Data Structures

Week 5: Stack and its applications

Irfan Mehmood

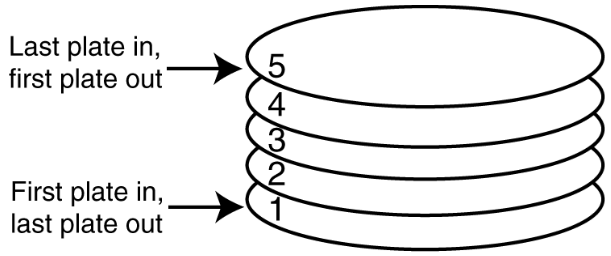
Sejong University

Yulgokgwan 205B

irfan@sejong.ac.kr

Introduction to the Stack

- A stack is a data structure that stores and retrieves items in a last-in-first-out (LIFO) manner.



Static and Dynamic Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list

Stack Operations

- Push
  - causes a value to be stored in (pushed onto) the stack
- Pop
  - retrieves and removes a value from the stack

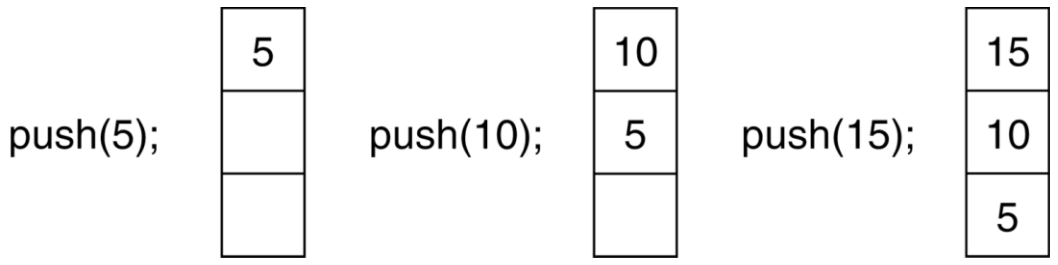
The Push Operation

- Suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations.

```
push(5);
push(10);
push(15);
```

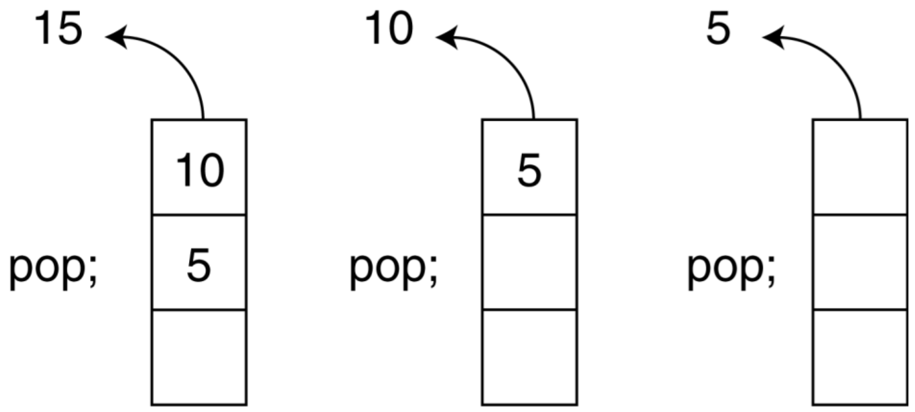
The Push Operation

The state of the stack after each of the push operations:



The Pop Operation

- Now, suppose we execute three consecutive pop operations on the same stack:



Other Stack Operations

- `isFull`: A Boolean operation needed for static stacks. Returns true if the stack is full. Otherwise, returns false.
- `isEmpty`: A Boolean operation needed for all stacks. Returns true if the stack is empty. Otherwise, returns false.

The `IntStack` Class

- **Member Variables**

Member Variable	Description
`*stackArray`	A pointer to int. When the constructor is executed, it uses stackArray to dynamically allocate an array for storage.
`stackSize`	An integer that holds the size of the stack.
`top`	An integer that is used to mark the top of the stack.

The IntStack Class

- **Member Functions**

Member Function	Description
IntStack	The class constructor accepts an integer argument, which specifies the size of the stack. An integer array of this size is dynamically allocated, and assigned to stackArray. Also, the variable top is initialized to –1.
push	The push function accepts an integer argument, which is pushed onto the top of the stack.
pop	The pop function uses an integer reference parameter. The value at the top of the stack is removed, and copied into the reference parameter.

