Data Structures & Algorithms

Prepared by: Mohamed Ayman

Machine Learning Researcher

spring 2019


sw.eng.MohamedAyman@gmail.com


facebook.com/sw.eng.MohamedAyman


linkedin.com/in/eng-mohamed-ayman


codeforces.com/profile/Mohamed_Ayman

Stack

Agenda

1- Stack Definition

2- Stack Representation

3- Basic Operations

4- Push Operation

5- Pop Operation

6- Top Method

7- Empty Method


Let's
START UP

Agenda

1- Stack Definition

2- Stack Representation

3- Basic Operations

4- Push Operation

5- Pop Operation

6- Top Method

7- Empty Method

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).

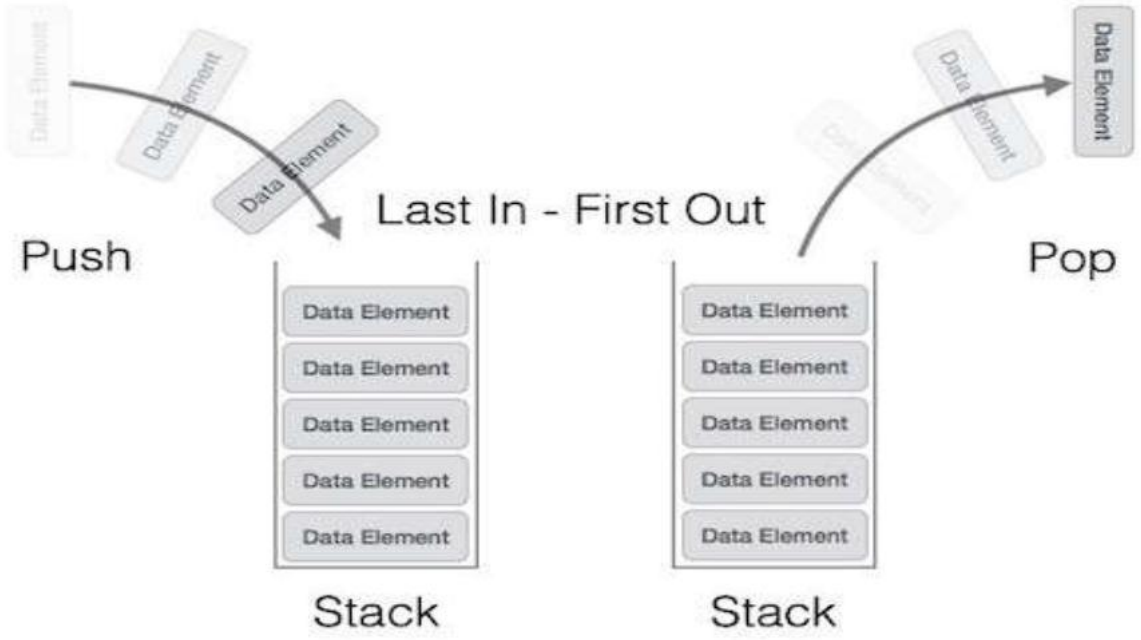
- A stack is a basic data structure that can be logically thought of as a linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it.

Agenda

- ✓ 1- Stack Definition
- **2- Stack Representation**
- 3- Basic Operations
- 4- Push Operation
- 5- Pop Operation
- 6- Top Method
- 7- Empty Method

Stack Representation

The following diagram depicts a stack



Stack Node in C++

Link: repl.it/repls/LovingAbleInterchangeability

```
4   // A stack node
5   struct node {
6       int data;
7       node* next;
8   };
9
10  node* head;
```

Agenda

- ✓ 1- Stack Definition
- ✓ 2- Stack Representation
- 3- Basic Operations
- 4- Push Operation
- 5- Pop Operation
- 6- Top Method
- 7- Empty Method

Basic Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it.
- Apart from these basic stuffs, a stack is used for the following two primary operations:

  1- push: pushing (storing) an element on the stack.

  2- pop: removing (accessing) an element from the stack.

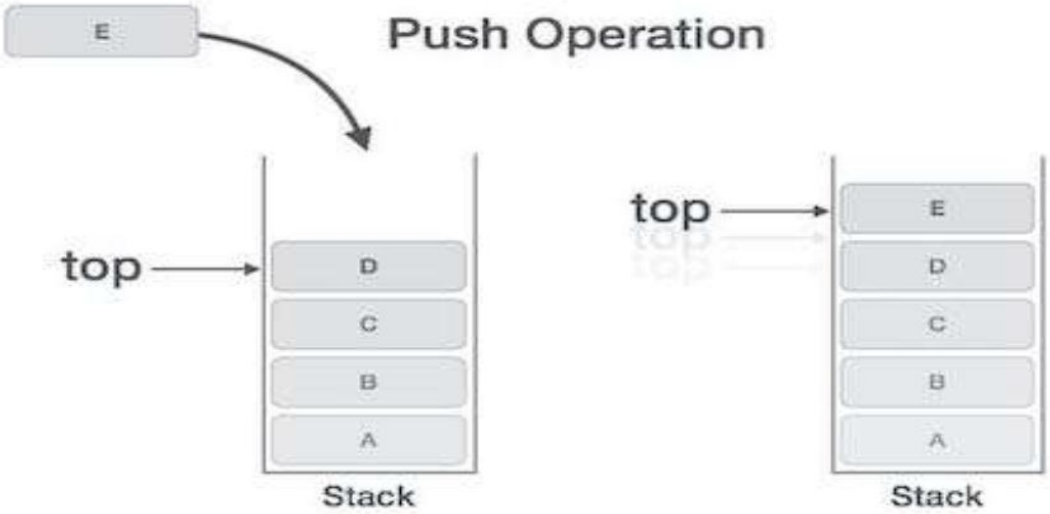
Agenda

- ✓ 1- Stack Definition
- ✓ 2- Stack Representation
- ✓ 3- Basic Operations
- **4- Push Operation**
- 5- Pop Operation
- 6- Top Method
- 7- Empty Method

