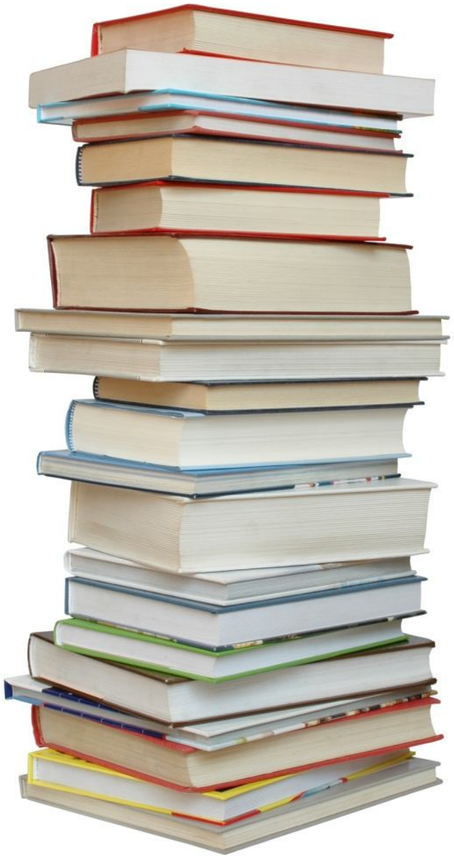
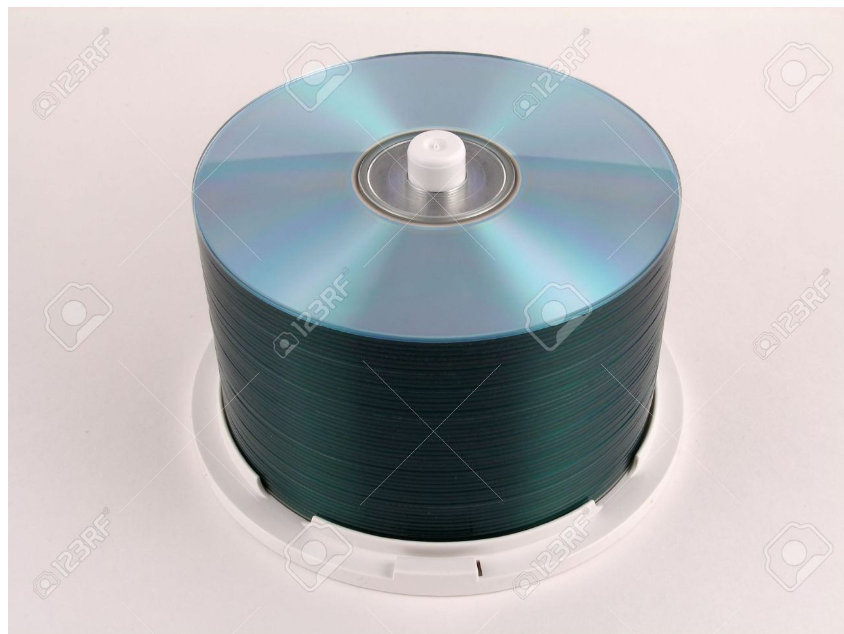


Stack



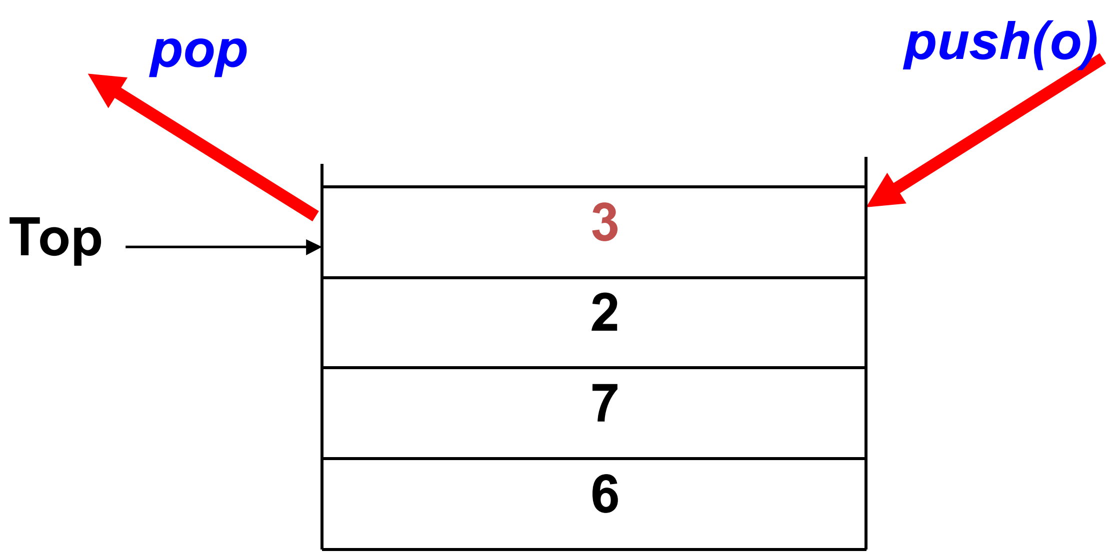
Stack Overview

- Stack ADT
- Basic operations of stack
  - Pushing, popping etc.
- Implementations of stacks using
  - array
  - linked list

What is a Stack?

- A *stack* is a list ADT with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the ***top***.