The IntStack Class

- **Member Functions (continued)**

Member Function	Description
isFull	Returns true if the stack is full and false otherwise. The stack is full when top is equal to stackSize – 1.
isEmpty	Returns true if the stack is empty, and false otherwise. The stack is empty when top is set to –1.

Contents of `IntStack.h`

```
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
private:
       int *stackArray;
       int stackSize;
       int top;

public:
       IntStack(int);
       void push(int);
       void pop(int &);
       bool isFull(void);
       bool isEmpty(void);
};

#endif
```

Contents of `IntStack.cpp`

```
#include <iostream.h>
#include "intstack.h‟

//*******************
// Constructor      *
//*******************

IntStack::IntStack(int size)
{
       stackArray = new int[size];
       stackSize = size;
       top = -1;
}
```

Contents of `IntStack.cpp`

```
//*******************************************************
// Member function push pushes the argument onto   *
// the stack.                                       *
//*******************************************************

void IntStack::push(int num)
{
       if (isFull())
       {
              cout << "The stack is full.\n";
       }
       else
       {
              top++;
              stackArray[top] = num;
       }
}
```

Contents of `IntStack.cpp`

```
//*******************************************************
// Member function pop pops the value at the top       *
// of the stack off, and copies it into the variable *
// passed as an argument.                              *
//*******************************************************

void IntStack::pop(int &num)
{
       if (isEmpty())
       {
              cout << "The stack is empty.\n";
       }
       else
       {
              num = stackArray[top];
              top--;
       }
}
```

Contents of `IntStack.cpp`

```
//*****************************************************
// Member function isFull returns true if the stack *
// is full, or false otherwise.                     *
//*****************************************************

bool IntStack::isFull(void)
{
	bool status;

	if (top == stackSize - 1)
		status = true;
	else
		status = false;


	return status;
}
```

Contents of `IntStack.cpp`

```
//****************************************************
// Member funciton isEmpty returns true if the stack *
// is empty, or false otherwise.                     *
//****************************************************

bool IntStack::isEmpty(void)

{

      bool status;

      if (top == -1)
            status = true;
      else
            status = false;

      return status;
}
```

Program

```
// This program demonstrates the IntStack class.
#include <iostream.h>
#include "intstack.h“

void main(void)
{
	IntStack stack(5);
	int catchVar;

	cout << "Pushing 5\n";
	stack.push(5);
	cout << "Pushing 10\n";
	stack.push(10);
	cout << "Pushing 15\n";
	stack.push(15);
	cout << "Pushing 20\n";
	stack.push(20);
	cout << "Pushing 25\n";
	stack.push(25);
```

Program 1(continued)

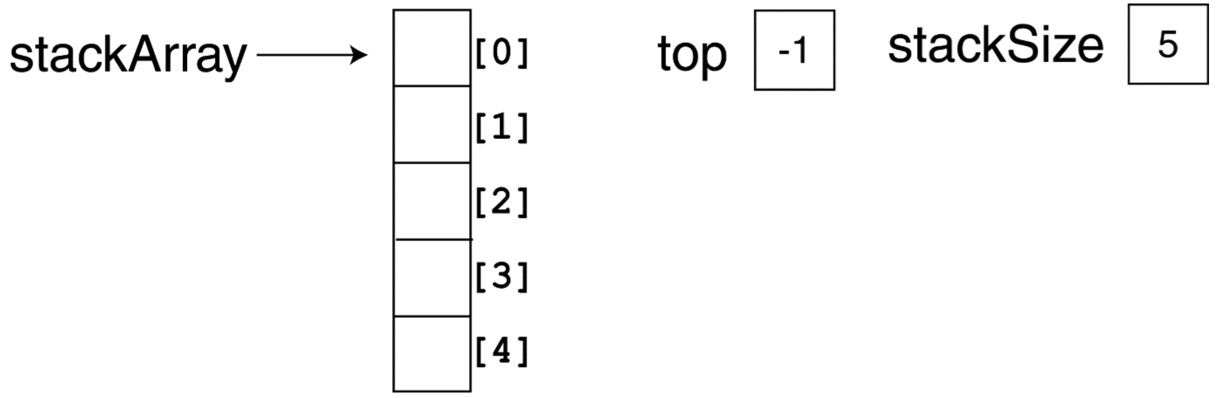
```
        cout << "Popping...\n";
        stack.pop(catchVar);
        cout << catchVar << endl;
        stack.pop(catchVar);
        cout << catchVar << endl;
        stack.pop(catchVar);
        cout << catchVar << endl;
        stack.pop(catchVar);
        cout << catchVar << endl;

        stack.pop(catchVar);
        cout << catchVar << endl;
}
```

