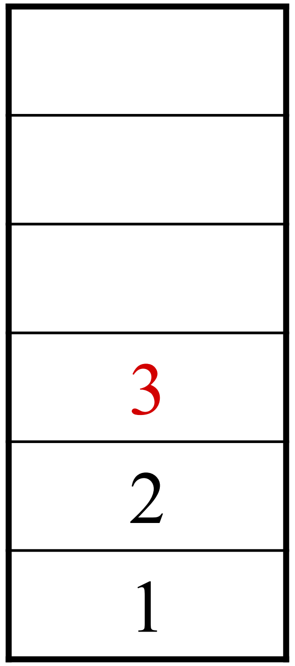
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 3 onto the stack

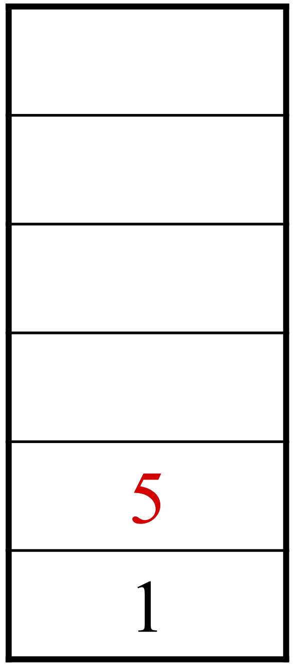
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 3 and 2 and push 2 + 3 = 5

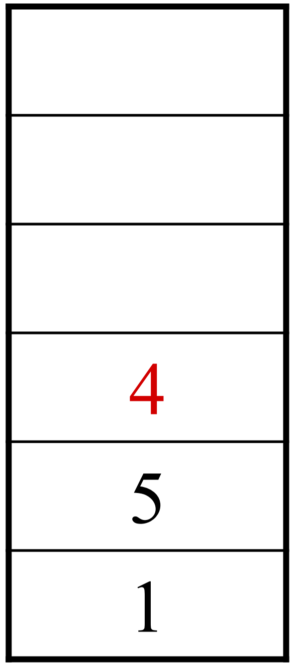
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 4 onto the stack

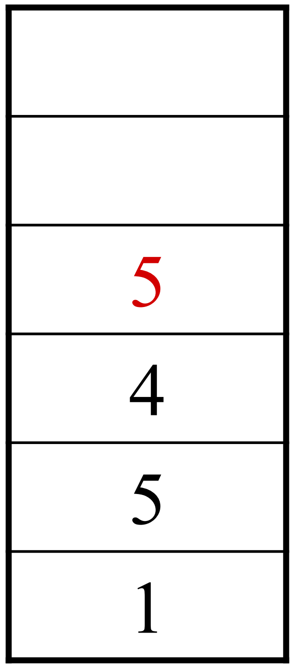
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 5 onto the stack

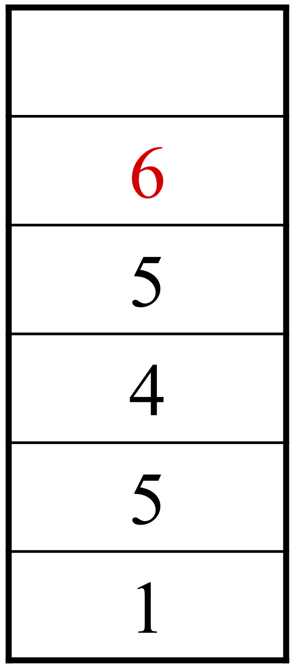
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 6 onto the stack

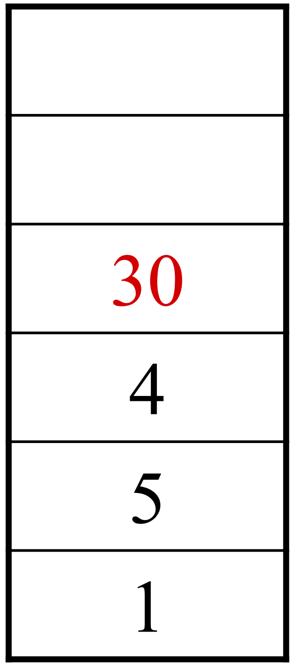
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 6 and 5 and push 5 × 6 = 30