Push Operation

The process of putting a new data element onto stack is known as a Push Operation.

Push operation involves a series of steps:



Push Operation in C++

Link: repl.it/repls/LovingAbleInterchangeability

```
12  // This function add node at begin of stack
13  void push(int new_data) { // O(1)
14      // allocate new node and put it's data
15      node* new_node = new node();
16      new_node->data = new_data;
17      // check if the stack is empty
18      if(head == NULL) {
19          head = new_node;
20      }
21      else {
22          // make next of new node as head
23          new_node->next = head;
24          // make the newNode as a head
25          head = new_node;
26      }
27  }
```

Agenda

- ✓ 1- Stack Definition
- ✓ 2- Stack Representation
- ✓ 3- Basic Operations
- ✓ 4- Push Operation
- **5- Pop Operation**
- 6- Top Method
- 7- Empty Method

Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation.

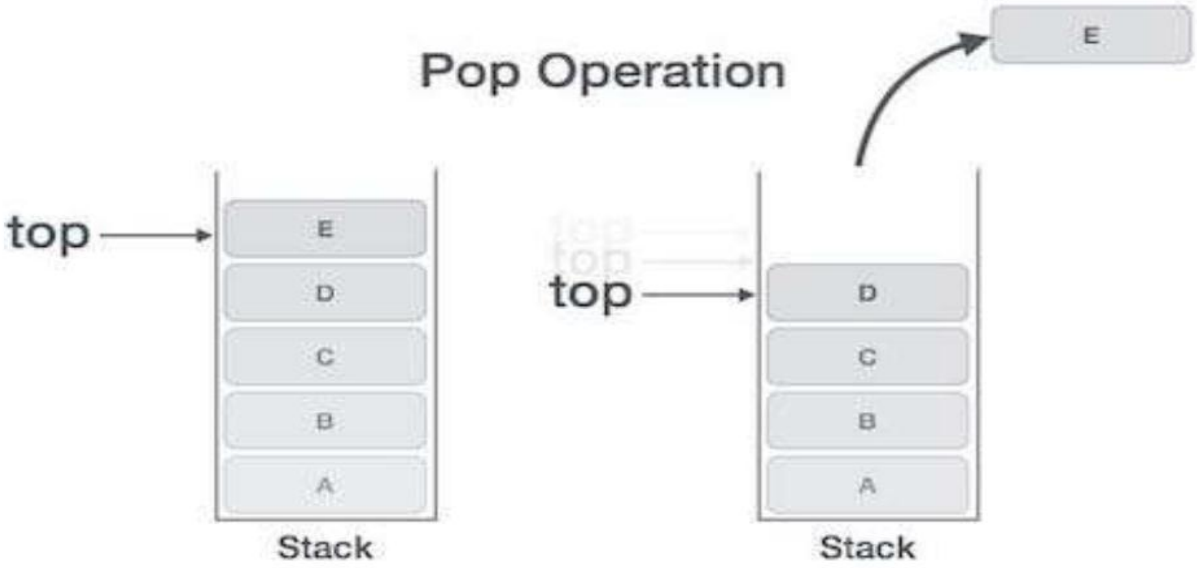
- In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.

But in linked-list implementation, pop() actually removes data element and de-allocates memory space.

Pop Operation

- The process of deleting data element at stack is known as a Pop Operation.

Pop operation involves a series of steps:



Pop Operation in C++

Link: repl.it/repls/LovingAbleInterchangeability

```
29  // This function delete first node in stack
30  void pop() { // O(1)
31      // check if the stack is empty
32      if(head == NULL)
33          return;
34      // get node which will be deleted
35      node* deleted_node = head;
36      head = head->next;
37      // delete node
38      delete(deleted_node);
39  }
```

Agenda

- ✓ 1- Stack Definition
- ✓ 2- Stack Representation
- ✓ 3- Basic Operations
- ✓ 4- Push Operation
- ✓ 5- Pop Operation
- **6- Top Method**
- 7- Empty Method

Top Method

Link: repl.it/repls/LovingAbleInterchangeability

```
41   // This function return value first node in stack
42   node* top() { // O(1)
43       return head;
44   }
```

Agenda

- ✓ 1- Stack Definition
- ✓ 2- Stack Representation
- ✓ 3- Basic Operations
- ✓ 4- Push Operation
- ✓ 5- Pop Operation
- ✓ 6- Top Method
- 7- Empty Method

Empty Method

Link: repl.it/repls/LovingAbleInterchangeability

```
46  // This function check stack is empty
47  bool empty() { // O(1)
48      return (head == NULL);
49  }
```

Agenda

- ✓ 1- Stack Definition
- ✓ 2- Stack Representation
- ✓ 3- Basic Operations
- ✓ 4- Push Operation
- ✓ 5- Pop Operation
- ✓ 6- Top Method
- ✓ 7- Empty Method


DO
MORE.