- The operations: **push (insert)** and **pop (delete)**



The Stack ADT

- **Fundamental operations:**
  - Push: Equivalent to an insert
  - Pop: Deletes the most recently inserted element
  - Top: Examines the most recently inserted element

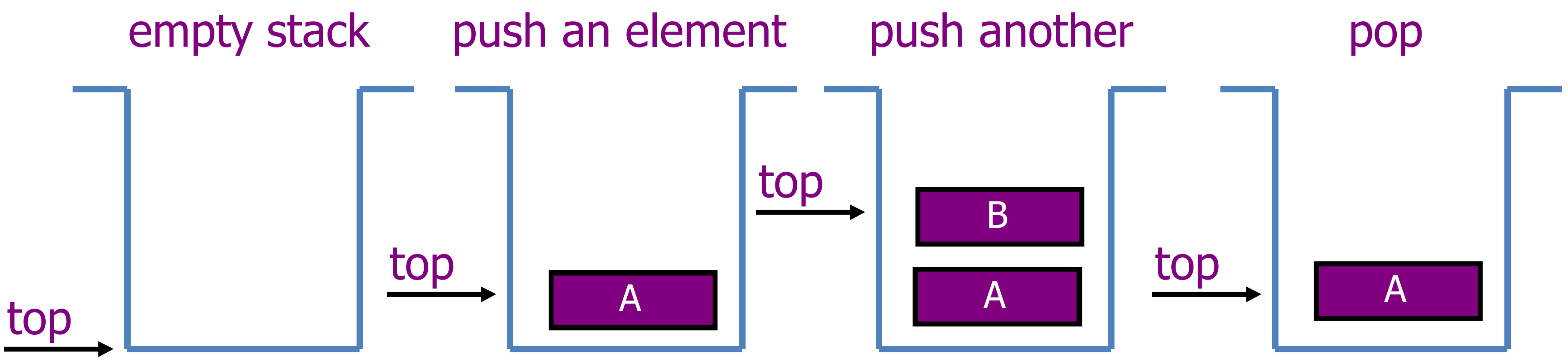
Stack ADT

- Stacks are less flexible
  - ✓ but are more efficient and easy to implement

- Stacks are known as LIFO (Last In, First Out) lists.
  - The last element inserted will be the first to be retrieved

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



Implementation of Stacks

- Any list implementation could be used to implement a stack
  - Arrays (static: the size of stack is given initially)
  - Linked lists (dynamic: never become full)
- We will explore implementations based on array and linked list
- Let's see how to use an array to implement a stack first

Array Implementation

- Need to declare an array size ahead of time
- Associated with each stack is TopOfStack
  - for an empty stack, set TopOfStack to -1

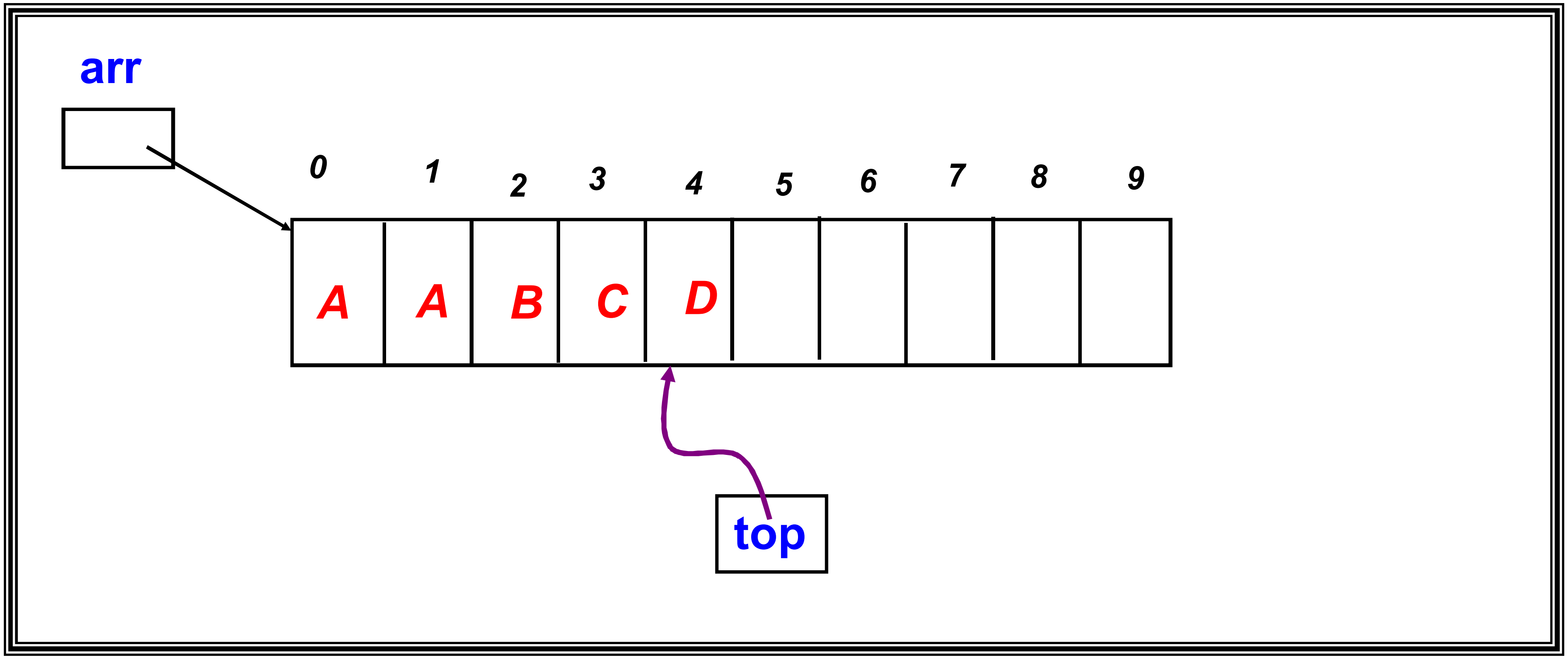
- Push
  - (1) Increment TopOfStack by 1.
  - (2) Set Stack[TopOfStack] = X