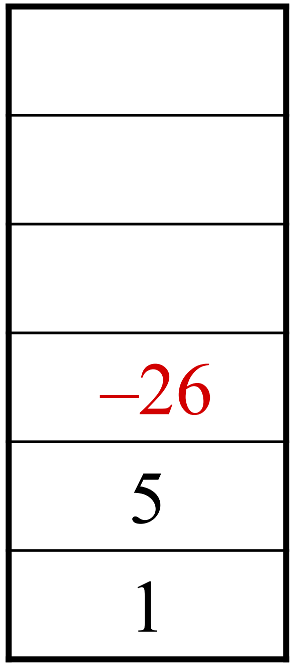
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 30 and 4 and push 4 – 30 = –26

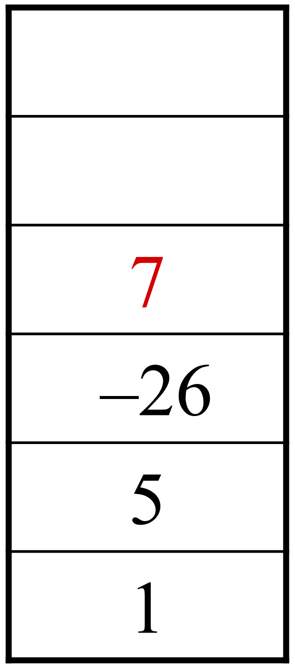
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 7 onto the stack

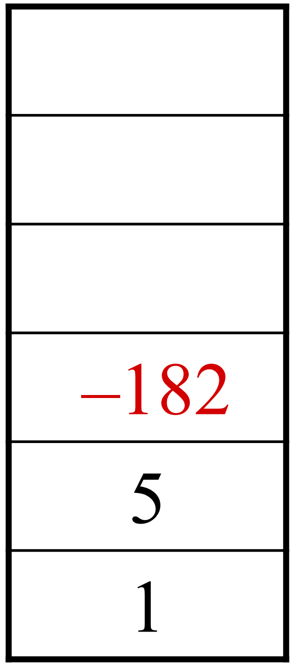
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 7 and –26 and push –26 × 7 = –182

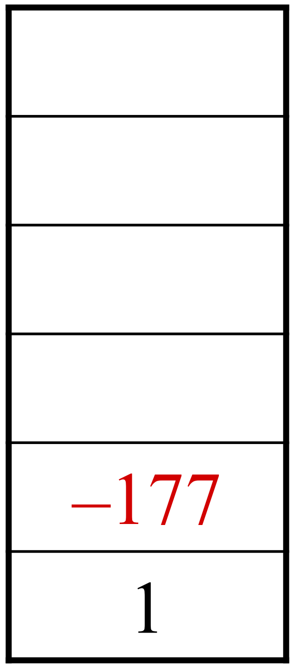
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

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

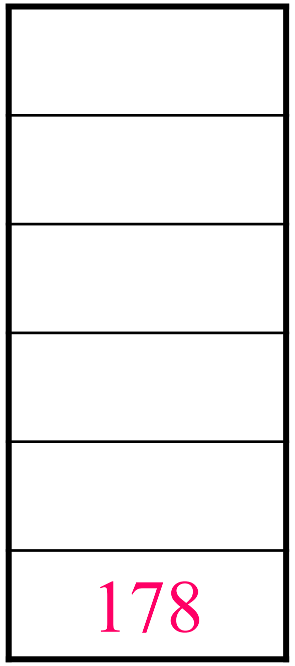
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