Practice

Practice

1- Reverse string using stack

2- Check string is palindrome or not

3- Convert Infix Expression to Postfix Expression

4- Convert Infix Expression to Prefix Expression

5- Convert Postfix Expression to Infix Expression

6- Convert Prefix Expression to Infix Expression

7- Convert Postfix Expression to Prefix Expression

8- Convert Prefix Expression to Postfix Expression

9- Evaluation of Postfix Expression

10- Reverse a stack using recursion

Practice

11- Check for balanced parentheses in an expression

12- Length of the longest valid substring

13- Minimum number of bracket reversals needed to make an expression balanced

14- Next Greater Element

15- Delete middle element of a stack

16- Reverse individual words

17- Largest Rectangular Area in a Histogram

18- Find maximum depth of nested parenthesis in a string

19- Expression contains redundant bracket or not

20- Check if two expressions with brackets are same

Practice

21- Delete consecutive same words in a sequence

22- Remove brackets from an algebraic string

23- Range Queries for Longest Correct Bracket Subsequence

24- Check if stack elements are pairwise consecutive

25- Reverse a number using stack

26- Tracking current Maximum Element in a Stack

27- Decode a string recursively encoded as count followed by substring

28- Find maximum difference between nearest left and right smaller elements

29- Find if an expression has duplicate parenthesis or not

30- Find index of closing bracket for a given opening bracket in an expression


Let's
START UP

Practice

1- Reverse string using stack

2- Check string is palindrome or not

3- Convert Infix Expression to Postfix Expression

4- Convert Infix Expression to Prefix Expression

5- Convert Postfix Expression to Infix Expression

6- Convert Prefix Expression to Infix Expression

7- Convert Postfix Expression to Prefix Expression

8- Convert Prefix Expression to Postfix Expression

9- Evaluation of Postfix Expression

10- Reverse a stack using recursion

Reverse string using stack

- Implement function which reverse string using stack and print result
- Function Name: reverse
- Parameters: str => string which we will process on it
- Return: string, a string reversal of parameter string
- Test Cases: abcdefg => gfedcba

Reverse string using stack

Link: repl.it/repls/DeafeningUnrulyControlpanel

```
4   // A stack based function to reverese a string
5   void reverse(string str) {
6       stack<char> stk;
7       // Push all characters of string to stack
8       for (int i = 0; i < str.size(); i++)
9           stk.push(str[i]);
10      // Pop all characters of string and put them back to str
11      for (int i = 0; i < str.size(); i++) {
12          str[i] = stk.top();
13          stk.pop();
14      }
15      cout << str << '\n';
16  }
```

Check string is palindrome or not

- Implement function which check if string is palindrome or not using stack
- Function Name: is palindrome
- Parameters: str => string which we will process on it
- Return: boolean, True if string is palindrome, False otherwise
- Test Cases: abcdedcba => palindrome

  abcdef => not palindrome

  klmnnmlk => palindrome

Check string is palindrome or not

Link: repl.it/repls/WorriedNauticalAssignment

```
4   // A stack based function to
5   // check a string is palindrome
6   bool is_palindrome(string str) {
7       stack<char> stk;
8       // Push all characters of string to stack
9       for (int i = 0; i < str.size(); i++)
10          stk.push(str[i]);
11      // Pop all characters of string at stack
12      // and compare them with characters of string
13      for (int i = 0; i < str.size(); i++) {
14          if(stk.top() != str[i])
15              return false;
16          stk.pop();
17      }
18      return true;
19  }
```

Convert Infix Expression to Postfix Expression

- Implement function which convert infix expression to postfix expression
- Function Name: infix to postfix
- Parameters: str => string which we will process on it
- Return: string result
- Test Cases: a + b * (c ^ d - e) ^ (f + g * h) - i => a b c d ^ e - f g h * + ^ * + i -

Convert Infix Expression to Postfix Expression

- Infix: The expression of the form a op b. When an operator is in-between every pair of operands.

- Postfix: The expression of the form a b op. When an operator is followed for every pair of operands.

- Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left.

- The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

- The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix (or prefix) form before evaluation.

Consider the below expression: a + b * c + d

The corresponding expression in postfix form is: a b c * d + +

The postfix expressions can be evaluated easily using a stack.

Convert Infix Expression to Postfix Expression

Algorithm

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

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 the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.

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.