- Pop
  - (1) Set return value to Stack[TopOfStack]
  - (2) Decrement TopOfStack by 1

- These operations are performed in very fast constant time

Implementation by Array

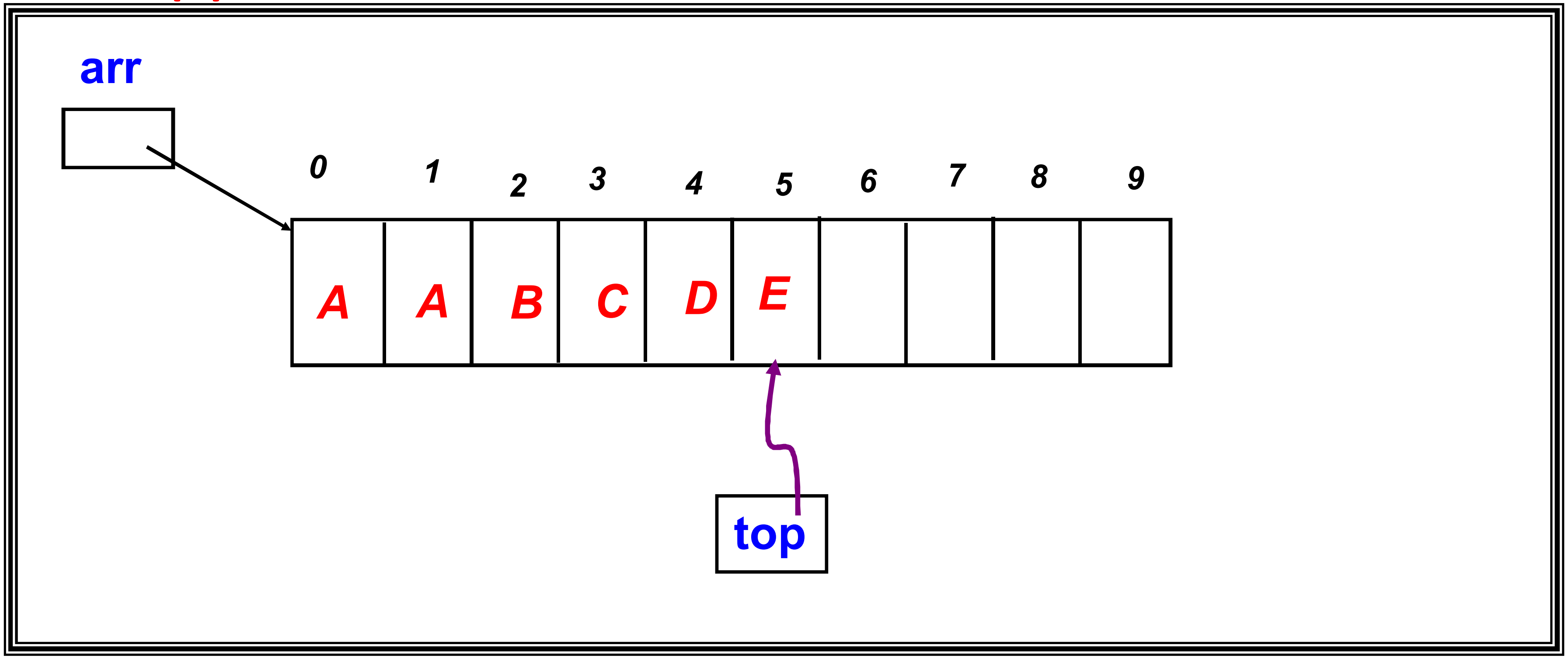
- use Array with a top index pointer as an implementation of stack