Program 1(continued)

```
Program Output
Pushing 5
Pushing 10
Pushing 15
Pushing 20
Pushing 25
Popping...
25
20
15
10
5
```

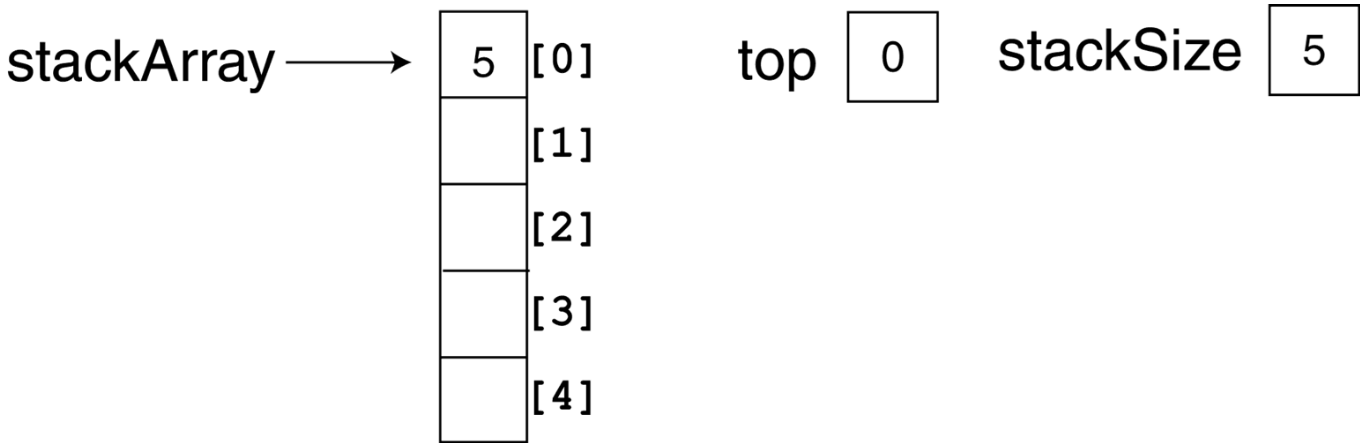
About Program

- In the program, the constructor is called with the argument 5. This sets up the member variables as shown in Figure. Since `top` is set to –1, the stack is empty



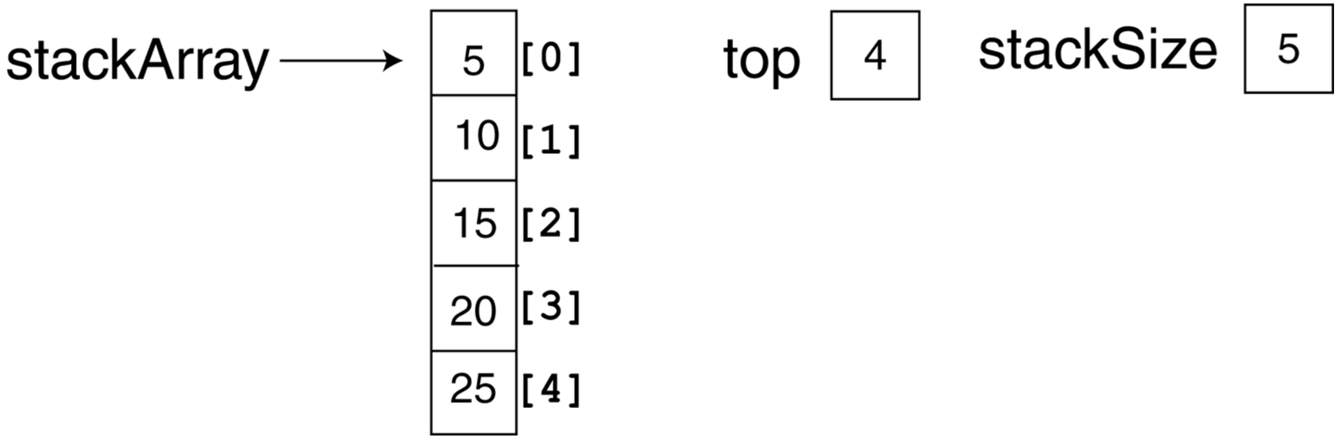
About Program 1

- Figure shows the state of the member variables after the `push` function is called the first time (with 5 as its argument). The top of the stack is now at element 0.