Convert Infix Expression to Postfix Expression

Link: repl.it/repls/AccomplishedSilverRoutes

```
4   //Function to return precedence of operators
5   int prec(char c) {
6       if(c == '^')
7           return 3;
8       else if(c == '*' || c == '/')
9           return 2;
10      else if(c == '+' || c == '-')
11          return 1;
12      else
13          return -1;
14  }
```

Convert Infix Expression to Postfix Expression

Link: repl.it/repls/AccomplishedSilverRoutes

```
16  // The main function to convert infix expression to postfix expression
17  string infix_to_postfix(string s) {
18      stack<char> stk;
19      stk.push('N');
20      string t;
21
22      for(int i = 0; i < s.size(); i++) {
23          // If the scanned character is an operand, add it to output string.
24          if( (s[i] >= 'a' && s[i] <= 'z') || (s[i] >= 'A' && s[i] <= 'Z') )
25              t += s[i];
26          // If the scanned character is an '(', push it to the stack.
27          else if(s[i] == '(')
28              stk.push('(');
```

Convert Infix Expression to Postfix Expression

Link: repl.it/repls/AccomplishedSilverRoutes

```
29          // If the scanned character is an ‘)’, pop and to output string
30          // from the stack until an ‘(‘ is encountered.
31          else if(s[i] == ')') {
32              while(stk.top() != 'N' && stk.top() != '(') {
33                  t += stk.top();
34                  stk.pop();
35              }
36              if(stk.top() == '(')
37                  stk.pop();
38          }
39          //If an operator is scanned
40          else {
41              while(stk.top() != 'N' && prec(s[i]) <= prec(stk.top())) {
42                  t += stk.top();
43                  stk.pop();
44              }
45              stk.push(s[i]);
46          }
47      }
```

Convert Infix Expression to Postfix Expression

Link: repl.it/repls/AccomplishedSilverRoutes

```
48      //Pop all the remaining elements from the stack
49      while(stk.top() != 'N') {
50          t += stk.top();
51          stk.pop();
52      }
53      return t;
54  }
```

Convert Infix Expression to Prefix Expression

- Implement function which convert infix expression to prefix expression
- Function Name: infix to prefix
- Parameters: str => string which we will process on it
- Return: string result
- Test Cases:

```
A*B+C/D    =>    +*AB/CD
(A-B/C)*(A/K-L)    =>    *-A/BC-/AKL
```

Convert Infix Expression to Prefix Expression

- To convert an infix to postfix expression. We use the same to convert Infix to Prefix.

- Step 1: Reverse the infix expression i.e A + B * C will become C * B + A.

Note while reversing each (will become) and each) become (.

- Step 2: Obtain the postfix expression of the modifies expression i.e C B * A +

- Step 3: Reverse the postfix expression. hence in our example prefix is + A * B C

Convert Infix Expression to Prefix Expression

Link: repl.it/repls/LimitedFlatDirectory

```
4   //Function to return precedence of operators
5   int prec(char c) {
6       if(c == '^')
7           return 3;
8       else if(c == '*' || c == '/')
9           return 2;
10      else if(c == '+' || c == '-')
11          return 1;
12      else
13          return -1;
14  }
```

Convert Infix Expression to Prefix Expression

Link: repl.it/repls/LimitedFlatDirectory

```
16  // The main function to convert infix expression to postfix expression
17  string infix_to_postfix(string s) {
18      stack<char> stk;
19      stk.push('N');
20      string t;
21
22      for(int i = 0; i < s.size(); i++) {
23          // If the scanned character is an operand, add it to output string.
24          if( (s[i] >= 'a' && s[i] <= 'z') || (s[i] >= 'A' && s[i] <= 'Z') )
25              t += s[i];
26          // If the scanned character is an '(', push it to the stack.
27          else if(s[i] == '(')
28              stk.push('(');
```

Convert Infix Expression to Prefix Expression

Link: repl.it/repls/LimitedFlatDirectory

```
29          // If the scanned character is an ‘)’, pop and to output string
30          // from the stack until an ‘(‘ is encountered.
31          else if(s[i] == ')') {
32              while(stk.top() != 'N' && stk.top() != '(') {
33                  t += stk.top();
34                  stk.pop();
35              }
36              if(stk.top() == '(')
37                  stk.pop();
38          }
39          //If an operator is scanned
40          else {
41              while(stk.top() != 'N' && prec(s[i]) <= prec(stk.top())) {
42                  t += stk.top();
43                  stk.pop();
44              }
45              stk.push(s[i]);
46          }
47      }
```

Convert Infix Expression to Prefix Expression

Link: repl.it/repls/LimitedFlatDirectory

```
48      //Pop all the remaining elements from the stack
49      while(stk.top() != 'N') {
50          t += stk.top();
51          stk.pop();
52      }
53      return t;
54  }
```

Convert Infix Expression to Prefix Expression

Link: repl.it/repls/LimitedFlatDirectory

```
56  // The main function to convert infix expression to prefix expression
57  string infix_to_prefix(string s) {
58      // Reverse String
59      reverse(s.begin(), s.end());
60      // Replace ( with ) and vice versa
61      for (int i = 0; i < s.size(); i++) {
62          if (s[i] == '(') {
63              s[i] = ')';
64          }
65          else if (s[i] == ')') {
66              s[i] = '(';
67          }
68      }
69      // Get Postfix Reverse Postfix
70      string prefix = infix_to_postfix(s);
71      // Reverse postfix
72      reverse(prefix.begin(), prefix.end());
73      return prefix;
74  }
```

Convert Postfix Expression to Infix Expression

- Implement function which convert postfix expression to infix expression
- Function Name: postfix to infix
- Parameters: str => string which we will process on it
- Return: string result
- Test Cases: a b c ++ => (a + (b + c))

  a b * c + => ((a * b) + c)

Convert Postfix Expression to Infix Expression

Algorithm

1.While there are input symbol left

...1.1 Read the next symbol from the input.