Implementation by Array

- use Array with a top index pointer as an implementation of stack

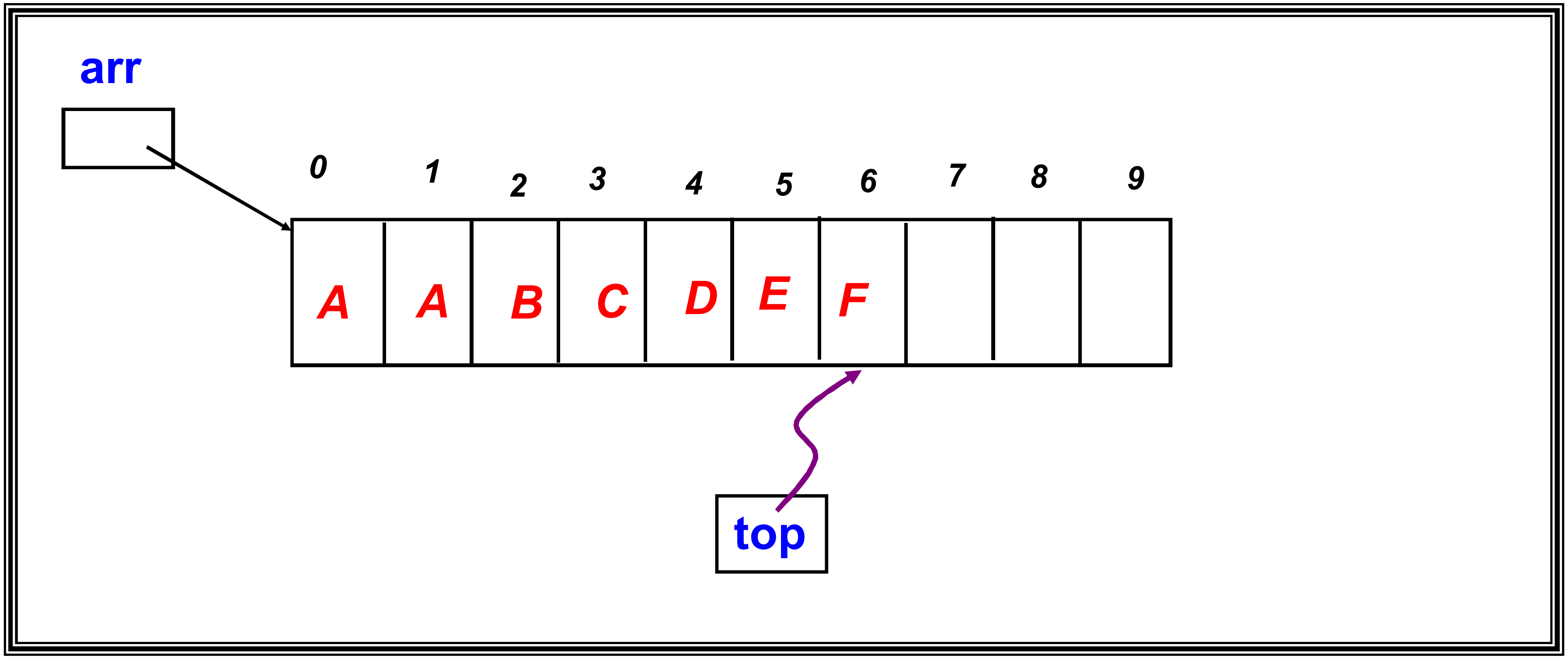
Push(E)



Implementation by Array

- use Array with a top index pointer as an implementation of stack

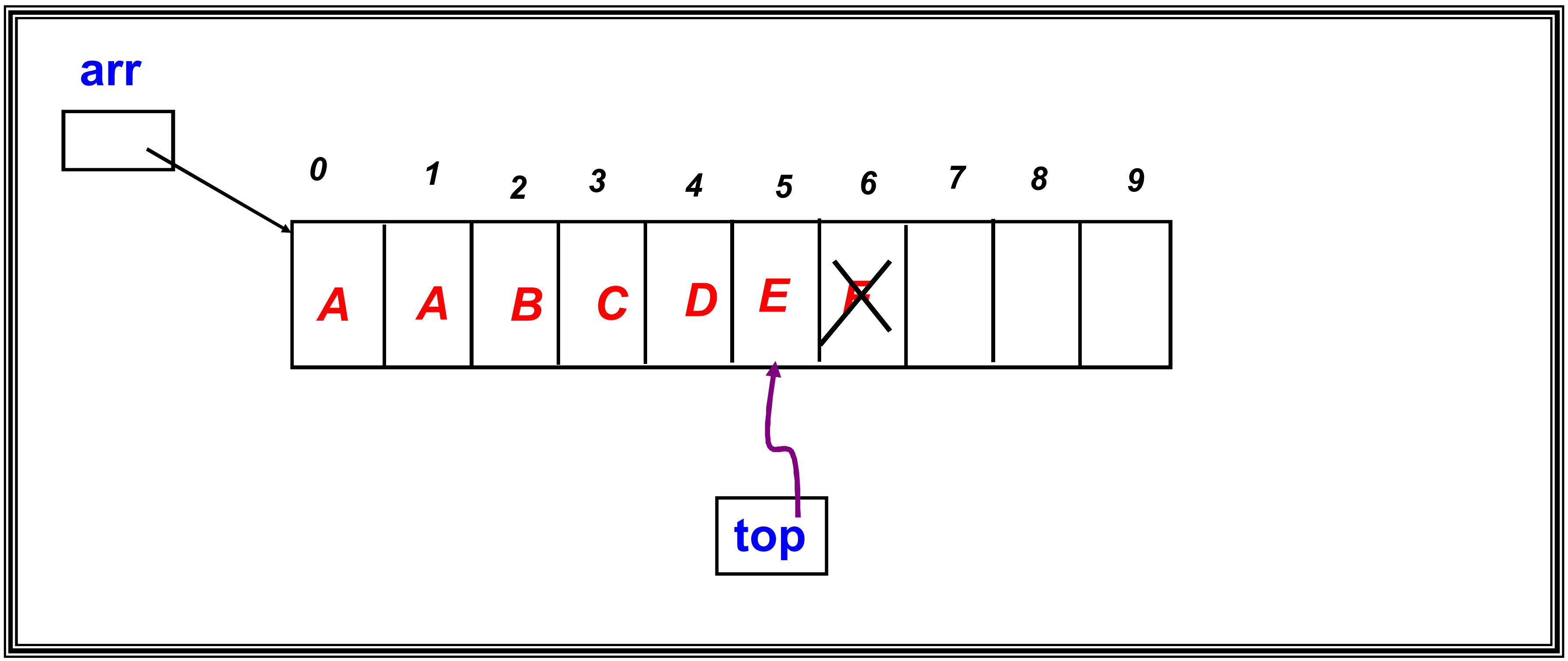
Push(F)