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

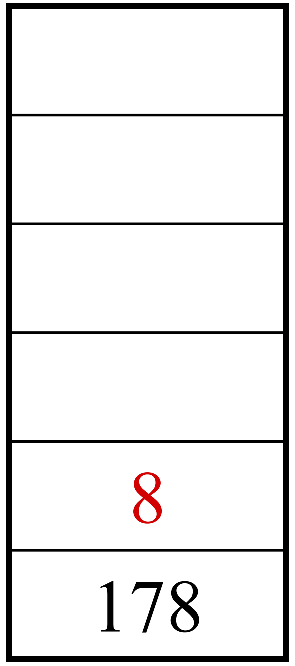
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 8 onto the stack

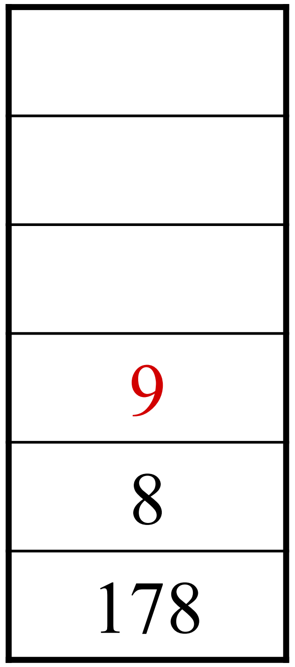
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Push 9 onto the stack

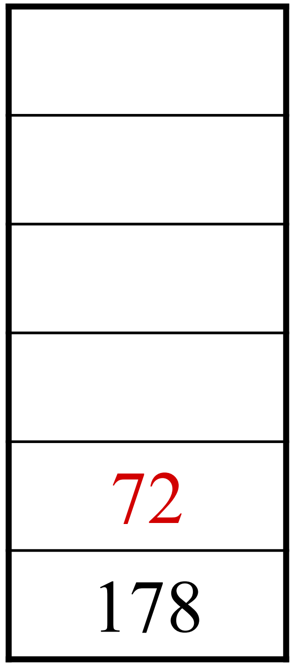
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 9 and 8 and push 8 × 9 = 72

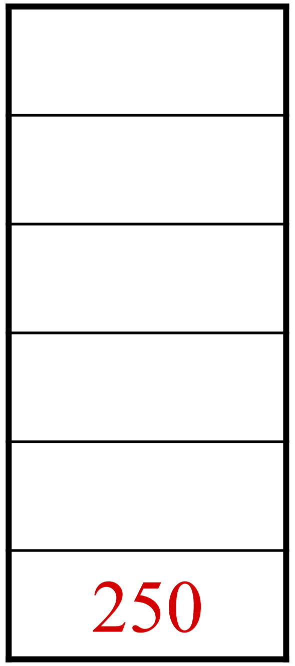
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Pop 72 and 178 and push 178 + 72 = 250

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

Thus

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

evaluates to the value on the top: 250

The equivalent in-fix notation is

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$

We reduce the parentheses using order-of-operations:

$$1 - (2 + 3 + (4 - 5 \times 6) \times 7) + 8 \times 9$$

Reverse-Polish Notation

Incidentally,

$$1 - 2 + 3 + 4 - 5 \times 6 \times 7 + 8 \times 9 = -132$$

which has the reverse-Polish notation of

$$1\ 2\ -\ 3\ +\ 4\ +\ 5\ 6\ 7\ \times\ \times\ -\ 8\ 9\ \times\ +$$

For comparison, the calculated expression was

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

Standard Template Library

The Standard Template Library (STL) has a *wrapper* class stack with the following declaration:

```
template <typename T>
class stack {
    public:
        stack();                    // not quite true...
        bool empty() const;
        int size() const;
        const T & top() const;
        void push( const T & );
        void pop();
};
```

Standard Template Library

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> istack;
    istack.push( 13 );
    istack.push( 42 );
    cout << "Top: " << istack.top() << endl;
    istack.pop();                         // no return value
    cout << "Top: " << istack.top() << endl;
    cout << "Size: " << istack.size() << endl;
    return 0;
}
```

Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/