About Program 1

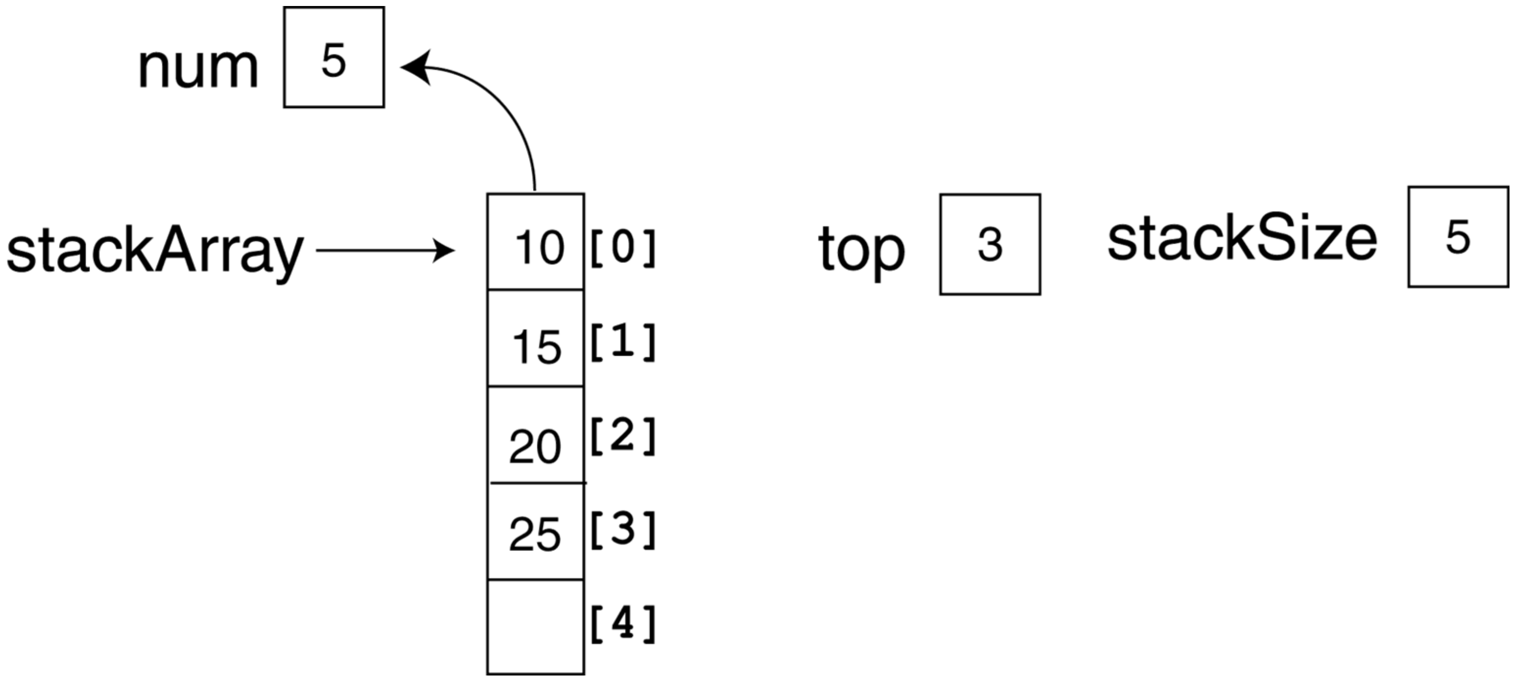
- Figure shows the state of the member variables after all five calls to the `push` function. Now the top of the stack is at element 4, and the stack is full.



About Program 1

- Notice that the `pop` function uses a reference parameter, `num`. The value that is popped off the stack is copied into `num` so it can be used later in the program. Figure (on the next slide) depicts the state of the class members, and the `num` parameter, just after the first value is popped off the stack.

About Program 1



Dynamic Stacks

- A dynamic stack is built on a linked list instead of an array.
- A linked list-based stack offers two advantages over an array-based stack.
  - No need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, and then expands by one node each time a value is pushed.
  - A dynamic stack will never be full, as long as the system has enough free memory.