Implementation by Array

- use Array with a top index pointer as an implementation of stack

Pop()



`Stack` class

```
class Stack {
public:
      Stack(int size);                   // constructor
      ~Stack() { delete [] values; }     // destructor
      bool IsEmpty() { return top == -1; }
      bool IsFull() { return top == maxTop; }
      double Top();
      void Push(double x);
      double Pop();
      void DisplayStack();
private:
      int maxTop          // max stack size = size - 1
      int top;            // current top of stack
      double* values;     // element array
}
```

`Stack` class

- Attributes of `Stack`
  - `maxTop`: the max size of stack
  - `top`: the index of the top element of stack
  - `values`: point to an array which stores elements of stack
- Operations of `Stack`
  - `IsEmpty`: return true if stack is empty, return false otherwise
  - `IsFull`: return true if stack is full, return false otherwise
  - `Top`: return the element at the top of stack
  - `Push`: add an element to the top of stack
  - `Pop`: delete the element at the top of stack
  - `DisplayStack`: print all the data in the stack

Create Stack

- The constructor of `Stack`
  - Allocate a stack array of `size`.
  - When the stack is full, `top` will have its maximum value, i.e. `size - 1`.
  - Initially `top` is set to -1. It means the stack is empty.

```
Stack::Stack(int size) {
      maxTop      =     size - 1;
      values      =     new double[size];
      top         =     -1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed after the initialization.

Push Stack

- `void Push(double x);`
  - Push an element onto the stack
  - If the stack is full, print the error information.
  - Note `top` always represents the index of the top element. After pushing an element, increment `top`.

```
void Stack::Push(double x) {
     if (IsFull())
          cout << "Error: the stack is full." << endl;
     else
          values[++top]=x;
}
```

Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don’t forgot to decrement `top`

```
double Stack::Pop() {
      if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else {
            return values[top--];
      }
}
```

Stack Top

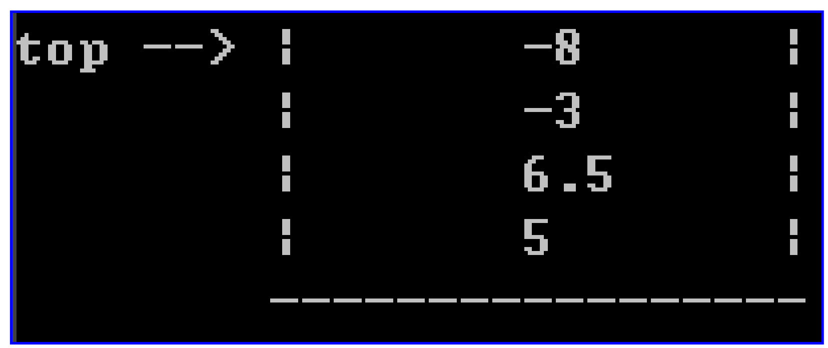
- `double Top()`
  - Return the top element of the stack
  - Unlike `Pop`, this function does not remove the top element

```
double Stack::Top() {
	if (IsEmpty()) {
		cout << "Error: the stack is empty." << endl;
		return -1;
	}
	else
		return values[top];
}
```

Printing all the elements

- `void DisplayStack()`
  - Print all the elements

```
void Stack::DisplayStack() {
      cout << "top -->";
      for (int i = top; i >= 0; i--)
            cout << "\t|\t" << values[i] << "\t|" << endl;
      cout << "\t|----------------|" << endl;
}
```



Using Stack

result

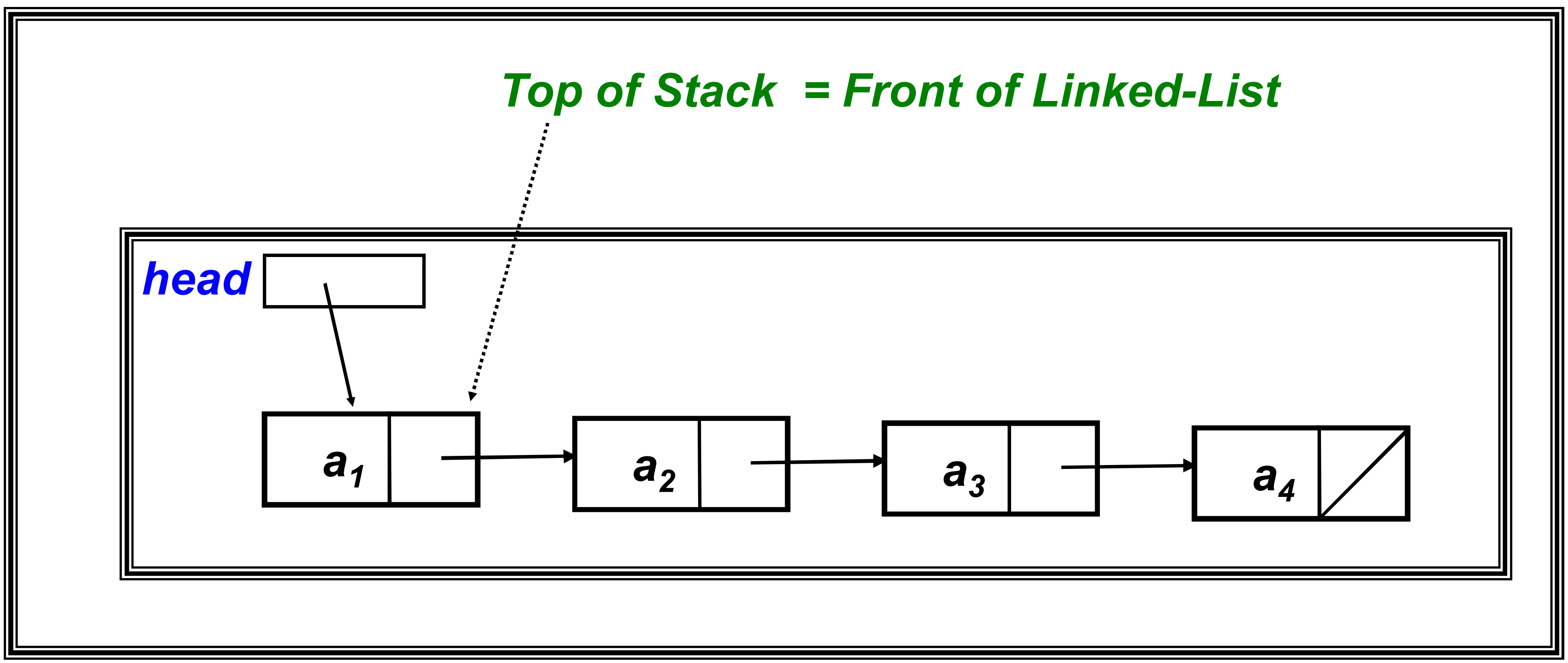
```
int main(void) {
       Stack stack(5);
       stack.Push(5.0);
       stack.Push(6.5);
       stack.Push(-3.0);
       stack.Push(-8.0);
       stack.DisplayStack();
       cout << "Top: " << stack.Top() << endl;

