Stacks

ICT 4303

Contents

- Definition
- Operations on stacks
- Evaluation of Arithmetic Expressions
- Conversion of Arithmetic Expressions
- Recursion
- Multiple Stacks

Definition

- **Special case** of ordered list.
- Stack is an ordered group of homogeneous items of elements.
- Elements are added to and removed at one end called the top of the stack.
- The most recently added items are at the top of the stack.
- The last element to be added is the first to be removed.
- Stack is known as **LIFO (Last-In, First-Out) list**. Why?

Stack Definition

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).

A stack of books

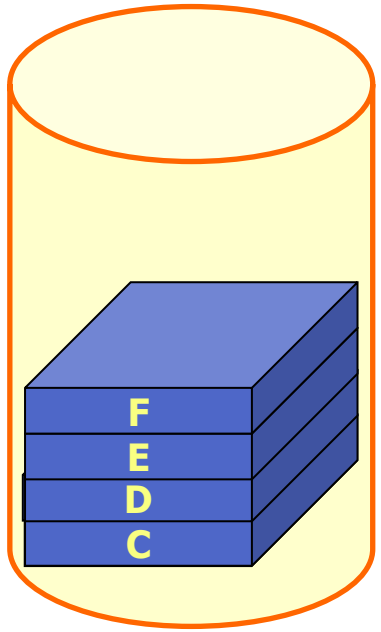
A pile of plates

A stack of dvd/cd

Stack Application

- Page visited history in a Web-browser
- Undo sequence in a text editor
- Program run-time environment

Pushing and Popping a stack



The Stack

Push box A onto the stack

Push box B onto the stack

Pop a box from the stack

Pop a box from the stack

Push box C onto the stack

Push box D onto the stack

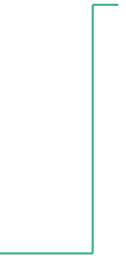
Push box E onto the stack

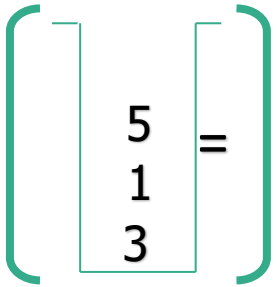
Push box F onto the stack

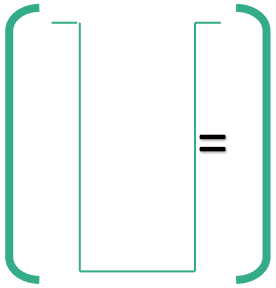
Stack as ADT

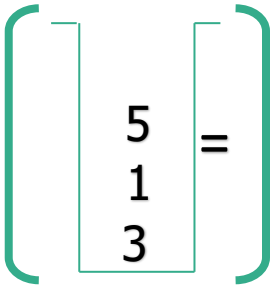
- Object: a finite ordered list with zero or more elements.
- Functions:
  - All operations takes place at a single end that is top of the stack and following operations can be performed:
  - *Stack* CreateS (): Create an empty stack whose maximum size is *maxStackSize.*
  - *Boolean* isFull(): Return true if the stack is full, otherwise return false.
  - *Stack* push(): Insert an element at one end of the stack called top.
  - *Boolean* isEmpty(): Return true if the stack is empty, otherwise return false.
  - *Element* pop(): Remove and return the element at the top of the stack, if it is not empty.
  - *Element* peek(): Return the element at the top of the stack without removing it, if the stack is not empty.
  - size(): Return the number of elements in the stack.