2.If the symbol is an operand

...2.1 Push it onto the stack.

3.Otherwise,

...3.1 the symbol is an operator.

...3.2 Pop the top 2 values from the stack.

...3.3 Put the operator, with the values as arguments and form a string.

...3.4 Push the resulted string back to stack.

4.If there is only one value in the stack

...4.1 That value in the stack is the desired infix string.

Convert Postfix Expression to Infix Expression

Link: repl.it/repls/ThriftyLustrousDebuggers

```
4   // Convert postfix to infix expression
5   string postfix_to_infix(string exp) {
6       stack<string> stk;
7       for (int i = 0; i < exp.size(); i++) {
8           // check if symbol is operand
9           if (exp[i] >= 'a' && exp[i] <= 'z' || exp[i] >= 'A' && exp[i] <= 'Z') {
10              stk.push(string(1, exp[i]));
11          }
12          else { // We assume that input is a valid postfix and expect an operator.
13              string op1 = stk.top();
14              stk.pop();
15              string op2 = stk.top();
16              stk.pop();
17              stk.push("(" + op2 + exp[i] + op1 + ")");
18          }
19      }
20      // There must be a single element in stack now which is the required infix.
21      return stk.top();
22  }
```

Convert Prefix Expression to Infix Expression

- Implement function which convert prefix expression to infix expression
- Function Name: prefix to infix
- Parameters: str => string which we will process on it
- Return: string result
- Test Cases:

```
*+AB-CD  => ((A+B)*(C-D))
*-A/BC-/AKL  => ((A-(B/C))*((A/K)-L))
```

Convert Prefix Expression to Infix Expression

Algorithm for Prefix to Infix:

- 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 = (operand1 + operator + operand2)

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

Convert Prefix Expression to Infix Expression

Link: repl.it/repls/SilverHoarseTelephone

```
4   // Convert prefix to infix expression
5   string prefix_to_infix(string exp) {
6       stack<string> stk;
7       for (int i = exp.size() - 1; i >= 0; i--) {
8           // check if symbol is operand
9           if (exp[i] >= 'a' && exp[i] <= 'z' || exp[i] >= 'A' && exp[i] <= 'Z') {
10              stk.push(string(1, exp[i]));
11          }
12          else { // We assume that input is a valid prefix and expect an operator.
13              string op1 = stk.top();
14              stk.pop();
15              string op2 = stk.top();
16              stk.pop();
17              stk.push("(" + op1 + exp[i] + op2 + ")");
18          }
19      }
20      // There must be a single element in stack now which is the required infix.
21      return stk.top();
22  }
```

Convert Postfix Expression to Prefix Expression

- Implement function which convert postfix expression to prefix expression
- Function Name: postfix to prefix
- Parameters: str => string which we will process on it
- Return: string result
- Test Cases:

A B + C D - * => * + A B - C D

A B C / - A K / L - * => * - A / B C - / A K L

Convert Postfix Expression to Prefix Expression

Algorithm for Prefix to Postfix:

- *Read the Postfix expression from left to right*
- *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 before them.

  string = operator + operand2 + operand1

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

Convert Postfix Expression to Prefix Expression

Link: repl.it/repls/RealTragicBundledsoftware

```
4   // Convert postfix to prefix expression
5   string postfix_to_prefix(string exp) {
6       stack<string> stk;
7       for (int i = 0; i < exp.size(); i++) {
8           // check if symbol is operand
9           if (exp[i] >= 'a' && exp[i] <= 'z' || exp[i] >= 'A' && exp[i] <= 'Z') {
10              stk.push(string(1, exp[i]));
11          }
12          else { // We assume that input is a valid prefix and expect an operator.
13              string op1 = stk.top();
14              stk.pop();
15              string op2 = stk.top();
16              stk.pop();
17              stk.push(exp[i] + op2 + op1);
18          }
19      }
20      // There must be a single element in stack now which is the required expression.
21      return stk.top();
22  }
```

Convert Prefix Expression to Postfix Expression

- Implement function which convert prefix expression to postfix expression
- Function Name: prefix to postfix
- Parameters: str => string which we will process on it
- Return: string result
- Test Cases:

```
*+AB-CD  => AB+CD-*
*-A/BC-/AKL  => ABC/-AK/L-*
```

Convert Prefix Expression to Postfix Expression

Algorithm for Prefix to Postfix:

- 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 after them.
  string = operand1 + operand2 + operator
  And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

Convert Prefix Expression to Postfix Expression

Link: repl.it/repls/AustereKeenMethods

```
4   // Convert prefix to postfix expression
5   string prefix_to_postfix(string exp) {
6       stack<string> stk;
7       for (int i = exp.size() - 1; i >= 0; i--) {
8           // check if symbol is operand
9           if (exp[i] >= 'a' && exp[i] <= 'z' || exp[i] >= 'A' && exp[i] <= 'Z') {
10              stk.push(string(1, exp[i]));
11          }
12          else { // We assume that input is a valid prefix and expect an operator.
13              string op1 = stk.top();
14              stk.pop();
15              string op2 = stk.top();
16              stk.pop();
17              stk.push(op1 + op2 + exp[i]);
18          }
19      }
20      // There must be a single element in stack now which is the required expression.
21      return stk.top();
22  }
```

Evaluation of Postfix Expression

- Implement function which evaluate postfix expression
- Function Name: evaluate postfix
- Parameters: str => string which we will process on it
- Return: int, result of expression
- Test Cases: 2 3 1 * + 4 - => 1

Evaluation of Postfix Expression

- The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Following is algorithm for evaluation postfix expressions.