       stack.Pop();
       cout << "Top: " << stack.Top() << endl;
       while (!stack.IsEmpty()) stack.Pop();
       stack.DisplayStack();
       return 0;
}
```

```
top --> |       -8       |
        |       -3       |
        |       6.5      |
        |       5        |
        |----------------|
Top: -8
Top: -3
top --> |----------------|
```

Implementation based on Linked List

- Now let us implement a stack based on a linked list



Stack class

```
struct Node {
 double data;       //data
 Node* next;        //pointer to next node
};

class Stack {
public:
      Stack() {top=NULL; size=0;}          // constructor
      ~Stack();                            // destructor
      bool IsEmpty() { return top==NULL; }
      double Top();
      void Push(double x);
      double Pop();
      void DisplayStack();
private:
      Node* top;
      int size;
}
```

Push Stack

```
void Stack::Push(double x) {
  node *newTop = new node;
  newTop->data = x;
  if(top == NULL){
   newTop->next = NULL;
  }
  else {
   newTop->next = top;
  }
  top = newTop; size++;

}
```

Pop Stack

```
double Stack::Pop() {
  if(IsEmpty()) {
   cout<<"Error: Stack is empty"<<endl;
   return(-1);
  }
  else {
   double val=top->data;
   node* old=top;
   top=top->next;
   size--;
   delete old;
   return(val);
  }
}
```

Stack Top

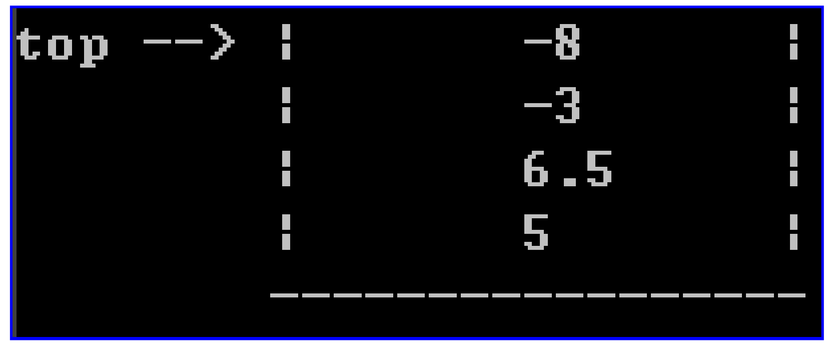
```
double Stack::Top() {
      if(IsEmpty()) {
            cout<<"Error: Stack is empty"<<endl;
            return(-1);
  }
      else
            return top->data;
}
```

Destructor

```
Stack::~Stack(void) {
   Node* currNode = top;
   Node* nextNode = NULL;
   while (currNode != NULL)
   {
     nextNode  =  currNode->next;
     // destroy the current node
     delete currNode;
     currNode  =  nextNode;
   }
}
```

Printing all the elements

- `void DisplayStack()`
  - Print all the elements




```
void Stack::DisplayStack() {
  cout << "top -->";
  Node* currNode  =     top;
    while (currNode != NULL){
      cout << “\t|\t” << currNode->data << “\t|”<<endl;
      currNode    =     currNode->next;
    }
      cout << "\t|---------------|" << endl;
}
```

Using `Stack (linked list)`

```
int main(void) {
       Stack *s = new Stack;
       s->Push(5.0);
       s->Push(6.5);
       s->Push(-3.0);
       s->Push(-8.0);
       s->DisplayStack();
       cout << "Top: " << s->Top() << endl;
       s->Pop();
       cout << "Top: " << s->Top() << endl;
       while (!s->IsEmpty()) s->Pop();
       s->DisplayStack();
       return 0;
}
```

result