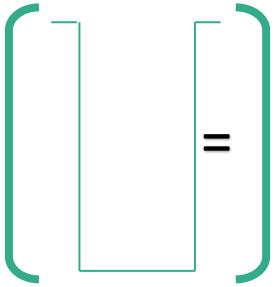
Basic Operations

Create() = 

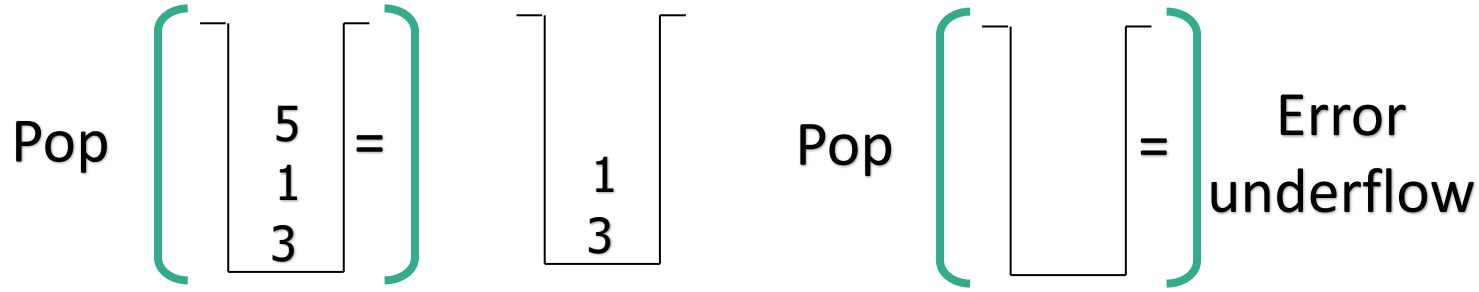
Empty  = false

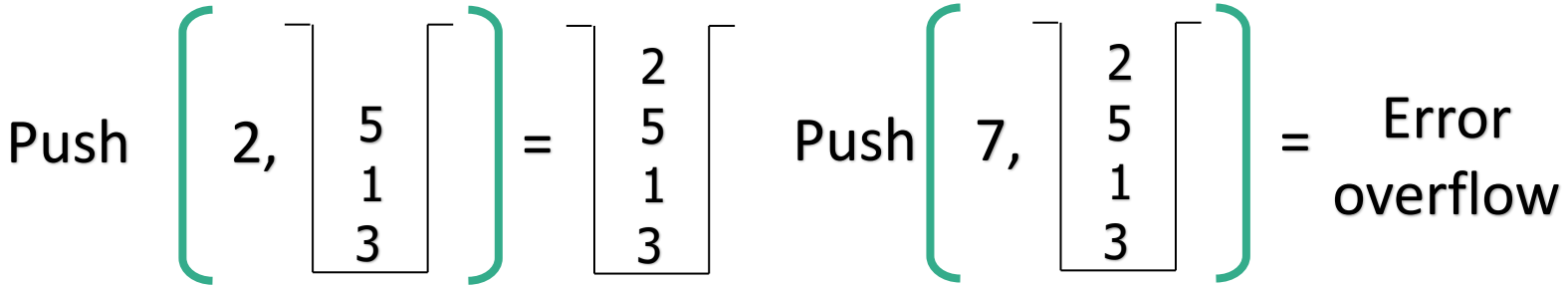
Empty  = true

Top  = 5

Top  = Error underflow

Basic Operations





Stack Class

```
#include <iostream>
using namespace std;
#define MAX 1000
class Stack {
        int top;
  public: int a[MAX]; // Maximum size of Stack
        Stack() { top = -1; }
        void push(int);
        int pop();
        int peek();
        void display();
        bool isEmpty();
};
```

Stack: Push ()

```
void Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
          }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
           }
}
```

Stack: Pop ()

```
int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
```

```
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}
```

Stack: isEmpty () and display ()

```
bool Stack::isEmpty()
{
    return (top < 0);
}
void Stack::display() {
   if(top>=0) {
            cout<<"Stack elements are:";
            for(int i=top; i>=0; i--)
                        cout<<a[i]<<" ";
            cout<<endl;
            } else
                        cout<<"Stack is empty";
}
```

Stack

```
int main()
{
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";
    s.display();
    cout<<"The element at the top of stack is"<<s.peek();
    return 0;
}
```

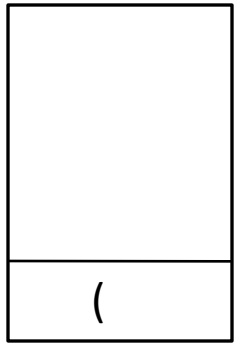
Checking for Palindrome

- Scan the string from left to right till the end and push every char you encounter during the scan to the stack.
- Rescan the string left to right till end and do the following for every char you encounter during scan.
  - Pop one char and compare current char with the popped one.
  - If no match report immediately non palindrome other wise proceed to the next char.