1) Create a stack to store operands (or values).

2) Scan the given expression and do following for every scanned element.

.....a) If the element is a number, push it into the stack

.....b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack

3) When the expression is ended, the number in the stack is the final answer

Evaluation of Postfix Expression

Link: repl.it/repls/TurboBountifulGzip

```
4   // The main function that returns value of a given postfix expression
5   void evaluate_postfix(string exp) {
6       stack<int> stk;
7       for (int i = 0; i < exp.size(); ++i) {
8           // If the scanned character is an operand, push it to the stack.
9           if (exp[i] >= '0' && exp[i] <= '9')
10              stk.push(exp[i]-'0');
11          // If the scanned character is an operator, pop two
12          // elements from stack apply the operator
13          else {
14              int val1 = stk.top();
15              stk.pop();
16              int val2 = stk.top();
17              stk.pop();
18              if(exp[i] == '+') stk.push(val2+val1);
19              if(exp[i] == '-') stk.push(val2-val1);
20              if(exp[i] == '*') stk.push(val2*val1);
21              if(exp[i] == '/') stk.push(val2/val1);
22          }
23      }
24      cout << stk.top();
25  }
```

Reverse a stack using recursion

- Implement function which reverse stack using recursion
- Function Name: reverse
- Parameters: stk => stack which we will process on it
- Return: None

Reverse a stack using recursion

Link: repl.it/repls/StrongWarpedCodeview

```
4   // Function that inserts an element at the bottom of a stack
5   void push_at_bottom(stack<int> &stk, int item) {
6       if (stk.empty()) {
7           stk.push(item);
8           return;
9       }
10      // Hold all items in Function Call Stack until we reach end
11      // of the stack. When the stack becomes empty, the above
12      // if part is executed and the item is inserted at the bottom
13      int top = stk.top();
14      stk.pop();
15      push_at_bottom(stk, item);
16      // Once the item is inserted at the bottom, push all
17      // the items held in Function Call Stack
18      stk.push(top);
19  }
```

Reverse a stack using recursion

Link: repl.it/repls/StrongWarpedCodeview

```
21  // Function that reverses the given stack
22  void reverse(stack<int> &stk) {
23      // check if stack is empty
24      if (stk.empty())
25          return;
26      // Hold all items in function call stack until we
27      // reach end of the stack
28      int item = stk.top();
29      stk.pop();
30      reverse(stk);
31      // Insert all the items (held in Function Call Stack) one by one
32      // from the bottom to top. Every item is inserted at the bottom
33      push_at_bottom(stk, item);
34  }
```

Practice

1- Reverse string using stack

2- Check string is palindrome or not

3- Convert Infix Expression to Postfix Expression

4- Convert Infix Expression to Prefix Expression

5- Convert Postfix Expression to Infix Expression

6- Convert Prefix Expression to Infix Expression

7- Convert Postfix Expression to Prefix Expression

8- Convert Prefix Expression to Postfix Expression

9- Evaluation of Postfix Expression

10- Reverse a stack using recursion


DO
MORE.

Practice

11- Check for balanced parentheses in an expression

12- Length of the longest valid substring

13- Minimum number of bracket reversals needed to make an expression balanced

14- Next Greater Element

15- Delete middle element of a stack

16- Reverse individual words

17- Largest Rectangular Area in a Histogram

18- Find maximum depth of nested parenthesis in a string

19- Expression contains redundant bracket or not

20- Check if two expressions with brackets are same

Check for balanced parentheses in an expression

- Implement function which check if string is balanced or not using stack, pairs and the orders of { , } , (,) , [,] are correct in expression
- Function Name: is balanced
- Parameters: str => string which we will process on it
- Return: boolean, True if string is balanced, False otherwise
- Test Cases: ( ) [ ] { } => balanced

  [( ]) => not balanced

  ([ ] { }) { ( ) [{ }] } => balanced

Check for balanced parentheses in an expression

Algorithm:

1) Declare a character stack S.

2) Now traverse the expression string exp.

   a) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.

   b) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.