```
top --> |       -8        |
        |       -3        |
        |       6.5       |
        |       5         |
        |-----------------|
Top: -8
Top: -3
top --> |-----------------|
```

Applications

- Many application areas use stacks:
  - line editing
  - bracket matching
  - postfix calculation
  - function call stack

Line Editing

- A line editor would place characters read into a buffer but may use a backspace symbol (denoted by ←) to do error correction
- *Refined Task*
  - read in a line
  - correct the errors via backspace
  - print the corrected line in reverse

Input : abc_defg~~h~~←2klp~~q~~~~r~~←←wxyz

Corrected Input : abc_defg2klpwxyz

Reversed Output : zyxwplk2gfed_cba

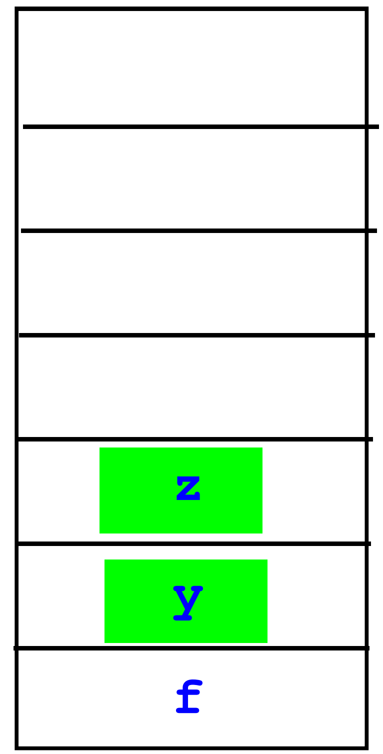
The Procedure

- Initialize a new stack
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output

Input : `fgh←r←←yz`

Corrected Input : `fyz`

Reversed Output : `zyf`



Stack

Bracket Matching Problem

- Ensures that pairs of brackets are properly matched

  - **An Example:** `{a,(b+f[4])*3,d+f[5]}`

  - **Bad Examples:**

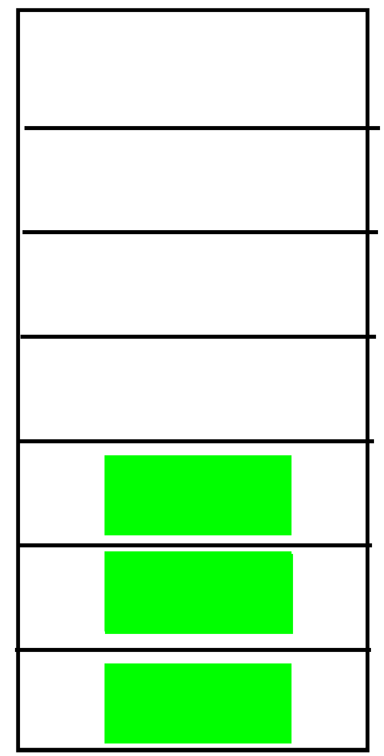
- `(..)..)` `// too many closing brackets`
- `(..(..)` `// too many open brackets`
- `[..(..]..)` `// mismatched brackets`

Informal Procedure

Initialize the stack to empty
For every char read
    if open bracket then *push onto stack*
    if close bracket, then
        return & remove most recent item
            from *the stack*
        if doesn't match then *flag error*
    if non-bracket, *skip the char read*

Example

`{a,(b+f[4])*3,d+f[5]}`



Stack

Prefix, Postfix, Infix Notation

Infix Notation

- To add A, B, we write

  A+B

- To multiply A, B, we write

  A*B

- The operators ('+' and '*') go in between the operands ('A' and 'B')
- This is *"Infix"* notation.

Prefix/Polish Notation

- Instead of saying "A plus B", we could say "add A,B " and write

  + A B

- "Multiply A,B" would be written

  * A B

- This is *Prefix* notation.

Postfix/Reverse Polish Notation

- Another alternative is to put the operators after the operands as in

A B +

and

A B *

- This is *Postfix* notation.

- The terms infix, prefix, and postfix tell us whether the operators go between, before, or after the operands.

Parentheses

- Evaluate 2+3*5.
- + First:

  (2+3)*5 = 5*5 = 25
- * First:

  2+(3*5) = 2+15 = 17
- Infix notation requires Parentheses.

What about Prefix Notation?

- + 2 * 3 5 =

  $= + \; 2 \; \underline{* \; 3 \; 5}$

  $= \underline{+ \; 2 \; 15} = 17$

- * + 2 3 5 =

  $= * \; \underline{+ \; 2 \; 3} \; 5$

  $= \underline{* \; 5 \; 5} = 25$

- No parentheses needed!

Postfix Notation

- 2 3 5 * + =

  = 2 <u>3 5 *</u> +

  = <u>2 15 +</u> = 17

- 2 3 + 5 * =

  = <u>2 3 +</u> 5 *

  = <u>5 5 *</u> = 25

- No parentheses needed here either!

Conclusion:

- Infix is the only notation that requires parentheses in order to change the order in which the operations are done.

Fully Parenthesized Expression

- A FPE has exactly one set of Parentheses enclosing each operator and its operands.
- Which is fully parenthesized?

(A + B) * C

((A + B) * C)

((A + B) * (C))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

((A + B) * (C + D))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

(+ A B * (C + D))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

* + A B (C + D)

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

* + A B + C D

Order of operands does not change!

Infix to Postfix

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

A B + C * D E + F / -

- Operand order does not change!
- Operators are in order of evaluation!

Computer Algorithm
FPE Infix To Postfix

- Assumptions:

1. Space delimited list of tokens represents a FPE infix expression
2. Operands are single characters.
3. Operators +,-,*,/

FPE Infix To Postfix

- Initialize a Stack for operators, output list
- Split the input into a list of tokens.
- for each token (left to right):

    if it is operand: append to output