Algorithm for checking expression

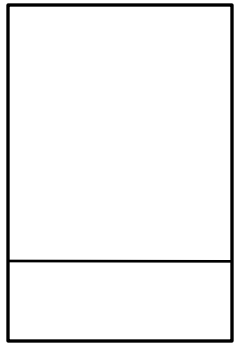
1. Scan the expression from left to right.
2. Whenever a scope opener (‘(‘, ’{‘, ’[‘) is encountered while scanning the expression, it is pushed to stack.
3. Whenever a scope ender (‘)’, ‘}’, ‘]’) is encountered, the stack is examined.
   - If stack is empty, scope ender does not have a matching opener and hence string is invalid.
   - If stack is non-empty, we pop the stack and check whether the popped item corresponds to scope ender.
   - If a match occurs, we continue. If it does not, the string is invalid.
4. When the end of string is reached, the stack must be empty; otherwise, 1 or more scopes have been opened which have not been closed and string is invalid.

Example 1:(a+b) * (c+d



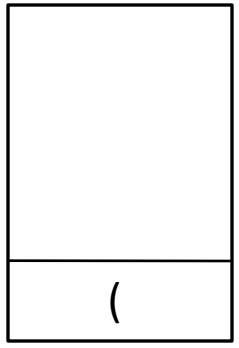
(

push ‘(‘



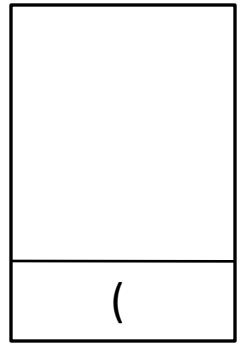
(a+b)

Pop ‘(‘ since there is ‘)’ & continue



(a+b) *(

Push ‘(‘

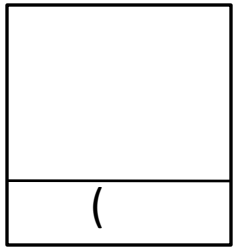


(a+b) *(c+d

End of string reached but stack not empty.

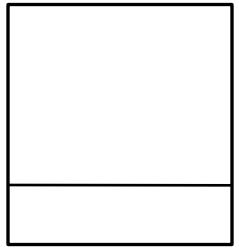
Hence, invalid

Example 2:(a+b)*c+d)



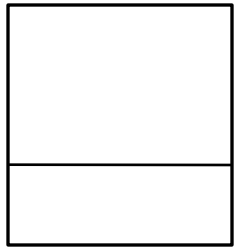
(

push ‘(‘



(a+b)

pop‘(‘ and continue

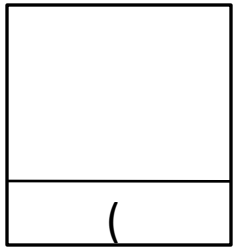


(a+b)*c+d)

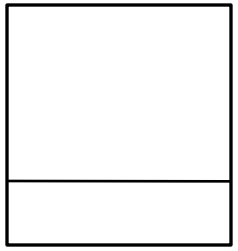
Stack empty when ‘)’ encountered.

Hence, invalid.

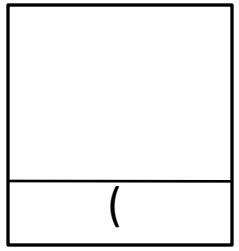
Example 3: (a+b)*({c*d)



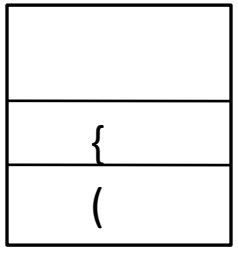
(
push ‘(‘



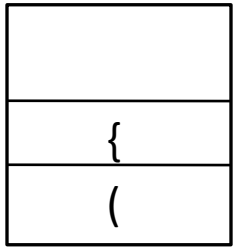
(a+b)
pop‘(‘ and
continue



(a+b)*(
push‘(‘ and
continue

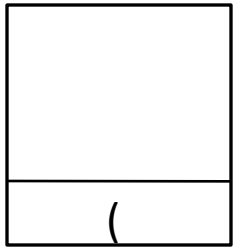


(a+b)*({
push ‘{ and
continue‘

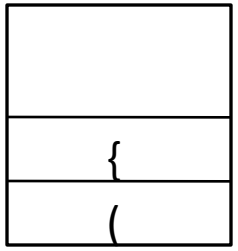


(a+b)*({c*d)
No match between closing scope ‘)’ and opening scope ‘{‘. Hence invalid.