3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

Check for balanced parentheses in an expression

Link: repl.it/repls/SizzlingUnkemptModes

```
4   // check if character1 and character2 are matching
5   bool is_matching_pair(char ch1, char ch2) {
6      return   (ch1 == '(' && ch2 == ')') ||
7               (ch1 == '{' && ch2 == '}') ||
8               (ch1 == '[' && ch2 == ']') ;
9   }
```

Check for balanced parentheses in an expression

Link: repl.it/repls/SizzlingUnkemptModes

```
11  // check if expression has balanced Parenthesis
12  bool is_balanced(string exp) {
13      stack<char> stk;
14      for (int i = 0; i < exp.size(); i++) {
15          // if the exp[i] is a starting parenthesis then push it
16          if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
17              stk.push(exp[i]);
18          // If exp[i] is a ending parenthesis then pop from stack and
19          // check if the popped parenthesis is a matching pair
20          if (exp[i] == '}' || exp[i] == ')' || exp[i] == ']') {
21              // If we see an ending parenthesis without a pair then return false
22              if (stk.empty())
23                  return false;
24              // Pop the top element from stack, if it is not a pair
25              // parenthesis of character then there is a mismatch.
26              // This happens for expressions like {(})
27              if (!is_matching_pair(stk.top(),exp[i]))
28                  return false;
29              stk.pop();
30          }
31      }
32      // If there is something left in expression then there is a starting
33      // parenthesis without a closing parenthesis
34      return stk.empty();
35  }
```

Length of the longest valid substring

- Implement function which calculate maximum length of valid substring expression
- Function Name: find max len
- Parameters: str => string which we will process on it
- Return: int, maximum length of valid substring expression
- Test Cases: ((( ) ( ) => 4 Explanation: ( ) ( )

) ((( ) ( ) (( )) => 8 Explanation: ( ) ( ) (( ))

Length of the longest valid substring

- An Efficient Solution can solve this problem in O(n) time.

- The idea is to store indexes of previous starting brackets in a stack.

- The first element of stack is a special element that provides index before beginning of valid substring (base for next valid string).

```
1) Create an empty stack and push -1 to it. The first element
   of stack is used to provide base for next valid string.

2) Initialize result as 0.

3) If the character is '(' i.e. str[i] == '(', push index
   'i' to the stack.

2) Else (if the character is ')')
   a) Pop an item from stack (Most of the time an opening bracket)
   b) If stack is not empty, then find length of current valid
      substring by taking difference between current index and
      top of the stack. If current length is more than result,
      then update the result.
   c) If stack is empty, push current index as base for next
      valid substring.

3) Return result.
```

Length of the longest valid substring

Link: repl.it/repls/WindySneakyMineral

```
4   int find_max_len(string str) {
5       stack<int> stk;
6       stk.push(-1);
7       int result = 0;
8       for (int i = 0; i < str.size(); i++) {
9           // If opening bracket, push index of it
10          if (str[i] == '(')
11              stk.push(i);
12          // If closing bracket, i.e.,str[i] = ')'
13          else {
14              stk.pop();
15              // Check if this length formed with base of
16              // current valid substring is more than max
17              if (!stk.empty())
18                  result = max(result, i - stk.top());
19              // If stack is empty. push current index as
20              // base for next valid substring
21              else
22                  stk.push(i);
23          }
24      }
25      return result;
26  }
```

Minimum number of bracket reversals needed to make an expression balanced

- Implement function which calculate minimum number of bracket reversals needed to make an expression balanced
- Function Name: count min reversals
- Parameters: str => string which we will process on it
- Return: int, minimum number of bracket reversals
- Test Cases:

```
}}{{  => 2
}{{{  => 3
{{{   => -1
}{{}}{{{ => 3
```

Minimum number of bracket reversals needed to make an expression balanced

- An Efficient Solution can solve this problem in O(n) time.
- The idea is to first remove all balanced part of expression. For example, convert } { { } } { { { to } { { { by removing highlighted part. If we take a closer look, we can notice that, after removing balanced part, we always end up with an expression of the form } } ... } { { ... { , an expression that contains 0 or more number of closing brackets followed by 0 or more numbers of opening brackets.
- How many minimum reversals are required for an expression of the form } } ... } { { ... { Let m be the total number of closing brackets and n be the number of opening brackets.
- We need ⌈m/2⌉ + ⌈n/2⌉ reversals. For example } } } } { { requires 2 + 1 reversals.

Minimum number of bracket reversals needed to make an expression balanced

Link: repl.it/repls/MemorableOverjoyedAbilities

```
4   // Returns count of minimum reversals for making expr balanced
5   // Returns -1 if expr cannot be balanced.
6   int count_min_reversals(string str) {
7       // length of expression must be even to make
8       // it balanced by using reversals.
9       if (str.size() % 2)
10          return -1;
11      // After this loop, stack contains unbalanced part of
12      // expression, i.e., expression of the form "}}..}{{..{"
13      stack<char> stk;
14      for (int i = 0; i < str.size(); i++) {
15          if (str[i] == '}' && !stk.empty() && stk.top() == '{')
16              stk.pop();
17          else
18              stk.push(str[i]);
19      }
```

Minimum number of bracket reversals needed to make an expression balanced

Link: repl.it/repls/MemorableOverjoyedAbilities

```
20      // Length of the reduced expression
21      int k = stk.size();
22      // count opening brackets at the end of stack
23      int n = 0;
24      while (!stk.empty() && stk.top() == '{') {
25          stk.pop();
26          n++;
27      }
28      int m = k - n;
29      // return ceil(m/2) + ceil(n/2)
30      return (n+1)/2 + (m+1)/2;
31  }
```

Next Greater Element

Problem Link: geeksforgeeks.org/next-greater-element

Delete middle element of a stack

Problem Link: geeksforgeeks.org/delete-middle-element-stack

Reverse individual words

- Implement function which reverse individual words
- Function Name: reverse words
- Parameters: str => string which we will process on it
- Return: None
- Test Cases: Hello World => olleH dlroW