Contents of `DynIntStack.h`

```
class DynIntStack
{
private:
        struct StackNode
        {
                int value;
                StackNode *next;
        };


        StackNode *top;

public:
        DynIntStack(void)
                {       top = NULL; }
        void push(int);
        void pop(int &);
        bool isEmpty(void);
};
```

Contents of DynIntStack.cpp

```
#include <iostream.h>
#include "dynintstack.h‟

//*******************************************************
// Member function push pushes the argument onto *
// the stack.                                      *
//*******************************************************

void DynIntStack::push(int num)
{
	StackNode *newNode;

	// Allocate a new node & store Num
	newNode = new StackNode;
	newNode->value = num;
```

Contents of `DynIntStack.cpp`

```
        // If there are no nodes in the list
        // make newNode the first node
        if (isEmpty())
        {
                top = newNode;
                newNode->next = NULL;
        }
        else    // Otherwise, insert NewNode before top
        {
                newNode->next = top;
                top = newNode;
        }
}


//*****************************************************
// Member function pop pops the value at the top     *
// of the stack off, and copies it into the variable *
// passed as an argument.                            *
//*****************************************************
```

Contents of `DynIntStack.cpp`

```
void DynIntStack::pop(int &num)
{
        StackNode *temp;

        if (isEmpty())
        {
                cout << "The stack is empty.\n";
        }
        else    // pop value off top of stack
        {
                num = top->value;
                temp = top->next;
                delete top;
                top = temp;
        }
}
```

Contents of `DynIntStack.cpp`

```
//*****************************************************
// Member funciton isEmpty returns true if the stack *
// is empty, or false otherwise.                     *
//*****************************************************

bool DynIntStack::isEmpty(void)
{
	bool status;

	if (!top)
		status = true;
	else
		status = false;

	return status;
}
```

Program 2

```
// This program demonstrates the dynamic stack
// class DynIntClass.

#include <iostream.h>
#include "dynintstack.h‟

void main(void)
{
       DynIntStack stack;
       int catchVar;


       cout << "Pushing 5\n";
       stack.push(5);
       cout << "Pushing 10\n";
       stack.push(10);
       cout << "Pushing 15\n";
       stack.push(15);
```

Program 2 (continued)

```
        cout << "Popping...\n";
        stack.pop(catchVar);
        cout << catchVar << endl;
        stack.pop(catchVar);
        cout << catchVar << endl;
        stack.pop(catchVar);
        cout << catchVar << endl;

        cout << "\nAttempting to pop again... ";
        stack.pop(catchVar);
}
```

Program Output

```
Pushing 5
Pushing 10
Pushing 15
Popping...
15
10
5

Attempting to pop again... The stack is empty.
```

The STL `stack` Container

- The STL `stack` container may be implemented as a `vector`, a `list`, or a `deque` (which you will learn later in this course).
- Because the `stack` container is used to adapt these other containers, it is often referred to as a *container adapter*.

The STL `stack` Container

- Here are examples of how to declare a stack of `int`s, implemented as a `vector`, a `list`, and a `deque`.

```
stack< int, vector<int> > iStack;  // Vector stack

stack< int, list<int> > iStack;    // List stack

stack< int > iStack;       // Default – deque stack
```

Program 3

```
// This program demonstrates the STL stack
// container adapter.

#include <iostream.h>
#include <vector>
#include <stack>
using namespace std;

void main(void)
{
        int x;
        stack< int, vector<int> > iStack;

        for (x = 2; x < 8; x += 2)
        {
                cout << "Pushing " << x << endl;
                iStack.push(x);
        }
```

Program 3 (continued)

```
        cout << "The size of the stack is ";
        cout << iStack.size() << endl;

        for (x = 2; x < 8; x += 2)
        {
                cout << "Popping " << iStack.top() << endl;
                iStack.pop();
        }
}
```

Program Output

```
Pushing 2
Pushing 4
Pushing 6
The size of the stack is 3
Popping 6
Popping 4
Popping 2
```

Stack: Array or List

- Since both implementations support stack operations in constant time, any reason to choose one over the other?
- Allocating and deallocating memory for list nodes does take more time than preallocated array.
- List uses only as much memory as required by the nodes; array requires allocation ahead of time.
- List pointers (head, next) require extra memory.
- Array has an upper limit; List is limited by dynamic memory allocation.

Use of Stack

- Example of use: prefix, infix, postfix expressions.
- Consider the expression A+B: we think of applying the *operator* “+” to the *operands* A and B.
- “+” is termed a *binary operator*: it takes two operands.
- Writing the sum as A+B is called the *infix* form of the expression.

Prefix, Infix, Postfix

- Two other ways of writing the expression are

  + A B *prefix*
  A B + *postfix*

- The prefixes "pre" and "post" refer to the position of the operator with respect to the two operands.

Prefix, Infix, Postfix

- Consider the infix expression

  A + B * C
- We "know" that multiplication is done before addition.
- The expression is interpreted as

  A + (B * C)
- Multiplication has *precedence* over addition.