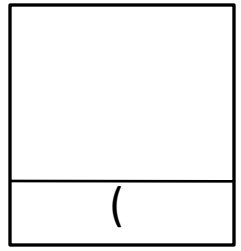
Example 4: (a+{b*c}+(c*d))



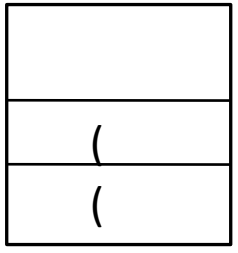
(
push ‘(‘ and
continue



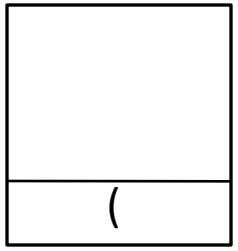
(a+{
push‘{‘ and
continue



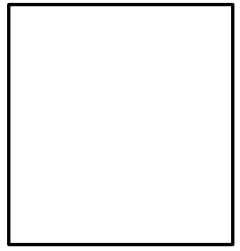
(a+{b*c}
pop‘{‘ and
continue



(a+{b*c}+(
push ‘(and
continue‘



(a+{b*c}+(c*d)
Pop ‘(‘



(a+{b*c}+(c*d))
Pop ‘(‘.
End of string and stack
empty. Hence valid

Arithmetic Expression Evaluation

- The stack organization is very effective in evaluating arithmetic expressions.
- Expressions may be represented in:
    - Infix notation
    - Polish notation (prefix notation)
    - Reverse Polish notation(postfix notation)

Arithmetic Expression Evaluation

- Infix notation
  - Each operator is written between two operands. Example: A + B.
  - Distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention.
  - The order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Arithmetic Expression Evaluation

Polish notation (prefix notation)

- It refers to the notation in which the operator is placed before its two operands.
- No parentheses are required.
- Example: +AB

Reverse Polish notation(postfix notation)

- It refers to the analogous notation in which the operator is placed after its two operands.
- No parentheses is required in Reverse Polish notation.
- Example: AB+

- Input expression given in postfix form
  - How to evaluate it?

Arithmetic Expressions

Infix form

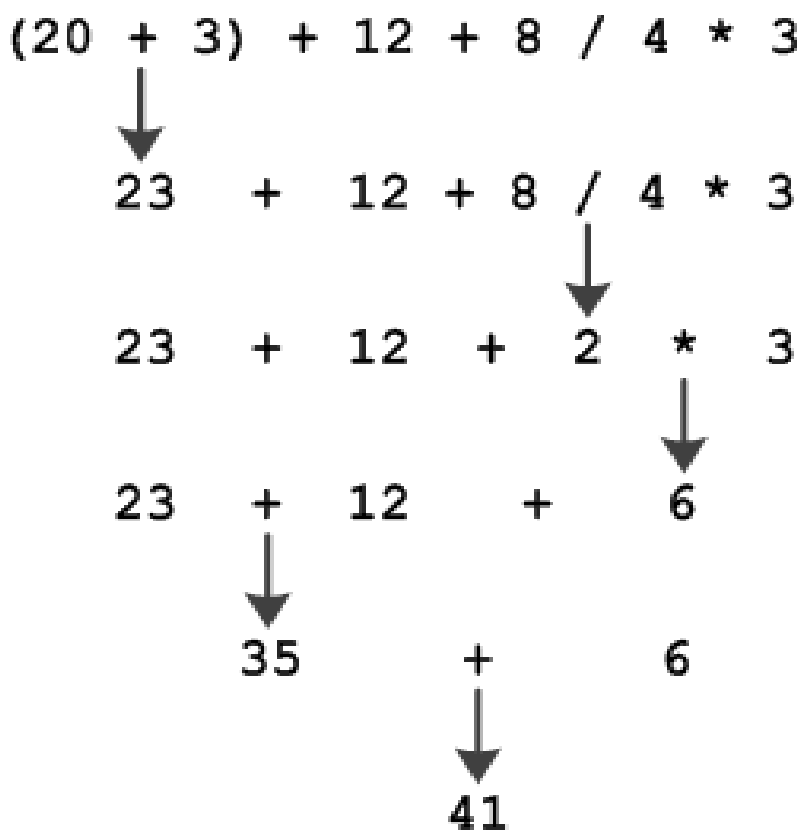
- operand *operator* operand
  - 2+3 or a+b
- Need precedence rules
- May use parentheses
  4*(3+5) or
  a*(b+c)

- **Postfix form**
  - Operator appears **after** the operands
    - (4+3)*5 : 4 3 + 5 *
    - 4+(3*5) : 4 3 5 * +
  - No precedence rules or parentheses!