Reverse individual words

Link: repl.it/repls/MulticoloredFixedTransversal

```
4   // Function reverses individual words of a string
5   void reverse_words(string str) {
6       stack<char> stk;
7       // Traverse given string and push all characters
8       // to stack until we see a space.
9       for (int i=0; i<str.size(); i++) {
10          if (str[i] != ' ')
11              stk.push(str[i]);
12          // When we see a space, we print contents of stack
13          else {
14              while (!stk.empty()) {
15                  cout << stk.top();
16                  stk.pop();
17              }
18              cout << ' ';
19          }
20      }
21      // Since there may not be space after last word
22      while (!stk.empty()) {
23          cout << stk.top();
24          stk.pop();
25      }
26  }
```

Largest Rectangular Area in a Histogram

Problem Link: geeksforgeeks.org/largest-rectangle-under-histogram

Find maximum depth of nested parenthesis in a string

Problem Link: geeksforgeeks.org/find-maximum-depth-nested-parenthesis-string

Expression contains redundant bracket or not

Problem Link: geeksforgeeks.org/expression-contains-redundant-bracket-not

Check if two expressions with brackets are same

Problem Link: geeksforgeeks.org/check-two-expressions-brackets

Practice

~~11- Check for balanced parentheses in an expression~~

~~12- Length of the longest valid substring~~

~~13- Minimum number of bracket reversals needed to make an expression balanced~~

~~14- Next Greater Element~~

~~15- Delete middle element of a stack~~

~~16- Reverse individual words~~

~~17- Largest Rectangular Area in a Histogram~~

~~18- Find maximum depth of nested parenthesis in a string~~

~~19- Expression contains redundant bracket or not~~

~~20- Check if two expressions with brackets are same~~


DO
MORE.

Practice

21- Delete consecutive same words in a sequence

22- Remove brackets from an algebraic string

23- Range Queries for Longest Correct Bracket Subsequence

24- Check if stack elements are pairwise consecutive

25- Reverse a number using stack

26- Tracking current Maximum Element in a Stack

27- Decode a string recursively encoded as count followed by substring

28- Find maximum difference between nearest left and right smaller elements

29- Find if an expression has duplicate parenthesis or not

30- Find index of closing bracket for a given opening bracket in an expression

Delete consecutive same words in a sequence

Problem Link: geeksforgeeks.org/delete-consecutive-words-sequence

Remove brackets from an algebraic string

Problem Link: geeksforgeeks.org/remove-brackets-algebraic-string-containing-operators

Range Queries for Longest Correct Bracket Subsequence

Problem Link: geeksforgeeks.org/range-queries-longest-correct-bracket-subsequence-set-2

Check if stack elements are pairwise consecutive

Problem Link: geeksforgeeks.org/check-if-stack-elements-are-pairwise-consecutive

Reverse a number using stack

Problem Link: geeksforgeeks.org/reverse-number-using-stack

Tracking current Maximum Element in a Stack

Problem Link: geeksforgeeks.org/tracking-current-maximum-element-in-a-stack

Decode a string recursively encoded as count followed by substring

Problem Link: geeksforgeeks.org/decode-string-recursively-encoded-count-followed-substring

Find maximum difference between nearest left and right smaller elements

Problem Link: geeksforgeeks.org/find-maximum-difference-between-nearest-left-and-right-smaller-elements

Find if an expression has duplicate parenthesis or not

Problem Link: geeksforgeeks.org/find-expression-duplicate-parenthesis-not

Find index of closing bracket for a given opening bracket in an expression

Problem Link: geeksforgeeks.org/find-index-closing-bracket-given-opening-bracket-expression

Practice

~~21- Delete consecutive same words in a sequence~~

~~22- Remove brackets from an algebraic string~~

~~23- Range Queries for Longest Correct Bracket Subsequence~~

~~24- Check if stack elements are pairwise consecutive~~

~~25- Reverse a number using stack~~

~~26- Tracking current Maximum Element in a Stack~~

~~27- Decode a string recursively encoded as count followed by substring~~

~~28- Find maximum difference between nearest left and right smaller elements~~

~~29- Find if an expression has duplicate parenthesis or not~~

~~30- Find index of closing bracket for a given opening bracket in an expression~~


DO
MORE.

Assignment

References

[01]	Online Course YouTube Playlists	https://bit.ly/2Pq88rN
[02]	Introduction to Algorithms Thomas H. Cormen	https://bit.ly/20NhuSn
[03]	Competitive Programming 3 Steven Halim	https://nus.edu/2z40vyK
[04]	Fundamental of Algorithmics Gilles Brassard and Paul Bartley	https://bit.ly/2QuvwbM
[05]	Analysis of Algorithms An Active Learning Approach	http://bit.ly/2EgCCYX
[06]	Data Structures and Algorithms Annotated Reference	http://bit.ly/2c37XEv
[07]	Competitive Programmer's Handbook	https://bit.ly/2APAbeG
[08]	GeeksforGeeks	https://geeksforgeeks.org
[09]	Codeforces Online Judge	http://codeforces.com
[10]	HackerEarth Online Judge	https://hackerearth.com
[11]	TopCoder Online Judge	https://topcoder.com

Questions ?