    if it is operator or '(': push onto Stack

    if it is ')': pop operators & append till '('

FPE Infix to Postfix

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

- stack: <empty>
- output: []

FPE Infix to Postfix

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

- stack: (
- output: []

FPE Infix to Postfix

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

- stack: ((
- output: []

FPE Infix to Postfix

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

- stack: (((
- output: []

FPE Infix to Postfix

+ B) * (C - E)) / (F + G))

- stack: (((
- output: [A]

FPE Infix to Postfix

B) * (C - E)) / (F + G))

- stack: (((+
- output: [A]

FPE Infix to Postfix

) * (C - E)) / (F + G))

- stack: (((+
- output: [A B]

FPE Infix to Postfix

* (C - E)) / (F + G))

- stack: ((
- output: [A B +]

FPE Infix to Postfix

(C - E)) / (F + G))

- stack: ((*
- output: [A B +]

FPE Infix to Postfix

C - E)) / (F + G))

- stack: ((* (
- output: [A B +]

FPE Infix to Postfix

- E)) / (F + G))

- stack: ((* (
- output: [A B + C]

FPE Infix to Postfix

E)) / (F + G))

- stack: ((* (-
- output: [A B + C]

FPE Infix to Postfix

)) / (F + G))

- stack: ((* (-
- output: [A B + C E]

FPE Infix to Postfix

) / (F + G))

- stack: ((*
- output: [A B + C E -]

FPE Infix to Postfix

/ (F + G))

- stack: (
- output: [A B + C E - *]

FPE Infix to Postfix

(F + G))

- stack: (/
- output: [A B + C E - *]

FPE Infix to Postfix

F + G))

- stack: (/ (
- output: [A B + C E - *]

FPE Infix to Postfix

+ G))

- stack: (/ (
- output: [A B + C E - * F]

FPE Infix to Postfix

G))

- stack: (/ (+
- output: [A B + C E - * F]

FPE Infix to Postfix

))

- stack: (/ (+
- output: [A B + C E - * F G]

FPE Infix to Postfix

)

- stack: (/
- output: [A B + C E - * F G +]

FPE Infix to Postfix

- stack: <empty>
- output: [A B + C E - * F G + /]

Problem with FPE

- Too many parentheses.
- Establish precedence rules:
- We can alter the previous program to use the precedence rules.

Infix to Postfix

- Initialize a Stack for operators, output list
- Split the input into a list of tokens.
- for each token (left to right):

    if it is operand: append to output

    if it is '(': push onto Stack

    if it is ')': pop & append till '('

    if it in '+-*/':

        while top has precedence ≥ it:

            pop & append the operator

        push onto Stack

  At end pop and append the rest of the operators in Stack.

Infix to Postfix

A * (B + C * D) + E

- stack: <empty>
- output: []

Infix to Postfix

* (B + C * D) + E

- stack: <empty>
- output: [A]

Infix to Postfix

(B + C * D) + E

- stack: <*>
- output: [A]

Infix to Postfix

B + C * D) + E

- stack: <*(>
- output: [A]

Infix to Postfix

+ C * D) + E

- stack: <*(>
- output: [AB]

Infix to Postfix

C * D) + E

- stack: <*(+>
- output: [AB]

Infix to Postfix

* D) + E

- stack: <*(+>
- output: [ABC]

Infix to Postfix

D) + E

- stack: <*(+*>
- output: [ABC]

Infix to Postfix

) + E

- stack: <*(+*>
- output: [ABCD]

Infix to Postfix

+ E

- stack: <*(+*)>
- output: [ABCD]

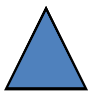
Infix to Postfix

+ E

- stack: <*>
- output: [ABCD*+]

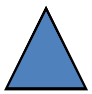
Infix to Postfix

E



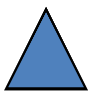
- stack: <+>
- output: [ABCD*+*]

Infix to Postfix



- stack: <+>
- output: [ABCD*+*E]

Infix to Postfix



- stack: <empty>
- output: [ABCD*+*E+]

Homework

- Convert infix to postfix using stack:

(A + B + C) ^ D * (E + F)/G

Precedence: ^

*,/

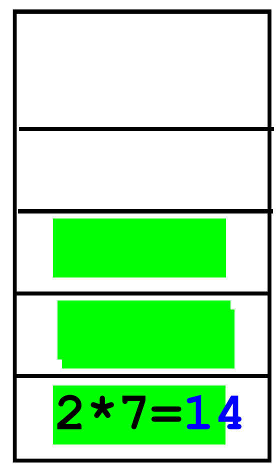
+,-

Evaluating Postfix Expression

Initialise stack S
For each item read.
  If it is an operand,
    push on the stack
  If it is an operator,
    pop arguments from stack;
    perform operation;
    push result onto the stack

2 3 4 + *

Expr	
2	push(2)
3	push(3)
4	push(4)
+	arg2=Pop()
	Arg1=Pop()
	push(arg1+arg2)
*	arg2=Pop()
	arg1=Pop()
	push(arg1*arg2)



Stack

Homework

- Evaluate the postfix expression using stack

4 5 + 2 + 2 ^ 3 4 + * 7 /