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition
A B C * +	postfix form

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition
(A B +) C *	convert multiplication
A B + C *	postfix form

Precedence of Operators

- The five binary operators are: addition, subtraction, multiplication, division and exponentiation.
- The order of precedence is (highest to lowest)
- Exponentiation ↑
- Multiplication/division *, /
- Addition/subtraction +, -

Precedence of Operators

- For operators of same precedence, the left-to-right rule applies:

  A+B+C means (A+B)+C.

- For exponentiation, the right-to-left rule applies

  $A \uparrow B \uparrow C$ means $A \uparrow (B \uparrow C)$

Infix to Postfix

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

Infix to Postfix

- Note that the postfix form an expression does not require parenthesis.
- Consider ‘4+3*5’ and ‘(4+3)*5’. The parenthesis are not needed in the first but they are necessary in the second.
- The postfix forms are:

4+3*5	435*+
(4+3)*5	43+5*

Evaluating Postfix

- Each operator in a postfix expression refers to the previous two operands.
- Each time we read an operand, we push it on a stack.
- When we reach an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack.

Evaluating Postfix

```
Stack s;
while( not end of input ) {
    e = get next element of input
    if( e is an operand )
        s.push( e );
    else {
        op2 = s.pop();
        op1 = s.pop();
        value = result of applying operator ‘e’ to op1 and op2;
        s.push( value );
    }
}
finalresult = s.pop();
```

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 $\uparrow$ 3 +

Input	**op1**	**op2**	**value**	**stack**
6				**6**
2				**6,2**
3				**6,2,3**
+	**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Converting Infix to Postfix

- Consider the infix expressions ‘A+B*C’ and ‘ (A+B)*C’.
- The postfix versions are ‘ABC*+’ and ‘AB+C*’.
- The order of operands in postfix is the same as the infix.
- In scanning from left to right, the operand ‘A’ can be inserted into postfix expression.

Converting Infix to Postfix

- The ‘+’ cannot be inserted until its second operand has been scanned and inserted.
- The ‘+’ has to be stored away until its proper position is found.
- When ‘B’ is seen, it is immediately inserted into the postfix expression.
- Can the ‘+’ be inserted now? In the case of ‘A+B*C’ cannot because * has precedence.

Converting Infix to Postfix

- In case of ‘(A+B)*C’, the closing parenthesis indicates that ‘+’ must be performed first.
- Assume the existence of a function ‘prcd(op1,op2)’ where op1 and op2 are two operators.
- Prcd(op1,op2) returns TRUE if op1 has precedence over op2, FASLE otherwise.

Converting Infix to Postfix

- prcd(‘*’,’+’) is TRUE
- prcd(‘+’,’+’) is TRUE
- prcd(‘+’,’*’) is FALSE
- Here is the algorithm that converts infix expression to its postfix form.
- The infix expression is without parenthesis.

Converting Infix to Postfix

```
Stack s;
While( not end of input ) {
    c = next input character;
    if( c is an operand )
        add c to postfix string;
    else {
        while( !s.empty() && prcd(s.top(),c) ){
             op = s.pop();
             add op to the postfix string;
         }
         s.push( c );
    }
    while( !s.empty() ) {
        op = s.pop();
        add op to postfix string;
    }
```

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC *	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC *	+
	ABC * +	

Converting Infix to Postfix

- Handling parenthesis
- When an open parenthesis ‘(‘ is read, it must be pushed on the stack.
- This can be done by setting prcd(op,‘(‘) to be FALSE.
- Also, prcd(‘(‘,op) == FALSE which ensures that an operator after ‘(‘ is pushed on the stack.

Converting Infix to Postfix

- When a ‘)’ is read, all operators up to the first ‘(‘ must be popped and placed in the postfix string.
- To do this, prcd(op,’)’) == TRUE.
- Both the ‘(‘ and the ‘)’ must be discarded: prcd(‘(‘,’)’) == FALSE.
- Need to change line 11 of the algorithm.

Converting Infix to Postfix

```
if( s.empty()  ||  symb != ‘)’ )
      s.push( c );
else
      s.pop(); // discard the ‘(‘
```

```
prcd( ‘(‘, op ) = FALSE   for any operator
prcd( op, ‘(’ ) = FALSE   for any operator
                             other than ‘(’
prcd( op, ‘)’ ) = TRUE     for any operator
                           other than ‘(‘
prcd( ‘)’, op ) = error      for any operator.
```