Prefix Form

- Operator appears before the operands
- (4+3) : +43

Arithmetic Expression



Precedence Rules

- **Arithmetic operators** follow the same **precedence** rules as in mathematics.
- These are:
    - Exponentiation is performed first (when available).
    - Multiplication and division are performed next.
    - Addition and subtraction are performed last.

Conversion of Arithmetic Expressions

- Infix to Postfix
- Infix to Prefix
- Postfix to Infix
- Prefix to Infix
- Postfix to Prefix : Special case!
- Prefix to Prefix : Special case!

Infix to Postfix Conversion: Steps

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, put it into postfix expression.
3. Else,

    3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.

    3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Infix to Postfix Conversion: Steps

4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.
6. Repeat steps 2-6 until infix expression is scanned.
7. Pop and output from the stack until it is not empty.

Arithmetic Expressions

- Convert the following expression:

3+4*5/6

Infix to Prefix Conversion: Steps

Expression = **(A+B^C)*D+E^5**

Step 1: Reverse the infix expression.

5^E+D*)C^B+A(

Step 2: Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3: Convert expression to postfix form.(While evaluating the operators with same precedence need not be popped out of the stack)

5E^DCB^A+*+

Step 4: Reverse the expression.

C++ Operation Priorities

Binary Operators

1. * / %
2. + -
3. < <= > >=
4. == !=
5. &&
6. ||
7. =

Unary Operators

1. -y
2. +y
3. +-y = -y
4. --y = y

- Use a different symbol to represent unary operators.

Question

- Represent the quadratic formula in infix notation.
- Convert it into prefix notation.
- Convert it into postfix notation.

Postfix to Infix Conversion: Steps

- While there are input symbol left
  - Read the next symbol from the input.
- If the **symbol is an operand**
  - Push it onto the stack
- Otherwise, **if the symbol is an operator.**
  - Pop the 2 top values from the stack. (top1=op1, top2=op2, op2 *operator* op1)
- Put the operator with the values as arguments and form a string.
- Push the resulted string back to stack.
- If there is only one value in the stack, that value in the stack is the desired infix string.

Prefix to Infix Conversion: Steps

- Read the Prefix expression in reverse order (from right to left).
- If the **symbol is an operand**, then push it onto the Stack.
- If the **symbol is an operator**, then pop two operands from the Stack.
- Create a string by concatenating the two operands and the operator between them.

string = (top1(op1)+ *operator* + top2(op2))

- Push the resultant string back to Stack.
- Repeat the above steps until end of Prefix expression.

+a/*bc-de = (a+((b*c)/(d-e)))

NOTE: Read the prefix string in reverse order.

SYMBOL	STACK
e	e
d	e l d
-	(d-e)
c	(d-e) l c
b	(d-e) l c l b
*	(d-e) l (b*c)
/	((b*c)/(d-e))
a	((b*c)/(d-e)) l a
+	(a+((b*c)/(d-e)))

Hence, the equivalent infix expression: (a+((b*c)/(d-e)))

Evaluating Postfix Expressions

- Use a **stack;** assume binary operators +,*
- Input: Postfix Expression
- Scan the input.
    - If operand,
        - **push** to stack.
    - If operator,
        - **Pop** the stack twice.
        - Apply ***operator***.
        - **Push** result back to stack.

Evaluate: 456*+

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Evaluation of Postfix Expression

- Evaluate: abc*+d-
- a = 4, b= 3, c = 2, d = 5

Operator/Operand	Action	Stack
4	Push	4
3	Push	4, 3
2	Push	4, 3, 2
*	Pop (2, 3) and 3*2 = 6 then Push 6	4, 6
+	Pop (6, 4) and 4+6 = 10 then Push 10	10
5	Push	10, 5
-	Pop (5, 10) and 10-5 = 5 then Push 5	5

Example

- **Input**

  5 9 8 + 4 6 * * 7 + *

- **Evaluation**

  push(5)

  push(9)

  push(8)

  push(pop() + pop())

  /* be careful of '-' */

  push(4)

What is the answer?

push(6)

push(pop() * pop())

push(7)

push(pop() + pop())

push(pop() * pop())

print(pop())

Exercise

- Input

  6 5 2 3 + 8 * + 3 + *

- Input

  a b c * + d e * f + g * +

- For each of the previous inputs, find the infix expression.

- 6 5 2 3 - 8 * + 3 + *

Books

- Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, Fundamentals of Data structures in C (2e), Silicon Press, 2008.
- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C++ (2e), Galgotia Publications, 2008.